CONVEX SCILIB User's Guide

Second Edition

CONVEX COMPUTER CORPORATION

CONVEX SCILIB
User's Guide

Document No. 710-013630-002

Second Edition
February 1993

CONVEX Computer Corporation
Richardson, Texas USA

CONVEX SCILIB User's Guide
Order No. DSW-360
Second Edition

Printed in the United States of America

Revision information for
CONVEX SCILIB User's Guide

Edition	Document No.	Description
Second	710-013630-002	Released with CONVEX SCILIB software V8.0, February 1993. Reorganized Chapter 1. Corrected miscellaneous errors.
First	710-013630-001	Initial release with CONVEX SCILIB software V7.0, 1991.

Table of Contents

List of Tables

Permuted Index

Preface

Purpose and Audience

This guide describes the CONVEX SCILIB software library and shows you how to use it. CONVEX SCILIB is a collection of FORTRAN-callable subprograms identical in name and operation to those found in the Cray Research Incorporated's UNICOS Math and Scientific Library, V5.0. SCILIB subprograms have been optimized for use on the CONVEX family of supercomputers.

The *CONVEX SCILIB User's Guide* addresses experienced FORTRAN programmers who:

- convert programs written in Cray FORTRAN to CONVEX FORTRAN.
- optimize existing software to improve performance and increase productivity on CONVEX supercomputers.
- use CONVEX FORTRAN to develop new programs that rely heavily on matrix operations.

Familiarity with the ConvexOS operating system is helpful, but not required, to use this guide. If you are not familiar with ConvexOS, refer to the "Associated Documents" section at the end of this preface.

Organization

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

To learn more about the subject of any given chapter, refer to the literature cited in the "Supplemental Reading" section of each chapter.

To identify subprograms by function, refer to the Permuted Index which lists subprogram functions and their chapter numbers and names. To find the page number on which the subprogram is described, use the Index or refer to the "Subprogram Descriptions" section in the chapter introduction.

This guide is organized into the following chapters:

- Chapter 1 introduces general concepts about CONVEX SCILIB.
- Chapter 2 describes basic vector operations included in SCILIB.
- Chapter 3 explains basic matrix operations.
- Chapter 4 describes linear equation subprograms in SCILIB.
- Chapter 5 explains the eigenanalysis capabilities available to SCILIB users.
- Chapter 6 describes the discrete Fourier transforms in SCILIB.
- Chapter 7 describes SCILIB subprograms that compute convolutions and correlations of data sets.

- Chapter 8 describes SCILIB subprograms that deal with linear recurrences.
- Chapter 9 describes miscellaneous subprograms to sort elements of a vector and to report errors detected in the usage of SCILIB routines.
- An index is included at the back of the manual.

Notational Conventions

The following conventions are used in this manual:

- *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

  fc ***prog_name*.o**

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

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

- Type in **lowercase boldface** 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 will be a matrix and **a** will be the FORTRAN array containing the matrix:

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

- UPPERCASE CONSTANT WIDTH represents FORTRAN programs.

- Brackets ([ ]) enclose optional entries.

- Many SCILIB subprogram names are prefixed to indicate the type of data they operate on.

  For example, the subprograms to copy a vector are SCOPY and CCOPY, for REAL and COMPLEX vector types, respectively.

  Note that in Cray FORTRAN, the single precision REAL type is 64 bits (one Cray word) in length. This is essentially equivalent to the type DOUBLE PRECISION in CONVEX FORTRAN. Similarly, the Cray FORTRAN type COMPLEX is 128 bits long; this is equivalent to the CONVEX FORTRAN type DOUBLE COMPLEX. The Cray double precision versions of REAL and COMPLEX types (128 and 256 bit, respectively) are not supported in SCILIB because of processing time considerations.

Associated Documents

Using this guide successfully may require information not specific to the tasks described herein or not within the scope of this guide. The following documents are provided by CONVEX Computer Corporation to help you:

- *CONVEX VECLIB User's Guide* (DSW-132). This guide provides definitions for many additional subprograms available to SCILIB users through inclusion of CONVEX VECLIB, but not documented in the *CONVEX SCILIB User's Guide.*

- *CONVEX VECLIB Quick Reference* (DSW-134). This compact reference lists the name, purpose and usage for each VECLIB subprogram. Its organization is similar to the *CONVEX VECLIB User's Guide.*

- *CONVEX SCILIB Quick Reference* (DSW-361). This compact reference lists the name, purpose and usage for each SCILIB subprogram. Its organization is similar to the *CONVEX SCILIB User's Guide.*

- *CONVEX LAPACK User's Guide* (DSW-036). This guide provides information on the subprograms provided with the CONVEX LAPACK library.

- *CONVEX FORTRAN Language Reference Manual* (DSW-037). This manual is a reference for the CONVEX FORTRAN programming language and is designed to provide a thorough working definition of the language.

- *CONVEX FORTRAN User's Guide* (DSW-038). This guide tells you how to use the CONVEX FORTRAN compiler, including how to compile, load, and execute programs.

- *CONVEX FORTRAN Optimization Guide* (DSW-034). This guide describes methods for optimizing FORTRAN programs.

- *CONVEX Performance Analyzer (CXpa) User's Guide* (DSW-251). This guide explains the operation of the CONVEX Performance Analyzer (CXpa) and the steps needed to create and interpret a CXpa profile.

- *CONVEX Application Compiler User's Guide* (DSW-401). This guide describes the CONVEX Application Compiler and how to use it to optimize programs.

- *CONVEX Consultant User's Guide* (DSW-025). This guide describes the functions and operations of the CONVEX *csd* debugger, the post-mortem dump (*pmd*) facility, and the *prof*, *bprof*, and *gprof* profilers.

- *ConvexOS Man Pages for Users* (DSW-331). This book contains copies of Sections 1 and 7 of the online man pages. These man pages are primarily concerned with operating system information for users.

- Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart. 1979. *LINPACK Users' Guide.* Society for Industrial and Applied Mathematics, Philadelphia. This guide provides information on non-optimized LINPACK subprograms included in CONVEX SCILIB, but not documented in the *CONVEX SCILIB User's Guide.*

- Garbow, B.S., et al. 1977. "Matrix Eigensystem Routines—EISPACK Guide Extension." *Lecture Notes in Computer Science*, Vol. 51. Springer-Verlag, New York.

This guide provides information on non-optimized EISPACK subprograms included in CONVEX SCILIB, but not documented in the *CONVEX SCILIB User's Guide.*

- Smith, B.T., et al. 1976. "Matrix Eigensystem Routines—EISPACK Guide." *Lecture Notes in Computer Science,* Vol. 6, 2nd edition. Springer-Verlag, New York. This guide provides information on non-optimized EISPACK subprograms included in CONVEX SCILIB, but not documented in the *CONVEX SCILIB User's Guide.*

Ordering Documentation

To order the current edition of this or any other CONVEX document, send requests to:

CONVEX Computer Corporation
Customer Service
P.O. Box 833851
Richardson, TX 75083-3851

Include the order number (beginning with the letters "DSW" or "DHW") or the exact title, as listed on the front cover.

To order an edition other than the current edition, include the 12-digit document number, as listed on the "Revision information" page.

Technical Assistance

If you have questions that are not answered in this book, contact the CONVEX Technical Assistance Center (TAC). Use the following phone numbers:

- Within the continental U.S. call 1(800)952-0379.
- Outside the continental U.S. contact the local CONVEX office.

1

Introduction to CONVEX SCILIB

Overview

CONVEX SCILIB is a collection of FORTRAN-callable mathematical subprograms which provides a look-alike implementation of the Scientific Library portion of Cray Research Incorporated's UNICOS Math and Scientific Library, V5.0.

All CONVEX SCILIB subroutines are optimized for use on the CONVEX family of supercomputers. This general library addresses a variety of linear algebra operations on multiple data types. It contains subprograms for:

- dense vector operations
- sparse vector operations
- matrix operations
- linear equation solution
- discrete Fourier transforms
- convolution and correlation
- linear recurrences
- sorting and error reporting

Although CONVEX SCILIB was designed for use with FORTRAN programs, C programs can call CONVEX SCILIB subprograms, as described in Appendix A of the *CONVEX VECLIB User's Guide*, which is included in this documentation set.

This chapter provides information necessary for efficient use of CONVEX SCILIB, including discussions of CONVEX SCILIB software standardization, how to access CONVEX SCILIB subprograms, optimizations, including parallel processing and interactions with other CONVEX analysis and optimization products, supported floating point formats, roundoff effects, CONVEX SCILIB subprogram capabilities, how to use the library and various compiler options, error handling, online documentation, and CONVEX support services.

Chapter Objectives

After reading this chapter you will:

- know how to access CONVEX SCILIB
- understand how CONVEX SCILIB works in a parallel computing environment
- know how CONVEX SCILIB interacts with the CONVEX Performance Analyzer and other profilers, the Application Compiler, and CONVEX's two floating-point formats
- understand roundoff effects
- be able to use FORTRAN type declarations and compiler options
- understand how CONVEX SCILIB handles errors
- know how to access the online *CONVEX SCILIB Programmer's Reference*
- know what to do if you are having trouble using CONVEX SCILIB subprograms

What You Need to Know to Use CONVEX SCILIB

You should be familiar with the following information to make efficient use of CONVEX SCILIB.

Standardization

CONVEX SCILIB is designed to provide CONVEX users with a look-alike implementation of UNICOS Math and Scientific Library subroutines. This allows programs written for Cray machines to be easily ported to CONVEX, and it provides a common programmer interface between the two machines, which can ease a Cray programmer's transition to writing code for CONVEX, especially when used in conjunction with the CONVEX FORTRAN compiler's **–cfc** command line flag.

Accessing CONVEX SCILIB

The CONVEX SCILIB library consists of compiled subprograms ready for you to incorporate into your programs with the linker. Simply include the appropriate declarations and **CALL** statements in your FORTRAN source program and specify that CONVEX SCILIB be used as an object library at link time by using the **–l** option on the *fc* command line, as follows:

fc [*options*] *file* **–lscilib**

CONVEX VECLIB is documented in the *CONVEX VECLIB User's Guide.* If you use subprograms from both CONVEX SCILIB and CONVEX VECLIB, access them follows:

fc [*options*] *file* **–lscilib –lveclib8**

See "Interactions Between VECLIB, SCILIB, and LAPACK" for details about how to order the two **–l** options. Do not try to use subprograms from both **–lscilib** and **–lveclib** in the same program.

CONVEX LAPACK is documented in the *CONVEX LAPACK User's Guide.* If you use subprograms from both CONVEX SCILIB and CONVEX LAPACK, use the following:

fc [*options*] *file* **–lscilib –llapack8**

Add the linker option **–lveclib8** if CONVEX VECLIB is also used. See "Interactions Between VECLIB, SCILIB, and LAPACK" for details about the order of the **–l** options. Do not try to use subprograms from both **–lscilib** and **–llapack** in the same program.

Interactions Between VECLIB, SCILIB, and LAPACK

Each of the five library files in CONVEX VECLIB, CONVEX SCILIB, and CONVEX LAPACK is complete in itself, meaning that you will not need to load one library merely because you have used subprograms from another. This is accomplished by including various subprograms in more than one library. For example, subroutine SGEMV is in all of these products, but with identical functionality. Thus, in general, you have to load only the libraries you need, and you may list them in any order on your load command line, as described in the previous section. However, there are a few differences between the libraries that may force you to put the libraries into a specific order to obtain the results you expect.

Differences between VECLIB and LAPACK

A subprogram name conflict exists between VECLIB and LAPACK subprograms SGTSV, DGTSV, CGTSV, and ZGTSV. Both sets of subprograms solve tridiagonal systems of linear equations, but their argument lists and functionality differ. If you use these subprograms, be sure to load the ones you want by specifying their library file name first.

Differences between VECLIB8 and LAPACK8

A subprogram name conflict exists between VECLIB8 and LAPACK8 subprograms SGTSV and CGTSV. Both sets of subprograms solve tridiagonal systems of linear equations, but their argument lists and functionality differ. If you use these subprograms, be sure to load the ones you want by specifying their library file name first.

Differences between VECLIB8 and SCILIB

Five subprograms common to VECLIB8 and SCILIB differ slightly in functionality. Subprograms ICAMAX, ISAMAX, ISAMIN, ISMAX, and ISMIN in VECLIB8 handle a negative **incx** argument by taking its absolute value and searching the **x** vector in forward order, while in SCILIB, a negative **incx** argument results in searching the array **x** in backward order. No VECLIB8 subprograms call any of these subprograms with a negative **incx** argument, so you may safely load SCILIB before VECLIB8 if you need the SCILIB functionality.

Two other subprograms in both VECLIB8 and SCILIB have the same functionality but different numbers of arguments. Subroutines SGEMMS and CGEMMS from the two libraries implement Strassen's method for matrix multiplication, but the SCILIB versions have an extra argument, for working storage, that is not needed in the VECLIB8 versions. Be certain that your calls to these subprograms have 14 arguments if you load SCILIB before VECLIB8.

Differences between LAPACK8 and SCILIB

Two subprograms common to LAPACK8 and SCILIB differ slightly in functionality. Subprograms ICAMAX and ISAMAX in LAPACK8 handle a negative **incx** argument by taking its absolute value and searching the **x** vector in forward order, while in SCILIB, a negative **incx** argument results in searching the array **x** in backward order. No LAPACK8 subprograms call either of these subprograms with a negative **incx** argument, so you may safely load SCILIB before LAPACK8 if you need the SCILIB functionality.

Performance Value

As computer architectures have become more complicated, it has become more important to know the architecture of the target computer to maximize program performance. When a program is moved from one computer to another, architectural considerations on which the program was based may no longer be valid. If, however, the computationally intensive part of the program is based on highly tuned subprograms from a vendor-supplied library, the vendor's knowledge of the architecture is transferred to the program. CONVEX SCILIB provides this feature, enabling you to achieve good performance at low cost.

Optimization

Keep in mind that while CONVEX SCILIB subroutines are identical in name and purpose to subroutines found in the UNICOS Math and Scientific Library, they have been optimized for use on CONVEX machines, and this required somewhat different implementations. Since the two machines use different architectures and different methods of carrying out various mathematical operations in hardware and software, CONVEX SCILIB subroutines cannot be expected to give

identical answers to their Cray counterparts in all cases. However, CONVEX SCILIB subroutines have been tested to insure that they give essentially equivalent answers in all circumstances.

CONVEX SCILIB takes full advantage of the CONVEX tightly integrated vector, scalar, and parallel architecture. Most CONVEX SCILIB subprograms have been coded in assembly language, but they are completely compatible with standard FORTRAN programs. Because you can easily incorporate these kernels into the computationally intensive parts of programs, you receive the performance benefits of highly tuned assembly language without having to become an expert in the CONVEX architecture or assembly-language programming.

Parallel Processing

Parallel processing is available on CONVEX C2 and C3 Series supercomputers with multiple processors. 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.

To support parallel processing, the CONVEX C2 and C3 Series hardware employs automatic self-allocating processors (ASAP) that effectively manage CPU assignment. When another processor can be used to assist in executing a process, the active thread posts a request for any idle CPU(s) to help. New threads are created and executed by available CPUs. If no CPUs are available, the original thread executes all the work for the process.

CONVEX SCILIB works in both single processor (CONVEX C1 Series) and parallel processor (CONVEX C2 and C3 Series) environments. At run time, a CONVEX SCILIB subprogram determines whether it is being used on a CONVEX C1, C2, or C3 Series processor. If it is running on a CONVEX C2 or C3 Series processor with multiple heads, it detects if the program is already using multiple threads. It uses this information to automatically choose between a single or parallel processor algorithm. Consequently, you can move programs that use CONVEX SCILIB between CONVEX C1, C2, and C3 Series systems freely, without losing compatibility or the advantages of either architecture.

If you are computing on a CONVEX C2 or C3 Series system with multiple processors, you can realize the performance benefits of parallel processing in two ways. First, you can simply call any parallelized CONVEX SCILIB subprogram and let it use parallelism internally. Alternatively, you can call CONVEX SCILIB subprograms in a parallelized loop or region. To obtain parallelism via the second mechanism, you need to be familiar with the concept of reentrancy and with the FORCE_PARALLEL, BEGIN_TASKS, NEXT_TASK, and END_TASKS compiler directives.

CONVEX SCILIB subprograms are reentrant, which means 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 CONVEX SCILIB subprograms in a parallelized loop or region. The compiler does not automatically parallelize loops containing a function reference or subroutine call. You can force it to parallelize such a loop by inserting a FORCE_PARALLEL compiler directive before the loop. For example, the following FORTRAN code makes parallel calls to subprogram CLUSEQ:

```
C$DIR FORCE_PARALLEL
      DO 10 J=1,N
         CALL CLUSEQ (N, X, I, A(J), INDX(I,J), NINDX(J))
   10 CONTINUE
```

While optimizing a parallel program, you might want to make parallel calls to a CONVEX SCILIB subprogram to perform independent operations, but where the call statements are not in a loop. The FORTRAN compiler does not automatically parallelize code outside a loop, but you can use the BEGIN_TASKS, NEXT_TASK, and END_TASKS compiler directives to tell the compiler to parallelize such code. For example, the following FORTRAN code makes parallel

calls to subprogram **INTMAX**:

```
C$DIR BEGIN_TASKS
      IX = INTMAX(NX,X,1)
C$DIR NEXT_TASK
      IY = INTMAX(NY,Y,1)
C$DIR END_TASKS
```

For more information on compiler directives, including usage cautions and warnings, refer to the *CONVEX FORTRAN Language Reference Manual* and the *CONVEX FORTRAN Optimization Guide.*

Profiling CONVEX SCILIB Applications

The CONVEX Performance Analyzer, CXpa, is an interactive tool that gathers and analyzes program execution timing (profiling) data. CXpa provides the programmer with the means to study the timing behavior of a program for the purposes of optimizing, benchmarking, and debugging. To use the performance analyzer, you must first compile your FORTRAN program with either the **–pa**, **–pab**, or **–par** compiler option. These options instrument the compiled program so that its performance can be measured at the subprogram level, the loop level, the block level, or the region level.

CONVEX SCILIB has been instrumented at the subprogram level so that the performance of CONVEX SCILIB subprograms can be included in the analysis. This instrumentation is nonintrusive, so it is not necessary to use a different version of CONVEX SCILIB when you desire to profile your program. Also, the CXpa instrumentation does not interfere with the *prof*, *bprof*, or *gprof* instrumentation in your program. However, you may not profile your program with both CXpa and *prof*, *bprof*, or *gprof* at the same time.

Subprogram-level profiling produces summary information about the subprograms that are called during profiled execution of the program. This information includes:

- the number of times each subprogram is called
- the CPU time in each subprogram and the percentage of the program total, either including or excluding the cumulative time in called subprograms
- a dynamic call graph, listing the subprogram calls that take place within a computer program

CXpa is an optional product. For more information about CXpa, refer to the *CONVEX Performance Analyzer (CXpa) User's Guide*, or contact your CONVEX sales representative.

Optimizing with the Application Compiler

The CONVEX Application Compiler is an interprocedural analyzer that tracks the flow of data and control between procedures. The information generated by this analysis removes scope restrictions on optimization, which allows the Application Compiler to generate more efficient code by taking the entire program, with all its dependencies, into account. The database of program information that the interprocedural analyzer builds up also allows the Application Compiler to do better error checking, leading to more robust and reliable programs. CONVEX SCILIB has been annotated to permit the Application Compiler to effectively use CONVEX SCILIB subprograms.

The CONVEX Application Compiler is an optional product. For more information about the Application Compiler, refer to the *CONVEX Application Compiler User's Guide*, or contact your CONVEX sales representative.

Floating Point Formats

C-Series CONVEX computers operate on data represented in either of two floating-point formats, called *native* format and *IEEE* format. ANSI/IEEE Standard 754 defines IEEE format, and both formats are described in the *CONVEX Architecture Reference (C Series)*.

CONVEX SCILIB operates in either floating point format by automatically determining the format the calling program is using, so you need not do anything special to incorporate CONVEX SCILIB subprograms into programs whether you compile them to use native or IEEE format. For further information on CONVEX floating point formats, refer to the *CONVEX FORTRAN User's Guide*.

Roundoff Effects

CONVEX SCILIB subprograms may use a different arithmetic order of evaluation than that employed by the UNICOS Math and Scientific Library or other mathematical software. Different roundoff characteristics may result. Accuracy of results is usually about the same, so using CONVEX SCILIB should not materially affect the accumulation of roundoff errors in a complete application program. If it does, you should examine the mathematical analysis of the problem, which will likely show that the problem 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 CONVEX SCILIB 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.

Required Data Item Byte Lengths and How to Get Them

In CONVEX SCILIB subprograms all INTEGER, REAL, and LOGICAL arguments must be 64-bit quantities, and all COMPLEX arguments must occupy 128 bits. Table 1-1 shows the correspondence between the lengths of data items declared in various ways:

Note that if the FORTRAN data types are not given length specifiers (for example, **REAL** is used instead of **REAL*8**) then any of the compiler options **–cfc**, **–p8**, or **–pd8** are compatible with the data types required by CONVEX SCILIB. On the other hand, if length specifiers are used, then the **–cfc** compiler options will override them and enforce the data types required by CONVEX SCILIB. If any DOUBLE PRECISION declarations or double precision constants occur in the program, the **–cfc** and **–pd8** compiler options causes them to be treated as 64-bit quantities. This leads us to recommend that you use either **–cfc** or **–pd8**.

CONVEX SCILIB subroutines will not accept INTEGERS or REALS of length 4 bytes, which is the default for these types in CONVEX FORTRAN. If you call CONVEX SCILIB subroutines and *do not* compile with **–cfc** or **–pd8**, you must be very careful that all variables and constants passed to CONVEX SCILIB subroutines are of the proper length.

For more information on FORTRAN data types and FORTRAN compiler options refer to the *CONVEX FORTRAN Language Reference Manual*.

Table 1-1: Data Item Byte Length vs. Declaration and Compiler Option

FORTRAN Declaration	FORTRAN Compiler Option			
	none	-cfc	-p8	-pd8
INTEGER	4	8	8	8
INTEGER*4	4	8	4	4
INTEGER*8	8	8	8	8
Integer by default	4	8	8	8
REAL	4	8	8	8
REAL*4	4	8	4	4
REAL*8	8	8	8	8
Real by default	4	8	8	8
DOUBLE PRECISION	8	16	16	8
Double Precision constant	8	16	16	8
COMPLEX	8	16	16	16
COMPLEX*8	8	16	8	8
COMPLEX*16	16	16	16	16
Complex constant	8	16	16	16
LOGICAL	4	8	8	8
LOGICAL*4	4	8	4	4
LOGICAL*8	8	8	8	8
Logical constant	4	8	8	8

Error Handling

Some CONVEX SCILIB subprograms do not have a success/error code in their argument lists, but instead call another SCILIB subprogram to process the error condition. Two error handlers are provided: XERBLA and XERSCI; these are documented in Chapter 3 and Chapter 9 of this guide, respectively. The documentation for each CONVEX SCILIB subprogram indicates if either of these error handlers is used. The standard versions of XERBLA and XERSCI write an error message onto the standard error file. If the main program is in FORTRAN, a call traceback is also written onto the standard error file. Execution is then terminated with a nonzero exit status. You may supply a version of XERBLA or XERSCI that alters this action; see the documentation for these subprograms for more information.

CONVEX SCILIB Programmer's Reference

The *CONVEX SCILIB Programmer's Reference* is online documentation that includes information from the *CONVEX SCILIB User's Guide*. This reference contains an introduction to CONVEX SCILIB and to each set of subprograms in CONVEX SCILIB, and reference entries for each subprogram. Subprogram entries include descriptions and examples of usage.

This reference is provided for users to easily and efficiently obtain online information on CONVEX SCILIB. Because of the limited number of fonts supported and the difficulty of presenting mathematical equations in the *man*(1) system, the *CONVEX SCILIB Programmer's Reference* is not a substitute for the *CONVEX SCILIB User's Guide*; the most detailed information on CONVEX SCILIB will be in the user's guide.

To access reference entries, use the ConvexOS command

man 3sci *entry_name*

For further explanation and a table of contents of reference entries, refer to the *scilib*(*3sci*) entry by typing

man 3sci scilib

Support Services

CONVEX maintains a staff to provide technical help if you have difficulty. Located in the CONVEX Technical Assistance Center (TAC), these people are the primary link between you and the company, and they stand ready to assist you with any difficulties. Note, though, that CONVEX SCILIB has been tested extensively and is very reliable. Therefore, before contacting the TAC about a CONVEX SCILIB problem, follow this procedure to isolate the cause of the trouble and to simplify the job of resolving it:

- Check any error response provided by the subprogram in question. The subprogram descriptions in this manual describe how to check an error response. If the answer is wrong because an error has been detected, correct the cause of the error and run the job again.

- Verify that the subprogram usage in the program matches the subprogram specifications in this manual. Pay special attention to the number of arguments in the **CALL** statement and to the declarations of arrays and integer constants or variables that describe them. If everything is in order, write out all the arguments immediately before and after the **CALL** statement.

- Make sure there really is a problem. For example, if an apparently incorrect answer is being computed, check to see if the answer does satisfy the problem as defined in the program. Also, for problems with more than one answer, CONVEX SCILIB may produce a different answer or give the answers in a different order than expected. If the problem is ill-conditioned, CONVEX SCILIB may not be able to compute a reliable answer at all. Again, error messages often suggest the cause of the problem.

- Isolate the problem. If possible, write a small test program that encounters the same difficulty. Perhaps data causing the problem may be written out from the original program and read into the small one. Try to remove the problem area from a large program and concentrate it in a small program. 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. Clearly, this is not always possible, but the process often leads to insight.

You will frequently discover a usage error and resolve the problem by following the steps above. If the trouble persists, contact the TAC for help. Providing a small test program and expected answers will help the TAC further analyze the problem. To report a software or documentation problem to the TAC, use the *contact* utility. The *contact* utility allows you to submit a problem or suggest an enhancement directly to the TAC from your own system.

For information about *contact*, use the ConvexOS command

man contact

Supplemental Reading

The SCILIB documentation set includes the *CONVEX VECLIB User's Guide*, the *CONVEX LAPACK User's Guide*, the *LINPACK Users' Guide*, the *EISPACK Guide*, and the *EISPACK Guide Extension*.

Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart. *LINPACK Users' Guide*. Philadelphia, PA: SIAM Publications. 1979.

Garbow, B.S., *et al.* "Matrix Eigensystem Routines—EISPACK Guide Extension." *Lecture Notes in Computer Science*, Vol. 51. New York: Springer-Verlag. 1977.

Smith, B.T., *et al.* "Matrix Eigensystem Routines—EISPACK Guide." *Lecture Notes in Computer Science*, Vol. 6, 2nd edition. New York: Springer-Verlag. 1976.

2

Basic Vector Operations

Overview

This chapter explains how to use the SCILIB vector subprograms that serve as building blocks for many user programs. It describes subprograms for performing dense and sparse vector operations, and it includes the FORTRAN equivalent of each subprogram. This set of SCILIB subprograms includes:

- the Basic Linear Algebra Subprograms (BLAS)
- Cray extensions to the BLAS

The term BLAS, as used in this section, refers to the standard BLAS operations and the Cray extensions to the BLAS.

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

What You Need to Know to Use These Subprograms

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, using SCILIB subprograms can improve program performance, as well as program modularity and readability. Note, however, that in some situations you can achieve better computational performance by entering FORTRAN code than by calling one of these subprograms.

BLAS Storage Conventions

The Basic Linear Algebra Subprograms (BLAS) were developed to enhance the portability of published linear algebra codes. In particular, LINPACK, the high-level public-domain linear equation package, uses the BLAS. Thus, if you use LINPACK from SCILIB you will normally be replacing the standard FORTRAN BLAS with the SCILIB BLAS and increasing the efficiency of LINPACK on your CONVEX supercomputer.

You need not limit your use of the SCILIB BLAS to LINPACK. Because these subprograms are portable, modular, self-documenting, and efficient, you can incorporate them into your programs.

To realize the full power of the BLAS, you must understand the following three subjects:

- FORTRAN storage of arrays
- FORTRAN array argument association
- BLAS 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 × 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. Since 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

The rest of this section describes dealing with stride arguments and handling forward and backward storage.

A vector in the BLAS is defined by three quantities:

1. The vector length.
2. The array or starting element within an array.
3. The increment, sometimes called the *stride*, which 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) $\times$ INCX). This is forward storage starting from X(1) with stride equal to INCX, ending with X(1 + (N- 1) $\times$ 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 routines 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) $\times$ |INCX|) or equivalently in X(1 - (N-I) $\times$ INCX). This is backward storage starting from X(1 - (N-1) $\times$ 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 permitted by some BLAS routines and is not permitted by others. When it 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 I is (N,B(I,1),LDB).

Examples

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

```
REAL*8 SDOT,S,X(1+(N-1)*|INCX|),Y(1+(N-1)*|INCY|)
S = SDOT (N, X,INCX, Y,INCY)
```

This sets S to the dot product of the vectors (N,X,INCX) and (N,Y,INCY).

Example 1
Compute the dot product T = X(1)*Y(1) + X(2)*Y(2) + X(3)*Y(3) + X(4)*Y(4):

```
REAL*8 SDOT,T,X(4),Y(4)
T = SDOT (4, X,1, Y,1)
```

Example 2
Compute the convolution T = X(1)*Y(4) + X(2)*Y(3) + X(3)*Y(2) + X(4)*Y(1):

```
REAL*8 SDOT,T,X(4),Y(4)
T = SDOT (4, X,1, 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), which 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*8 SDOT,A(LDA,3),X(3),Y(LDA)
N = 3
Y(2) = SDOT (N, A(2,1),LDA, X,1)
```

Supplemental Reading

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.

Subprogram Descriptions

CLUSEQ/.../CLUSILT Find Clusters of Selected Vector Elements

Purpose

Given a real or integer vector x of length n, these subprograms search sequentially through the vector and fill an array with the beginning and ending indices i of the maximal contiguous groups (clusters) of elements which satisfy a specified relationship with a given scalar a.

A cluster is a set of one or more elements $\{x_i, x_{i+1}, \ldots, x_j\}$, with the following properties:

1) for every k with $i \leq k \leq j$, x_k satisfies the specified relationship with a,
2) either $i = 1$ or x_{i-1} does not satisfy the relationship, and
3) either $j = n$ or x_{j+1} does not satisfy the relationship.

At most, there are $\lceil n/2 \rceil$ clusters, where $\lceil x \rceil$ represents the smallest integer greater than or equal to x.

The last two characters of the subprogram name specify the relationship of interest between the elements of the vector and the scalar. These characters and the corresponding cluster relationship may be

xx	Cluster relationship
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	$\{i : x_i \neq a\}$

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

Usage

SCILIB:

```
INTEGER*8 n, incx, indx(2, (n+1)/2), nindx
REAL*8    x(lenx), a
CALL CLUSEQ (n, x, incx, a, indx, nindx)

INTEGER*8 n, x(lenx), incx, a, indx(2, (n+1)/2), nindx
CALL CLUSEQ (n, x, incx, a, indx, nindx)

INTEGER*8 n, incx, indx(2, (n+1)/2), nindx
REAL*8    x(lenx), a
CALL CLUSNE (n, x, incx, a, indx, nindx)

INTEGER*8 n, x(lenx), incx, a, indx(2, (n+1)/2), nindx
CALL CLUSNE (n, x, incx, a, indx, nindx)

INTEGER*8 n, incx, indx(2, (n+1)/2), nindx
REAL*8    x(lenx), a
CALL CLUSFxx (n, x, incx, a, indx, nindx)

INTEGER*8 n, x(lenx), incx, a, indx(2, (n+1)/2), nindx
CALL CLUSIxx (n, x, incx, a, indx, nindx)
```

Continued CLUSEQ/CLUSNE/CLUSFGE/CLUSFGT/.../CLUSILT

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**:

incx ≥ 0 x is stored forward in array **x**, i.e., x_i is stored in **x**((*i*–1)× **incx**+1).
incx < 0 x is stored backward in array **x**, i.e., x_i is stored in **x**((*i*–**n**)× **incx**+1).

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

a The scalar a.

Output

indx **indx**(1,*k*) and **indx**(2,*k*) contain the indices of the beginning and ending elements of the *k*th cluster of x respectively. Only the first **nindx** elements of **indx** are changed.

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

Notes

These subprograms are sometimes useful for optimizing a loop containing an **IF** statement.

FORTRAN Equivalent

```
      SUBROUTINE CLUSEQ (N,X,INCX,A,INDX,NINDX)
      INTEGER*8 N,X(*),INCX,A,INDX(2,*),NINDX
      LOGICAL*8 INCLUS
      IX = 1
      IF ( INCX .LT. 0 ) IX = 1 - (N-1) * INCX
      NINDX = 0
      INCLUS = .FALSE.
      DO 10 I = 1, N
         IF ( .NOT. INCLUS ) THEN
            IF ( X(IX) .EQ. A ) THEN
               NINDX = NINDX + 1
               INDX(1,NINDX) = I
               INCLUS = .TRUE.
            END IF
         ELSE
            IF ( X(IX) .NE. A ) THEN
               INDX(2,NINDX) = I-1
               INCLUS = .FALSE.
            END IF
         END IF
         IX = IX + INCX
   10 CONTINUE
      IF ( INCLUS ) INDX(2,NINDX) = N
      RETURN
      END
```

CLUSEQ/CLUSNE/CLUSFGE/CLUSFGT/.../CLUSILT Continued

Example Find the clusters 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*8 N,INCX,INDX(2,11),NINDX
REAL*8    A,X(20)
N = 10
INCX = 1
A = 0.0
CALL CLUSGT (N,X,INCX,A,INDX,NINDX)
```

Gather Sparse Vector — GATHER

Purpose

Given a dense real vector y stored in full storage form, and a set of indices of *interesting* elements of y, this subprogram gathers 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

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

Usage

SCILIB:

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

Input

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

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

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

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

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

Output

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

Notes

Cray Research, Inc. has declared this subprogram obsolete in release 6.0 of the UNICOS Math and Scientific Library.

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

IILZ — Count Initial Zero Elements

Purpose

Given an integer vector x of length n, this subprogram counts the number of zero elements of x before the first nonzero element. Given a logical vector x, IILZ counts the number of .FALSE. elements of x before the first .TRUE. element.

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

Usage

SCILIB:

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

INTEGER*8 i, IILZ, n, incx
LOGICAL*8 x(lenx)
i = IILZ (n, x, incx)
```

Input

n Number of elements of vector x. If **n** $\leq$ 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**:

incx $\geq$ 0 x is stored forward in array **x**, i.e., x_i is stored in **x**((*i*–1)× **incx**+1).

incx $<$ 0 x is stored backward in array **x**, i.e., x_i is stored in **x**((*i*–**n**)× **incx**+1).

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

Output

i If **n** $\leq$ 0, then **i** = 0. If **n** $>$ 0 and all elements of x are zero or .TRUE., then **i** = **n**. Otherwise, **i** is the number of the zero or .FALSE. elements x_i of x before the first nonzero or .TRUE. element.

FORTRAN Equivalent

```
      INTEGER*8 FUNCTION IILZ (N,X,INCX)
      INTEGER*8 N,X(*),INCX
      IILZ = 0
      IX = 1
      IF ( INCX .LT. 0 ) IX = 1 - (N-1) * INCX
      DO 10 I = 1, N
         IF ( X(IX) .NE. 0 ) RETURN
         IX = IX + INCX
         IILZ = I
   10 CONTINUE
      RETURN
      END
```

Example

Determine the number of initial zero elements of an INTEGER*8 vector x, where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20.

```
INTEGER*8 I,IILZ,N,X(20),INCX
N = 10
INCX = 1
I = IILZ (N,X,INCX)
```

Count Initial Positive Elements — ILLZ

Purpose

Given a vector x of length n, this subprogram counts the number of positive elements of x before the first negative element. In this context, positive means that the leftmost or high-order bit is zero, and negative means that the leftmost bit is one.

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

Usage

SCILIB:

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

Input

n Number of elements of vector x. If **n** $\leq$ 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**:

incx $\geq$ 0 x is stored forward in array **x**, i.e., x_i is stored in **x**((i–1)× **incx**+1).

incx $<$ 0 x is stored backward in array **x**, i.e., x_i is stored in **x**((i–**n**)× **incx**+1).

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

Output

i If **n** $\leq$ 0, then **i** = 0. If **n** $>$ 0 and all elements of x are zero or .TRUE., then **i** = **n**. Otherwise, **i** is the number of the positive elements x_i of x before the first negative element.

FORTRAN Equivalent

```
      INTEGER*8 FUNCTION ILLZ (N,X,INCX)
      INTEGER*8 N,X(*),INCX
      ILLZ = 0
      IX = 1
      IF ( INCX .LT. 0 ) IX = 1 - (N-1) * INCX
      DO 10 I = 1, N
         IF ( X(IX) .LT. 0 ) RETURN
         IX = IX + INCX
         ILLZ = I
   10 CONTINUE
      RETURN
      END
```

Example

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

```
INTEGER*8 I,ILLZ,N,X(20),INCX
N = 10
INCX = 1
I = ILLZ (N,X,INCX)
```

ILSUM — Count TRUE Vector Elements

Purpose

Given a logical vector x of length n, this subprogram counts the number of elements of the vector that have the logical value .TRUE.

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

Usage

SCILIB:

```
INTEGER*8 i, ILSUM, n, incx
LOGICAL*8 x(lenx)
i = ILSUM (n, x, incx)
```

Input

n Number of elements of vector x. 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**:

incx ≥ 0 x is stored forward in array **x**, i.e., x_i is stored in **x**((*i*–1)× **incx**+1).

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

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., 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 number of elements of x that have the logical value .TRUE.

FORTRAN Equivalent

```
      INTEGER*8 FUNCTION ILSUM (N,X,INCX)
      INTEGER*8 N,INCX
      LOGICAL*8 X(*)
      ILSUM = 0
      IX = 1
      IF ( INCX .LT. 0 ) IX = 1 - (N-1) * INCX
      DO 10 I = 1, N
         IF ( X(IX) ) ILSUM = ILSUM + 1
         IX = IX + INCX
   10 CONTINUE
      RETURN
      END
```

Example

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

```
INTEGER*8  I,ILSUM,N,INCX
LOGICAL*8  X(20)
N = 10
INCX = 1
I = ILSUM (N,X,INCX)
```

Index of Maximum Element of Vector — INFLMAX

Purpose

Given a vector x of length n, these subprograms determine the index of the first element x_i in which a specified group of bits attains its maximum value in the vector. Specifically, the subprograms determine the smallest index i such that

$$\mathbf{AND(SHIFTR}(x_i,rshift),mask) = \max\left(\mathbf{AND(SHIFTR}(x_j,rshift),mask) : j = 1, 2, \ldots, n\right).$$

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

Usage

SCILIB:

```
INTEGER*8 i, INFLMAX, n, x(lenx), incx, mask, rshift
i = INFLMAX (n, x, incx, mask, rshift)
```

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**:

incx ≥ 0 x is stored forward in array **x**, i.e., x_i is stored in **x**((*i*–1)× **incx**+1).

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

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

mask Mask of 1-bits to extract desired group of bits from the shifted elements of x with a bitwise logical product operation. Refer to "Purpose."

rshift Number of bits by which to right shift each element of x so as to align the specified group of bits with **a**, 0 ≤ **rshift** ≤ 63. Refer to "Purpose."

Output

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

INFLMAX Continued

FORTRAN Equivalent

```
      INTEGER*8 FUNCTION INFLMAX (N,X,INCX,MASK,RSHIFT)
      INTEGER*8 N,X(*),INCX,MASK,RSHIFT,TEMP,XMAX
      INFLMAX = 1
      IF ( N .GT. 1 ) THEN
         IX = 1 + INCX
         IF ( INCX .LT. 0 ) IX = 1 - (N-2) * INCX
         XMAX = AND(SHIFTR(X(IX-INCX),RSHIFT),MASK)
         DO 10 I = 2, N
            TEMP = AND(SHIFTR(X(IX),RSHIFT),MASK)
            IF ( TEMP .GT. XMAX ) THEN
               INFLMAX = I
               XMAX = TEMP
            END IF
            IX = IX + INCX
   10    CONTINUE
      ELSE IF ( N .LT. 1 ) THEN
         INFLMAX = 0
      END IF
      RETURN
      END
```

Example Locate the element of an INTEGER*8 vector x in which the field consisting of the rightmost 8 bits achieves its maximum value, where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20.

```
      INTEGER*8 I,INFLMAX,N,X(20),INCX,MASK,RSHIFT
      N = 10
      INCX = 1
      MASK = 'FF'X
      RSHIFT = 0
      I = INFLMAX (N,X,INCX,MASK,RSHIFT)
```

Index of Minimum Element of Vector — INFLMIN

Purpose

Given a vector x of length n, these subprograms determine the index of the first element x_i in which a specified group of bits attains its minimum value in the vector. Specifically, the subprograms determine the smallest index i such that

$$\mathbf{AND(SHIFTR}(x_i,rshift),mask) = \min\left(\mathbf{AND(SHIFTR}(x_j,rshift),mask) : j = 1, 2, \ldots, n\right).$$

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

Usage

SCILIB:

```
INTEGER*8 i, INFLMIN, n, x(lenx), incx, mask, rshift
i = INFLMIN (n, x, incx, mask, rshift)
```

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**:

incx ≥ 0	x is stored forward in array **x**, i.e., x_i is stored in **x**((*i*–1)× **incx**+1).
incx < 0	x is stored backward in array **x**, i.e., x_i is stored in **x**((*i*–**n**)× **incx**+1).

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

mask Mask of 1-bits to extract desired group of bits from the shifted elements of x with a bitwise logical product operation. Refer to "Purpose."

rshift Number of bits by which to right shift each element of x so as to align the specified group of bits with **a**, 0 ≤ **rshift** ≤ 63. Refer to "Purpose."

Output

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

INFLMIN Continued

FORTRAN Equivalent

```
      INTEGER*8 FUNCTION INFLMIN (N,X,INCX,MASK,RSHIFT)
      INTEGER*8 N,X(*),INCX,MASK,RSHIFT,TEMP,XMIN
      INFLMIN = 1
      IF ( N .GT. 1 ) THEN
         IX = 1 + INCX
         IF ( INCX .LT. 0 ) IX = 1 - (N-2) * INCX
         XMIN = AND(SHIFTR(X(IX-INCX),RSHIFT),MASK)
         DO 10 I = 2, N
            TEMP = AND(SHIFTR(X(IX),RSHIFT),MASK)
            IF ( TEMP .LT. XMIN ) THEN
               INFLMIN = I
               XMIN = TEMP
            END IF
            IX = IX + INCX
10       CONTINUE
      ELSE IF ( N .LT. 1 ) THEN
         INFLMIN = 0
      END IF
      RETURN
      END
```

Example Locate the element of an INTEGER*8 vector x in which the field consisting of the rightmost 8 bits achieves its minimum value, where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20.

```
INTEGER*8 I,INFLMIN,N,X(20),INCX,MASK,RSHIFT
N = 10
INCX = 1
MASK = 'FF'X
RSHIFT = 0
I = INFLMIN (N,X,INCX,MASK,RSHIFT)
```

Index of Maximum of Magnitudes — ISAMAX/ICAMAX

Purpose

Given a real or integer vector x of length n, ISAMAX 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 determines the smallest index i such that

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

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. If the index i is used for pivot selection in matrix factorization, no significant difference in numerical stability should result.

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

Usage

SCILIB:

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

INTEGER*8    i, ICAMAX, n, incx
COMPLEX*16 x(lenx)
i = ICAMAX (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**:

incx ≥ 0 x is stored forward in array **x**, i.e., x_i is stored in **x**((*i*–1)×**incx**+1).

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

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., 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.

ISAMAX/ICAMAX — Continued

Notes The handling of **incx** < 0 differs between these subprograms and ISAMAX/ICAMAX in VECLIB.

FORTRAN Equivalent

```
      INTEGER*8 FUNCTION ISAMAX (N,X,INCX)
      INTEGER*8 N,INCX
      REAL*8 X(*),TEMP,XMAX
      ISAMAX = 1
      IF ( N .GT. 1 ) THEN
         IX = 1 + INCX
         IF ( INCX .LT. 0 ) IX = 1 - (N-2) * INCX
         XMAX = ABS ( X(IX - INCX) )
         DO 10 I = 2, N
            TEMP = ABS ( X(IX) )
            IF ( TEMP .GT. XMAX ) THEN
               ISAMAX = I
               XMAX = TEMP
            END IF
            IX = IX + INCX
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*8 I,ISAMAX,N,INCX
      REAL*8    X(20)
      N = 10
      INCX = 1
      I = ISAMAX (N,X,INCX)
```

Index of Minimum of Magnitudes — ISAMIN

Purpose

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

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

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

Usage

SCILIB:

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

Input

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

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**, i.e., x_i is stored in **x**((*i*–1)× **incx**+1).

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

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., 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.

Notes

The handling of **incx** < 0 differs between ISAMIN in SCILIB and VECLIB.

FORTRAN Equivalent

```
      INTEGER*8 FUNCTION ISAMIN (N,X,INCX)
      INTEGER*8 N,INCX
      REAL*8 X(*),TEMP,XMIN
      ISAMIN = 1
      IF ( N .GT. 1 ) THEN
         IX = 1 + INCX
         IF ( INCX .LT. 0 ) IX = 1 - (N-2) * INCX
         XMIN = ABS ( X(IX-INCX) )
         DO 10 I = 2, N
            TEMP = ABS ( X(IX) )
            IF ( TEMP .LT. XMIN ) THEN
               ISAMIN = I
               XMIN = TEMP
            END IF
            IX = IX + INCX
   10    CONTINUE
      ELSE IF ( N .LT. 1 ) THEN
         ISAMIN = 0
      END IF
      RETURN
      END
```

ISAMIN Continued

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*8 I,ISAMIN,N,INCX
REAL*8    X(20)
N = 10
INCX = 1
I = ISAMIN (N,X,INCX)
```

Index of Maximum Element of Vector — ISMAX/INTMAX

Purpose

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

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

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

Usage

SCILIB:

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

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

incx ≥ 0 x is stored forward in array **x**, i.e., x_i is stored in **x**((*i*–1)×**incx**+1).

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

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., 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.

Notes

The handling of **incx** < 0 differs between ISMAX in SCILIB and VECLIB.

ISMAX/INTMAX Continued

FORTRAN Equivalent

```
      INTEGER*8 FUNCTION ISMAX (N,X,INCX)
      INTEGER*8 N,INCX
      REAL*8 X(*),XMAX
      ISMAX = 1
      IF ( N .GT. 1 ) THEN
         IX = 1 + INCX
         IF ( INCX .LT. 0 ) IX = 1 - (N-2) * INCX
         XMAX = X(IX - INCX)
         DO 10 I = 2, N
            IF ( X(IX) .GT. XMAX ) THEN
               ISMAX = I
               XMAX = X(IX)
            END IF
            IX = IX + INCX
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*8 I,ISMAX,N,INCX
      REAL*8    X(20)
      N = 10
      INCX = 1
      I = ISMAX (N,X,INCX)
```

Index of Minimum Element of Vector — ISMIN/INTMIN

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

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

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

Usage

SCILIB:

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

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

Input

n Number of elements of vector x to be used. If **n** $\leq$ 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**:

incx $\geq$ 0 x is stored forward in array **x**, i.e., x_i is stored in **x**((*i*–1)× **incx**+1).

incx $<$ 0 x is stored backward in array **x**, i.e., x_i is stored in **x**((*i*–**n**)× **incx**+1).

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

Output

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

Notes

The handling of **incx** $<$ 0 differs between ISMIN in SCILIB and VECLIB.

ISMIN/INTMIN Continued

FORTRAN Equivalent

```
      INTEGER*8 FUNCTION ISMIN (N,X,INCX)
      INTEGER*8 N,INCX
      REAL*8 X(*),XMIN
      ISMIN = 1
      IF ( N .GT. 1 ) THEN
         IX = 1 + INCX
         IF ( INCX .LT. 0 ) IX = 1 - (N-2) * INCX
         XMIN = X(IX - INCX)
         DO 10 I = 2, N
           IF ( X(IX) .LT. XMIN ) THEN
              ISMIN = I
              XMIN = X(IX)
           END IF
           IX = IX + INCX
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*8 I,ISMIN,N,INCX
      REAL*8    X(20)
      N = 10
      INCX = 1
      I = ISMIN (N,X,INCX)
```

Search Vector for Element — ISRCHEQ/ISRCHNE/.../ISRCHILT

Purpose

Given a real or integer vector x of length n, these subprograms search sequentially through the vector for the first element x_i that satisfies a specified relationship with 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. These characters and the corresponding function values may 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	$\min\{i : x_i \neq a\}$

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

Usage

SCILIB:

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

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

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

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

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

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

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

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

ISRCHEQ/ISRCHNE/ISEARCH/ISRCHFGE/.../ISRCHILT Continued

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**:

incx ≥ 0 x is stored forward in array **x**, i.e., x_i is stored in **x**((*i*–1)× **incx**+1).

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

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., 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. If **n** > 0 and no element of x satisfies the relationship with **a** specified by the subprogram name, then **i** = **n** + 1. 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.

FORTRAN Equivalent

```
INTEGER*8 FUNCTION ISRCHEQ (N,X,INCX,A)
INTEGER*8 N,X(*),INCX,A
ISRCHEQ = 0
IF ( N .LE. 0 ) RETURN
IX = 1
IF ( INCX .LT. 0 ) IX = 1 - (N-1) * INCX
DO 10 I = 1, N
   IF ( X(IX) .EQ. A ) THEN
      ISRCHEQ = I
      RETURN
   END IF
   IX = IX + INCX
10 CONTINUE
   ISRCHEQ = N+1
   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*8 I,ISRCHFGT,N,INCX
REAL*8    X(20),A
N = 10
INCX = 1
A = 0.0
I = ISRCHFGT (N,X,INCX,A)
```

Search Vector for Element — ISRCHMEQ/ISRCHMGE/.../ISRCHMNE

Purpose

Given a vector x of length n, these subprograms search sequentially through the vector for the first element x_i which contains a specified group of bits that satisfies a specified relationship with 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. These characters and the corresponding function values, may be

xx	Function value
EQ	$\min\{i : \mathbf{AND(SHIFTR}(x_i,rshift),mask) = a\}$
GE	$\min\{i : \mathbf{AND(SHIFTR}(x_i,rshift),mask) \geq a\}$
GT	$\min\{i : \mathbf{AND(SHIFTR}(x_i,rshift),mask) > a\}$
LE	$\min\{i : \mathbf{AND(SHIFTR}(x_i,rshift),mask) \leq a\}$
LT	$\min\{i : \mathbf{AND(SHIFTR}(x_i,rshift),mask) < a\}$
NE	$\min\{i : \mathbf{AND(SHIFTR}(x_i,rshift),mask) \neq a\}$

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

Usage

SCILIB:

```
INTEGER*8 i, ISRCHMxx, n, x(lenx), incx, a, mask, rshift
i = ISRCHMxx (n, x, incx, a, mask, rshift)
```

Input

n Number of elements of vector x to be compared to **a**. If **n** $\leq$ 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**:

incx $\geq$ 0 x is stored forward in array **x**, i.e., x_i is stored in **x**((i–1)× **incx**+1).

incx $<$ 0 x is stored backward in array **x**, i.e., x_i is stored in **x**((i–**n**)× **incx**+1).

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

a The scalar a.

mask Mask of 1-bits to extract desired group of bits from the shifted elements of x with a bitwise logical product operation. Refer to "Purpose."

rshift Number of bits by which to right shift each element of x so as to align the specified group of bits with **a**, 0 $\leq$ **rshift** $\leq$ 63. Refer to "Purpose."

Output

i If **n** $\leq$ 0, then **i** = 0. If **n** $>$ 0 and no element of x satisfies the relationship with **a** specified by the subprogram name, then **i** = **n** + 1. 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.

ISRCHMEQ/ISRCHMGE/ISRCHMGT/.../ISRCHMNE Continued

FORTRAN Equivalent

```
      INTEGER*8 FUNCTION ISRCHMEQ (N,X,INCX,A,MASK,RSHIFT)
      INTEGER*8 N,X(*),INCX,A,MASK,RSHIFT
      ISRCHMEQ = 0
      IF ( N .LE. 0 ) RETURN
      IX = 1
      IF ( INCX .LT. 0 ) IX = 1 - (N-1) * INCX
      DO 10 I = 1, N
         IF ( AND(SHIFTR(X(IX),RSHIFT),MASK) .EQ. A ) THEN
            ISRCHMEQ = I
            RETURN
         END IF
         IX = IX + INCX
   10 CONTINUE
      ISRCHMEQ = N+1
      RETURN
      END
```

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

```
INTEGER*8 I,ISRCHMGT,N,X(20),INCX,A,MASK,RSHIFT
N = 10
INCX = 1
A = 1
MASK = 1
RSHIFT = 0
I = ISRCHMEQ (N,X,INCX,A,MASK,RSHIFT)
```

Search Ordered Vector for Element — OSRCHF/OSRCHI

Purpose

Given an ordered real or integer vector x of length n, these subprograms search sequentially through the vector for the first element x_i that equals a given scalar a and return the index i of that element. They also return the number of elements of the vector that are equal to the scalar, and the index of the location within the vector where the scalar should fit in the array, whether they find it or not.

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

Usage

SCILIB:

```
INTEGER*8 n, incx, indx, ifound, iwould, icount
REAL*8    x(lenx), a
CALL OSRCHF (n, x, incx, a, ifound, iwould, icount)

INTEGER*8 n, x(lenx), incx, a, ifound, iwould, icount
CALL OSRCHI (n, x, incx, a, ifound, iwould, icount)
```

Input

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

x Array of length **lenx** = (**n**–1)×|**incx**|+1 containing the n-vector x. The elements of x are assumed to be in ascending order.

incx Increment for the array **x**:

incx $\geq$ 0 x is stored forward in array **x**, i.e., x_i is stored in **x**((i–1)×**incx**+1).
incx $<$ 0 x is stored backward in array **x**, i.e., x_i is stored in **x**((i–**n**)×**incx**+1).

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

a The scalar a.

icount Flag indicating whether to count the number of occurrences of **a** in x:

icount $\neq$ 0 do count the number of occurrences of **a** in x.
icount = 0 do not count the number of occurrences of **a** in x.

Output

ifound If **n** $\leq$ 0, then **ifound** = 0. If **n** $>$ 0 and no element of x equals **a**, then **ifound** = **nf** + 1. Otherwise, **ifound** is the index i of the first element x_i of x that equals **a**.

iwould If **n** $\leq$ 0, then **iwould** = 0. If **n** $>$ 0, then **iwould** is the index i of the first element x_i of x such that $x_i \leq$ **a** $\leq x_{i-1}$. If **a** is found in x, then **iwould** = **ifound**.

icount If **icount** $\neq$ 0 on entry, the number of occurrences of **a** in x. Not used as output if **icount** = 0.

Notes

No check is made to ensure that the elements of x are in ascending order. The output is undefined if the elements are unordered.

OSRCHF/OSRCHI Continued

FORTRAN Equivalent

```
      SUBROUTINE ORSCHF (N,X,INCX,A,IFOUND,IWOULD,ICOUNT)
      INTEGER*8 N,INCX,IFOUND,IWOULD,ICOUNT
      REAL*8 X(*),A
      IF ( N .LE. 0 ) RETURN
         IFOUND = 0
         IWOULD = 0
         IF ( ICOUNT .NE. 0 ) ICOUNT = 0
         RETURN
      END IF
      IX = 1
      IF ( INCX .LT. 0 ) IX = 1 - (N-1) * INCX
      DO 20 I = 1, N
         IF ( X(IX) .GE. A ) THEN
            IF ( X(IX) .EQ. A ) THEN
               IFOUND = I
               IWOULD = I
               IF ( ICOUNT .NE. 0 ) THEN
                  ICOUNT = 1
                  IX = IX + INCX
                  DO 10 J = I+1, N
                     IF ( X(IX) .EQ. A ) THEN
                        ICOUNT = ICOUNT + 1
                        IX = IX + INCX
                     ELSE
                        RETURN
                     END IF
   10             CONTINUE
               END IF
            ELSE
               IFOUND = N+1
               IWOULD = I
               IF ( ICOUNT .NE. 0 ) ICOUNT = 0
            END IF
            RETURN
         END IF
         IX = IX + INCX
   20 CONTINUE
      IFOUND = N+1
      IWOULD = N+1
      IF ( ICOUNT .NE. 0 ) ICOUNT = 0
      RETURN
      END
```

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

```
      INTEGER*8 N,INCX,IFOUND,IWOULD,ICOUNT
      REAL*8    X(20),A
      N = 10
      INCX = 1
      A = 10.0
      CALL OSRCHF (N,X,INCX,A,IFOUND,IWOULD,ICOUNT)
```

Search Ordered Vector for Element — OSRCHM

Purpose

Given an ordered integer vector x of length n, this subprogram searches sequentially through the vector for the first element x_i which contains a specified group of bits that equals a given scalar a and returns the index i of that element. The index is such that

$$\mathbf{AND(SHIFTR}(x_i,rshift),mask) = a$$

It also returns the number of elements of the vector that are equal to the scalar, and the index of the location within the vector where the scalar should fit in the array, whether it finds it or not.

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

Usage

SCILIB:

```
INTEGER*8 n, x(lenx), incx, a, mask, rshift, ifound, iwould, icount
CALL OSRCHM (n, x, incx, a, mask, rshift, ifound, iwould, icount)
```

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. The elements of x are assumed to be in ascending order.

incx Increment for the array **x**:

incx ≥ 0 x is stored forward in array **x**, i.e., x_i is stored in **x**((i–1)× **incx**+1).
incx < 0 x is stored backward in array **x**, i.e., x_i is stored in **x**((i–**n**)× **incx**+1).

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

a The scalar a.

mask Mask of 1-bits to extract desired group of bits from the shifted elements of x with a bitwise logical product operation. Refer to "Purpose."

rshift Number of bits by which to right shift each element of x so as to align the specified group of bits with **a**, 0 ≤ **rshift** ≤ 63. Refer to "Purpose."

icount Flag indicating whether to count the number of occurrences of **a** in x:

icount ≠ 0 do count the number of occurrences of **a** in x.
icount = 0 do not count the number of occurrences of **a** in x.

Output

ifound If **n** ≤ 0, then **ifound** = 0. If **n** > 0 and no element of x equals **a**, then **ifound** = **n** + 1. Otherwise, **ifound** is the index i of the first element x_i of x that equals **a**.

iwould If **n** ≤ 0, then **iwould** = 0. If **n** > 0, then **iwould** is the index i of the first element x_i of x such that $x_i \le \mathbf{a} \le x_{i+1}$. If **a** is found in x, then **iwould** = **ifound**.

icount If **icount** ≠ 0 on entry, the number of occurrences of **a** in x. Not used as output if **icount** = 0.

Notes No check is made to ensure that the specified group of bits of the elements of x are in ascending order. The output is undefined if the elements are unordered.

FORTRAN Equivalent

```
      SUBROUTINE ORSCHM (N,X,INCX,A,MASK,RSHIFT,IFOUND,IWOULD,ICOUNT)
      INTEGER*8 N,X(*),INCX,A,IFOUND,IWOULD,ICOUNT
      IF ( N .LE. 0 ) THEN
         IFOUND = 0
         IWOULD = 0
         IF ( ICOUNT .NE. 0 ) ICOUNT = 0
         RETURN
      END IF
      IX = 1
      IF ( INCX .LT. 0 ) IX = 1 - (N-1) * INCX
      DO 20 I = 1, N
         IF ( AND(SHIFTR(X(IX),RSHIFT),MASK) .GE. A ) THEN
            IF ( AND(SHIFTR(X(IX),RSHIFT),MASK) .EQ. A ) THEN
               IFOUND = I
               IWOULD = I
               IF ( ICOUNT .NE. 0 ) THEN
                  ICOUNT = 1
                  IX = IX + INCX
                  DO 10 J = I+1, N
                     IF ( AND(SHIFTR(X(IX),RSHIFT),MASK) .EQ. A ) THEN
                        ICOUNT = ICOUNT + 1
                        IX = IX + INCX
                     ELSE
                        RETURN
                     END IF
10                CONTINUE
               END IF
            ELSE
               IFOUND = N+1
               IWOULD = I
               IF ( ICOUNT .NE. 0 ) ICOUNT = 0
            END IF
            RETURN
         END IF
         IX = IX + INCX
20    CONTINUE
      IFOUND = N+1
      IWOULD = N+1
      IF ( ICOUNT .NE. 0 ) ICOUNT = 0
      RETURN
      END
```

Example Search for the first element of an INTEGER*8 vector x such that the second group of eight bits from the right contain the value 13, where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20.

```
INTEGER*8 N,X(20),INCX,A,MASK,RSHIFT,IFOUND,IWOULD,ICOUNT
N = 10
INCX = 1
A = 13
MASK = 'FF'X
RSHIFT = 8
CALL OSRCHM (N,X,INCX,A,MASK,RSHIFT,IFOUND,IWOULD,ICOUNT)
```

SASUM/SCASUM — Sum of Magnitudes

Purpose

Given a real or integer vector x of length n, SASUM computes the l_1 norm of x, i.e., 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 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

$$t = \|x\|_1 = \sum_{i=1}^{n} \left\{ Re(x_i)^2 + Im(x_i)^2 \right\}^{1/2}$$

s is computed instead of t since it is faster because it does not require the n square roots. Since $t \leq s \leq \sqrt{2}t$, s will often be an acceptable substitute for t.

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

Usage

SCILIB:

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

INTEGER*8    n, incx
REAL*8       s, SCASUM
COMPLEX*16 x(lenx)
s = SCASUM (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**|, i.e., x_i is stored in **x**((*i*–1)×|**incx**|+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., 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*8 FUNCTION SASUM (N, X,INCX)
      INTEGER*8 N, INCX
      REAL*8 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*8 N,INCX
REAL*8    S,SASUM,X(20)
N = 10
INCX = 1
S = SASUM (N,X,INCX)
```

SAXPY/CAXPY — 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

$$y \leftarrow ax + y$$

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

Usage

SCILIB:

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

INTEGER*8    n, incx, incy
COMPLEX*16 a, x(lenx), y(leny)
CALL CAXPY (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. Refer to "Purpose."

incx Increment for the array **x**:

incx ≥ 0	x is stored forward in array **x**, i.e., x_i is stored in **x**((i–1)×**incx**+1).
incx < 0	x is stored backward in array **x**, i.e., x_i is stored in **x**((i–**n**)×**incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., 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**, i.e., y_i is stored in **y**((i–1)×**incy**+1).
incy < 0	y is stored backward in array **y**, i.e., y_i is stored in **y**((i–**n**)×**incy**+1).

Use **incy** = 1 if the vector y is stored contiguously in array **y**, i.e., 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)
      INTEGER*8 N, INCX, INCY
      REAL*8 X(*),Y(*), A
      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*8 N,INCX,INCY
REAL*8    A,X(20),Y(20)
N = 10
A = 2.0
INCX = 1
INCY = 1
CALL SAXPY (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*8 N, INCX, INCY
REAL*8    A,B(20,21)
N = 10
A = -3.0
INCX = 20
INCY = 20
CALL SAXPY (N,A,B(4,1),INCX,B(5,1),INCY)
```

SCATTER — Scatter Sparse Vector

Purpose

Given a sparse vector x stored in compact form via a set of indices, this subprogram scatters 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

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

Usage

SCILIB:

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

Input

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

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

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

and

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

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

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

Output

y Array containing the elements of y, **y**$(i) = y_i$. If **m** ≤ 0, then **y** is unchanged. Otherwise, only the elements of **y** whose indices are included in **indx** are changed.

Notes

Cray Research, Inc. has declared this subprogram obsolete in release 6.0 of the UNICOS Math and Scientific Library.

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 SCATTER (M, Y, INDX, X)
      INTEGER*8 M,INDX(*)
      REAL*8 Y(*),X(*)
      DO 10 I = 1, M
        Y(INDX(I)) = X(I)
   10 CONTINUE
      RETURN
      END
```

Continued

SCATTER

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

SCOPY/CCOPY — Copy Vector

Purpose

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

$$y \leftarrow x$$

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

Usage

SCILIB:

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

INTEGER*8    n, incx, incy
COMPLEX*16 x(lenx), y(leny)
CALL CCOPY (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. Refer to "Purpose."

incx Increment for the array **x**:

incx ≥ 0	x is stored forward in array **x**, i.e., x_i is stored in **x**((i–1)× **incx**+1).
incx < 0	x is stored backward in array **x**, i.e., x_i is stored in **x**((i–**n**)× **incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., 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**, i.e., y_i is stored in **y**((i–1)× **incy**+1).
incy < 0	y is stored backward in array **y**, i.e., y_i is stored in **y**((i–**n**)× **incy**+1).

Use **incy** = 1 if the vector y is stored contiguously in array **y**, i.e., 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 x_i = **x**(1) for all i. This can be used to initialize all elements of y to a constant. Refer to "Example 2."

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

Example 2 Initialize a one-dimensional array to zero.

```
INTEGER*8 N, INCX, INCY
REAL*8    Y(20)
N = 10
INCX = 0
INCY = 1
CALL SCOPY (N,0.0,INCX,Y,INCY)
```

SDOT/CDOTC/CDOTU — 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 may be stored in one-dimensional arrays or in either rows or columns of two-dimensional arrays. Indexing through the arrays may be either forward or backward.

Usage

SCILIB:

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

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

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

Input

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

x Array of length **lenx** = (**n**–1)×|**incx**|+1 containing the n-vector x. x is used in unconjugated form by the subprograms. Refer to "Purpose."

incx Increment for the array **x**:

incx ≥ 0	x is stored forward in array **x**, i.e., x_i is stored in **x**((i–1)× **incx**+1).
incx < 0	x is stored backward in array **x**, i.e., x_i is stored in **x**((i–**n**)× **incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., 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**, i.e., y_i is stored in **y**((i–1)× **incy**+1).
incy < 0	y is stored backward in array **y**, i.e., y_i is stored in **y**((i–**n**)× **incy**+1).

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

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

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

unless the subprogram name is CDOTC, in which case

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

Notes If **incx** = 0, then $x_i = \mathbf{x}(1)$ for all i. If **incy** = 0, then $y_i = \mathbf{y}(1)$ for all i. In either of these cases, another SCILIB subprogram would be more efficient.

FORTRAN Equivalent

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

SDOT/CDOTC/CDOTU

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*8 N, INCX, INCY
REAL*8    S,SDOT,X(20,21),Y(20)
N = 10
S = SDOT (N,X(4,1),20,Y,1)
```

Euclidean Norm — SNRM2/SCNRM2

Purpose

Given a real, integer, or complex vector x of length n, these subprograms compute the Euclidean (i.e., l_2) norm of the vector

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

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

Usage

SCILIB:

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

INTEGER*8    n, incx
REAL*8       s, SCNRM2
COMPLEX*16 x(lenx)
s = SCNRM2 (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**|, i.e., x_i is stored in **x**((*i*–1)×|**incx**|+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., 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.

SNRM2/SCNRM2 Continued

FORTRAN Equivalent

```
REAL*8 FUNCTION SNRM2 (N, X,INCX)
REAL*8 T,X(*)
SNRM2 = 0.0
IF ( N .LE. 0 ) RETURN
T = 0.0
IX = 1
INCXA = ABS ( INCX )
disable overflow and underflow traps
DO 10 I = 1, N
   T = T + X(IX) ** 2
   IX = IX + INCXA
10 CONTINUE
   IF ( no overflow occurred ) THEN
     IF ( T .GT. N * 2.0 ** -104 .OR. no underflow occurred ) THEN
        SNRM2 = SQRT ( T )
        RETURN
     ELSE
        SCALE = 2.0 ** 72
     END IF
   ELSE
     SCALE = 0.5 ** 72
   END IF
   T = 0.0
   IX = 1
   DO 20 I = 1, N
      T = T + ( SCALE * X(IX) ) ** 2
      IX = IX + INCXA
20 CONTINUE
   reenable overflow trap if originally enabled
   SNRM2 = SQRT ( T ) / SCALE
   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*8 N,INCX
REAL*8    S,SNRM2,X(20)
N = 10
INCX = 1
S = SNRM2 (N,X,INCX)
```

Sparse Elementary Vector Operation — SPAXPY

Purpose

Given a real 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, this subprogram performs 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 $\{k_1, k_2, \ldots, k_m\}$ be the indices of these elements. All *uninteresting* elements of x are assumed to be zero. Let y be an ordinary n-vector. If x is represented by arrays **x** and **indx** such that $\mathbf{indx}(i) = k_i$ and $\mathbf{x}(i) = x_{k_i}$ then these subprograms compute

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

Usage

SCILIB:

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

Input

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

a The scalar a.

x Array of length **m** containing the interesting elements of x. $\mathbf{x}(j) = x_i$ if $\mathbf{indx}(j) = i$.

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

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

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

and

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

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

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.

SPAXPY Continued

FORTRAN Equivalent

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

Sparse Dot Product — SPDOT

Purpose

Given a real sparse vector x stored in compact form via an index vector, and a dense vector y stored in full storage form, this subprogram computes the sparse dot product

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

More precisely, let x be a sparse n-vector with $m \leq n$ *interesting* (usually nonzero) elements, let $\{k_1, k_2, \ldots, k_m\}$ be the indices of these elements. (While some interesting elements of x may 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 $\mathbf{indx}(i) = k_i$ and $\mathbf{x}(i) = x_{k_i}$, then these subprograms compute

$$s = \sum_{i=1}^{m} \mathbf{x}_i y_{k_i}$$

Usage

SCILIB:

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

Input

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

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

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**.

x Array of length **m** containing the interesting elements of x.

Output

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

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

FORTRAN Equivalent

```
      REAL*8 FUNCTION SPDOT (M, Y, INDX, X)
      INTEGER*8 M,INDX(*)
      REAL*8 Y(*),X(*)
      SPDOT = 0.0
      DO 10 I = 1, M
         SPDOT = SPDOT + X(I) * Y(INDX(I))
   10 CONTINUE
      RETURN
      END
```

SPDOT — Continued

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

Apply Givens Rotation — SROT/CROT

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

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

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

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

Usage

SCILIB:

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

INTEGER*8    n, incx, incy
REAL*8       c
COMPLEX*16 x(lenx), y(leny), s
CALL CROT (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**, i.e., x_i is stored in **x**((*i*–1)×**incx**+1).

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

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., 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**, i.e., y_i is stored in **y**((*i*–1)×**incy**+1).

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

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

c The scalar c.

SROT/CROT — Continued

	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 resulting 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.

SCILIB 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 on the CONVEX supercomputer.

FORTRAN Equivalent

```
      SUBROUTINE SROT (N, X,INCX, Y,INCY, C,S)
      REAL*8 C,S,TEMP,X(*),Y(*)
      INTEGER*8 N, INCX, INCY
      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*8 N,INCX,INCY
      REAL*8    X(20),Y(20),C,S
      N = 10
      INCX = 1
      INCY = 1
      CALL SROT (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" in the description of SROTM.

```
      INTEGER*8 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 SROTG (A(I,I),A(J,I),C,S)
            CALL SROT  (N,A(I,I+1),INCA,A(J,I+1),INCA,C,S)
   10    CONTINUE
   20 CONTINUE
```

Construct Givens Rotation — SROTG/CROTG

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

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

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, or CROT to apply the Givens rotation to a pair of vectors.

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

Usage

SCILIB:

```
REAL*8 a, b, c, s
CALL SROTG (a, b, c, s)

REAL*8         c
COMPLEX*16 a, b, s
CALL CROTG (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. In SROTG, 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. Since 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 may be reconstructed from z as follows:

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

c The rotation scalar c.

s The rotation scalar s.

Notes

SCILIB 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 on the CONVEX supercomputer.

SROTG/CROTG Continued

Example Construct a Givens plane rotation that will rotate 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 SROTG (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 SROT.

Apply Modified Givens Rotation — SROTM

Purpose

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

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

Refer to the description of the companion subprogram SROTMG for more details about the modified Givens rotation.

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

Usage

SCILIB:

```
INTEGER*8 n, incx, incy
REAL*8    x(lenx), y(leny), param(5)
CALL SROTM (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**, i.e., x_i is stored in **x**((i –1)× **incx**+1).
incx < 0 x is stored backward in array **x**, i.e., x_i is stored in **x**((i –**n**)× **incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., 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**, i.e., y_i is stored in **y**((i –1)× **incy**+1).
incy < 0 y is stored backward in array **y**, i.e., y_i is stored in **y**((i –**n**)× **incy**+1).

Use **incy** = 1 if the vector y is stored contiguously in array **y**, i.e., 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** will usually have been set by the companion subprogram SROTMG; refer to the description of this companion subprogram for the specific contents of **param**.

SROTM — Continued

Output

x and **y** If **n** ≤ 0 or if **param**(1) = -2, **x** and **y** are unchanged. Otherwise, the resulting 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.

SCILIB 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 on the CONVEX supercomputer.

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*8 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" in the description of SROT.)

```
      INTEGER*8 INCA,I,J,N
      REAL*8    A(20,21),D(20),PARAM(5)
      INCA = 20
      DO 10 I = 1, 10
        D(I) = 1.0
10    CONTINUE
      DO 30 I = 1, 9
        N = 10 - I
        DO 20 J = I+1, 10
          CALL SROTMG (D(I),D(J),A(I,I),A(J,I),PARAM)
          CALL SROTM  (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 SSCAL (N,SQRT(D(I)),A(I,I),INCA)
40    CONTINUE
```

Construct Modified Givens Rotation — SROTMG

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 $\sim 4n$ floating-point multiplications, $\sim 2n$ 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 will usually have two elements equal to ± 1. When this is true, computing H and applying it to a pair of n-vectors requires $\sim 2n$ floating-point multiplications, $\sim 2n$ floating-point additions, and no square roots. Companion SCILIB subprograms SROTM 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 as the modified Givens rotations are constructed, and 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.

Usage

SCILIB:

```
REAL*8  d1, d2, x1, y1, param(5)
CALL SROTMG (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 will be annihilated by the rotation.

Output

d1 The updated scale factor for the "x" row.

SROTMG — Continued

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** will usually be an argument to the companion subprogram SROTM.

param(1) specifies the form of the rotation matrix H, as follows:

$$\textbf{param}(1) = -2 \quad H = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\textbf{param}(1) = -1 \quad H = \begin{bmatrix} \textbf{param}(2) & \textbf{param}(4) \\ \textbf{param}(3) & \textbf{param}(5) \end{bmatrix}$$

$$\textbf{param}(1) = 0 \quad H = \begin{bmatrix} 1 & \textbf{param}(4) \\ \textbf{param}(3) & 1 \end{bmatrix}$$

$$\textbf{param}(1) = 1 \quad H = \begin{bmatrix} \textbf{param}(2) & 1 \\ -1 & \textbf{param}(5) \end{bmatrix}$$

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 SCILIB 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 on the CONVEX supercomputer.

Example Construct a modified Givens plane rotation that will rotate 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 SROTMG (D1,D2,X(1),Y(1),PARAM)
```

X(1) is the rotated result, so it should not be rotated by a subsequent call to SROTM.

Scale Vector — SSCAL/CSCAL/CSSCAL

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

$$x \leftarrow ax \text{ and } x \leftarrow a\bar{x}$$

where $\bar{x}$ is the complex conjugate of x. The vector may be stored in a one-dimensional array or in either a row or a column of a two-dimensional array.

Usage

SCILIB:

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

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

INTEGER*8    n, incx
REAL*8       a
COMPLEX*16 x(lenx)
CALL CSSCAL (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 unconjugated form by the subprograms. Refer to "Purpose."

incx Increment for the array **x**, **incx** ≠ 0. x is stored forward in array **x** with increment |**incx**|, i.e., x_i is stored in **x**((*i*–1)×|**incx**|+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., 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, ax replaces the input.

Notes

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

FORTRAN Equivalent

```
      SUBROUTINE SSCAL (N,A, X,INCX)
      REAL*8 A,X(*)
      INTEGER*8 N, INCX
      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
```

SSCAL/CSCAL/CSSCAL — Continued

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*8 N,INCX
REAL*8    A,X(20)
N = 10
INCX = 1
A = 2.0
CALL SSCAL (N,A,X,INCX)
```

Vector Sum — SSUM/CSUM

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

Usage

SCILIB:

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

INTEGER*8    n, incx
COMPLEX*16 s, CSUM, x(lenx)
s = CSUM (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**|, i.e., x_i is stored in **x**((*i*–1)×|**incx**|+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., 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*8 FUNCTION SSUM (N, X,INCX)
      INTEGER*8 N, INCX
      REAL*8 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*8 N,INCX
REAL*8    S,SSUM,X(20)
N = 10
INCX = 1
S = SSUM (N,X,INCX)
```

SSWAP/CSWAP — 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 may be stored in one-dimensional arrays or in either rows or columns of two-dimensional arrays, and the indexing through the arrays may be either forward or backward.

Usage

SCILIB:

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

INTEGER*8    n, incx, incy
COMPLEX*16 x(lenx), y(leny)
CALL CSWAP (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**, i.e., x_i is stored in **x**((i–1)×**incx**+1).

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

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., 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**, i.e., y_i is stored in **y**((i–1)×**incy**+1).

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

Use **incy** = 1 if the vector y is stored contiguously in array **y**, i.e., 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*8 TEMP,X(*),Y(*)
      INTEGER*8 N, INCX, INCY
      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*8 N,INCX,INCY
REAL*8    X(20),Y(20)
N = 10
INCX = 1
INCY = 1
CALL SSWAP (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*8 N,INCA
REAL*8    A(20,21)
N = 10
INCA = 20
CALL SSWAP (N,A(3,1),INCA,A(6,1),INCA)
```

WHENEQ/WHENNE/.../WHENILT — Find Selected Vector Elements

Purpose

Given a real or integer 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 with 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. These characters and the corresponding list contents may 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	$\{i : x_i \neq a\}$

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

Usage

SCILIB:

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

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

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

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

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

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

Input

n Number of elements of vector x to be compared to **a**. If **n** $\leq$ 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**:

incx $\geq$ 0	x is stored forward in array **x**, i.e., x_i is stored in **x**((i–1)× **incx**+1).
incx < 0	x is stored backward in array **x**, i.e., x_i is stored in **x**((i–**n**)× **incx**+1).

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

Continued WHENEQ/WHENNE/WHENFGE/WHENFGT/.../WHENILT

a The scalar a.

Output

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.

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

Notes

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

FORTRAN Equivalent

```
      SUBROUTINE WHENEQ (N,X,INCX,A,INDX,NINDX)
      INTEGER*8 N,X(*),INCX,A,INDX(*),NINDX
      IX = 1
      IF ( INCX .LT. 0 ) IX = 1 - (N-1) * INCX
      NINDX = 0
      DO 10 I = 1, N
         IF ( X(IX) .EQ. A ) THEN
            NINDX = NINDX + 1
            INDX(NINDX) = I
         END IF
         IX = IX + INCX
   10 CONTINUE
      RETURN
      END
```

Example 1 Find the zero 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*8 N,INCX,INDX(20),NINDX
      REAL*8    A,X(20)
      N = 10
      INCX = 1
      A = 0.0
      CALL WHENEQ (N,X,INCX,A,INDX,NINDX)
```

WHENEQ/WHENNE/WHENFGE/WHENFGT/.../WHENILT Continued

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

```
      INTEGER*8 I,N
      REAL*8    A,B,D,DLIM,R
      REAL*8    F(20000),X(20000),Y(20000),Z(20000)
      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 vectorizes, so even though it computes a few more exponentials than the original code, it is still considerably faster. WHENFLE 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*8 I,J,N,INDX(20000),NINDX
      REAL*8    A,B,DLIM,R
      REAL*8    D(20000),F(20000),X(20000),Y(20000),Z(20000)
      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 WHENFLE (N,D,1,DLIM,INDX,NINDX)
      DO 20 J = 1, NINDX
        I = INDX(J)
        CALL FORCE (D(I),F(I))
   20 CONTINUE
```

Find Selected Vector Elements WHENMEQ/WHENMGE/.../WHENMNE

Purpose

Given a vector x of length n, these subprograms search sequentially through the vector and fill an array with a list of the indices of the elements x_i which contain a specified group of bits that satisfy a specified relationship with 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. These characters and the corresponding list contents may be

xx	List contents
EQ	$\{i : \mathbf{AND(SHIFTR}(x_i,rshift),mask) = a\}$
GE	$\{i : \mathbf{AND(SHIFTR}(x_i,rshift),mask) \geq a\}$
GT	$\{i : \mathbf{AND(SHIFTR}(x_i,rshift),mask) > a\}$
LE	$\{i : \mathbf{AND(SHIFTR}(x_i,rshift),mask) \leq a\}$
LT	$\{i : \mathbf{AND(SHIFTR}(x_i,rshift),mask) < a\}$
NE	$\{i : \mathbf{AND(SHIFTR}(x_i,rshift),mask) \neq a\}$

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

Usage

SCILIB:

```
INTEGER*8 n, x(lenx), incx, a, indx(n), nindx, mask, rshift
CALL WHENMxx (n, x, incx, a, indx, nindx, mask, rshift)
```

Input

n Number of elements of vector x to be compared to **a**. If **n** $\leq$ 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**:

incx $\geq$ 0 x is stored forward in array **x**, i.e., x_i is stored in **x**((i-1)× **incx**+1).

incx $<$ 0 x is stored backward in array **x**, i.e., x_i is stored in **x**((i-**n**)× **incx**+1).

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

a The scalar a.

mask Mask of 1-bits to extract desired group of bits from the shifted elements of x with a bitwise logical product operation. Refer to "Purpose."

rshift Number of bits by which to right shift each element of x so as to align the specified group of bits with **a**, 0 $\leq$ **rshift** $\leq$ 63. Refer to "Purpose."

Output

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.

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

WHENMEQ/WHENMGE/WHENMGT/.../WHENMNE Continued

FORTRAN Equivalent

```
      SUBROUTINE WHENMEQ (N,X,INCX,A,INDX,NINDX,MASK,RSHIFT)
      INTEGER*8 N,X(*),INCX,A,INDX(*),NINDX,MASK,RSHIFT
      IX = 1
      IF ( INCX .LT. 0 ) IX = 1 - (N-1) * INCX
      NINDX = 0
      DO 10 I = 1, N
         IF ( AND(SHIFTR(X(IX),RSHIFT),MASK) .EQ. A ) THEN
            NINDX = NINDX + 1
            INDX(NINDX) = I
         END IF
         IX = IX + INCX
   10 CONTINUE
      RETURN
      END
```

Example Find the odd elements of an INTEGER*8 vector x, where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20.

```
INTEGER*8 N,X(20),INCX,A,INDX(20),NINDX,MASK,RSHIFT
N = 10
INCX = 1
A = 1
MASK = 1
RSHIFT = 0
CALL WHENMEQ (N,X,INCX,A,INDX,NINDX,MASK,RSHIFT)
```

3

Basic Matrix Operations

Overview

This chapter describes the subprograms in the Level 2 (two-loop) BLAS and the Level 3 (three-loop) BLAS. Collectively, these two sets of subprograms are called the Extended BLAS. The most important of these subprograms have been coded in highly tuned CONVEX assembly language.

This chapter explains how to use the SCILIB matrix subprograms, which perform common computationally-intensive linear algebra operations. The operations covered are:

- basic matrix/vector operations
- basic matrix/matrix operations

Chapter 4 discusses matrix inverse operations.

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

What You Need to Know to Use These Subprograms

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 first letter 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
C	Single 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). CTRSM solves a system of linear equations with one triangular coefficient matrix and a matrix of right-hand sides, using the single 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		GE	MM	MV	R					
	C	GE	MM	MV					RC	RU
S	C	GB		MV						
	C	HE	MM	MV	R	R2	RK	R2K		
	C	HB		MV						
	C	HP		MV	R	R2				
S		SY	MM	MV	R	R2	RK	R2K		
	C	SY	MM				RK	R2K		
S		SB		MV						
S		SP		MV	R	R2				
S	C	TR	MM	MV					SM	SV
S	C	TB		MV						SV
S	C	TP		MV						SV

Supplemental Reading

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.

Subprogram Descriptions

Specialized Matrix-Matrix Multiply — MXM

Purpose

This subprogram computes the matrix-matrix product $C = AB$, where A is an m-by-k matrix, B is a k-by-n matrix, and C is an m-by-n matrix. The elements of the matrices must be stored in consecutive memory locations in two-dimensional arrays of size m by k, k by n, and m by n, respectively. SCILIB subprograms SGEMM, SGEMMS, and MXMA allow more general matrix storage and also admit the transposes of A, B, and, in the case of MXMA, C.

Usage

SCILIB:

```
INTEGER*8 m, k, n
REAL*8    a(m, k), b(k, n), c(m, n)
CALL MXM (a, m, b, k, c, n)
```

Input

a Array containing the m-by-k matrix A.

m Number of rows in matrices A and C, **m** ≥ 0. If **m** $= 0$, the subprogram does not reference **a**, **b**, or **c**.

b Array containing the k-by-n matrix B.

k Number of columns in matrix A and number of rows in matrix B, **k** ≥ 0. If **k** $= 0$, the subprogram computes $C \leftarrow 0$ without referencing **a** or **b**.

n Number of columns in matrices B and C, **n** ≥ 0. If **n** $= 0$, the subprogram does not reference **a**, **b**, or **c**.

Output

c The result C matrix.

Notes

Cray Research, Inc. has declared this subprogram obsolete in release 6.0 of the UNICOS Math and Scientific Library.

If an error in the arguments is detected, the subprograms call error handler XERSCI, which writes an error message onto the standard error file and terminates execution. The standard version of XERSCI (refer to Chapter 9, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are

m < 0,
n < 0, and
k < 0.

MXM Continued

FORTRAN Equivalent Except for the argument error checking, the following FORTRAN subroutine is equivalent to MXMA, and illustrates the meanings of the six increment arguments.

```
      SUBROUTINE MXM (A,M,B,K,C,N)
      INTEGER*8 M,K,N
      REAL*8 A(M,K),B(K,N),C(M,N)
      DO 110 J = 1, N
        DO 120 I = 1, M
          C(I,J) = 0.0
110     CONTINUE
120   CONTINUE
      DO 150 J = 1, N
        DO 140 L = 1, K
          DO 130 I = 1, M
            C(I,J) = C(I,J) + A(I,L) * B(L,J)
130       CONTINUE
140     CONTINUE
150   CONTINUE
      RETURN
      END
```

Example Form the REAL*8 matrix product $C = AB$, where A is a 9-by-6 real matrix stored in an array A whose dimensions are 9 by 6, B is a 6 by 8 real matrix stored in an array B of dimension 6 by 8, and C is a 9 by 8 real matrix stored in an array C, also of dimension 9 by 8.

```
INTEGER*8 M,K,N
REAL*8    A(9,6),B(6,8),C(9,8)
M = 9
N = 8
K = 6
CALL MXM (A,M,B,K,C,N)
```

Generalized Matrix-Matrix Multiply — MXMA

Purpose

This subprogram computes the matrix-matrix product $C = AB$, where A is an m-by-k matrix, B is a k-by-n matrix, and C is an m-by-n matrix. The rows and columns of the matrices may be stored with unit or non-unit strides, effectively allowing A, B, and C, or their transposes, to be stored in two-dimensional arrays via the storage-association rules of FORTRAN.

SCILIB subprograms SGEMM and SGEMMS allow matrix and transposed matrix storage without resorting to storage association, also admitting the ability to add or subtract the product matrix from the original contents of the result matrix.

Usage

SCILIB:

```
INTEGER*8 ia, ja, ib, jb, ic, jc, m, k, n
REAL*8    a(lena), b(lenb), c(lenc)
CALL MXMA (a, ia, ja, b, ib, jb, c, ic, jc, m, k, n)
```

Input

a Array containing the m-by-k matrix A. Typically, **a** will be a two-dimensional array with the rows and columns of A comprising one-dimensional array sections of **a**. Refer to "Notes" for suggested usages. Treating **a** as a one-dimensional array results in

$$\mathbf{lena} = (\mathbf{m}-1)\times|\mathbf{ia}| + (\mathbf{k}-1)\times|\mathbf{ja}| + 1.$$

A_{ij}, $1 \le \mathbf{i} \le \mathbf{m}$, $1 \le \mathbf{j} \le \mathbf{k}$, is stored in

$$\mathbf{a}((i-1)\times\mathbf{ia} + (j-1)\times\mathbf{ja} + 1).$$

Note that negative **ia** or **ja** will result in subscript values that lie outside the **a** array as declared above. This need not be an error; see "Example 3" for details.

ia Storage increment between successive elements in the same column of matrix A in array **a**. Refer to "Notes" for suggested values.

ja Storage increment between successive elements in the same row of matrix A in array **a**. Refer to "Notes" for suggested values.

b Array containing the k-by-n matrix B. Typically, **b** will be a two-dimensional array with the rows and columns of B comprising one-dimensional array sections of **b**. Refer to "Notes" for suggested usages. Treating **b** as a one-dimensional array results in

$$\mathbf{lenb} = (\mathbf{k}-1)\times|\mathbf{ib}| + (\mathbf{n}-1)\times|\mathbf{jb}| + 1.$$

B_{ij}, $1 \le \mathbf{i} \le \mathbf{k}$, $1 \le \mathbf{j} \le \mathbf{n}$, is stored in

$$\mathbf{b}((i-1)\times\mathbf{ib} + (j-1)\times\mathbf{jb} + 1).$$

Note that negative **ib** or **jb** will result in subscript values that lie outside the **b** array as declared above. This need not be an error; see "Example 3" for details.

ib Storage increment between successive elements in the same column of matrix B in array **b**. Refer to "Notes" for suggested values.

jb Storage increment between successive elements in the same row of matrix B in array **b**. Refer to "Notes" for suggested values.

ic Storage increment between successive elements in the same column of matrix C in array **c**. Refer to "Notes" for suggested values.

jc Storage increment between successive elements in the same row of matrix C in array **c**. Refer to "Notes" for suggested values.

m Number of rows in matrices A and C, $\mathbf{m} \geq 0$. If $\mathbf{m} = 0$, the subprogram does not reference **a**, **b**, or **c**.

k Number of columns in matrix A and number of rows in matrix B, $\mathbf{k} \geq 0$. If $\mathbf{k} = 0$, the subprogram computes $C \leftarrow 0$ without referencing **a** or **b**.

n Number of columns in matrices B and C, $\mathbf{n} \geq 0$. If $\mathbf{n} = 0$, the subprogram does not reference **a**, **b**, or **c**.

Output

c The result C matrix. Typically, **c** will be a two-dimensional array with the rows and columns of C comprising one-dimensional array sections of **c**. Refer to "Notes" for suggested usages. Treating **c** as a one-dimensional array results in

$$\mathbf{lenc} = (\mathbf{m}-1)\times|\mathbf{ic}| + (\mathbf{n}-1)\times|\mathbf{jc}| + 1.$$

C_{ij}, $1 \leq \mathbf{i} \leq \mathbf{m}$, $1 \leq \mathbf{j} \leq \mathbf{n}$, is stored in

$$\mathbf{c}((i-1)\times\mathbf{ic} + (j-1)\times\mathbf{jc} + 1).$$

Note that negative **ic** or **jc** will result in subscript values that lie outside the **c** array as declared above. This need not be an error; see "Example 3" for details.

Notes

Cray Research, Inc. has declared this subprogram obsolete in release 6.0 of the UNICOS Math and Scientific Library.

Typically, **a**, **b**, and **c** will be two-dimensional arrays with the rows and columns comprising one-dimensional sections of the arrays, i.e., one subscript will vary within a row or column of the matrix, and the other will be constant.

If **a**, for example, is a two-dimensional array of dimension **lda** by **mda**, and A is stored in untransposed form in **a**, then $\mathbf{ia} = 1$ and $\mathbf{ja} = \mathbf{lda}$, while if A is stored in transposed form (A^T is stored), then $\mathbf{ia} = \mathbf{lda}$ and $\mathbf{ja} = 1$.

If an error in the arguments is detected, the subprograms call error handler XERSCI, which writes an error message onto the standard error file and terminates execution. The standard version of XERSCI (refer to Chapter 9, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are

$\mathbf{m} < 0$,
$\mathbf{n} < 0$,
$\mathbf{k} < 0$,
$\mathbf{ia} = 0$,
$\mathbf{ja} = 0$,
$\mathbf{ib} = 0$,
$\mathbf{jb} = 0$,
$\mathbf{ic} = 0$, and
$\mathbf{jc} = 0$.

FORTRAN Equivalent Except for the argument error checking, the following FORTRAN subroutine is equivalent to MXMA, and illustrates the meanings of the six increment arguments.

```
      SUBROUTINE MXMA (A,IA,JA, B,IB,JB, C,IC,JC, M,K,N)
      INTEGER*8 IA,JA,IB,JB,IC,JC,M,K,N
      REAL*8 A(*),B(*),C(*)
      DO 120 J = 1, N
         DO 110 I = 1, M
            C((I-1)*IC+(J-1)*JC+1) = 0.0      ! C(I,J) = 0.0
  110    CONTINUE
  120 CONTINUE
      DO 150 J = 1, N
         DO 140 L = 1, K
            DO 130 I = 1, M
               C((I-1)*IC+(J-1)*JC+1) =       ! C(I,J) =
     1         C((I-1)*IC+(J-1)*JC+1) +       ! C(I,J) +
     2         A((I-1)*IA+(L-1)*JA+1) *       ! A(I,L) *
     3         B((L-1)*IB+(J-1)*JB+1)         ! B(L,J)
  130       CONTINUE
  140    CONTINUE
  150 CONTINUE
      RETURN
      END
```

Example 1 Form the REAL*8 matrix product $C = AB$, where A is a 9-by-6 real matrix stored in an array A of dimension 10 by 11, B is a 6-by-8 real matrix stored in an array B of dimension 12 by 13, and C is a 9-by-8 real matrix stored in an array C, of dimension 14 by 15.

```
INTEGER*8 IA,JA,IB,JB,IC,JC,M,K,N
REAL*8    A(10,11),B(12,13),C(14,15)
IA = 1
JA = 10
IB = 1
JB = 12
IC = 1
JC = 14
M = 9
N = 8
K = 6
CALL MXMA (A,IA,JA, B,IB,JB, C,IC,JC, M,K,N)
```

Example 2 Form the REAL*8 matrix product $C = A^T B$, where A is a 6 by 9 real matrix stored in an array A of dimension 10 by 11, B is a 6 by 8 real matrix stored in an array B of dimension 12 by 13, and C is a 9-by-8 real matrix stored in an array C, of dimension 14 by 15.

```
INTEGER*8 IA,JA,IB,JB,IC,JC,M,K,N
REAL*8    A(10,11),B(12,13),C(14,15)
IA = 10
JA = 1
IB = 1
JB = 12
IC = 1
JC = 14
M = 9
N = 8
K = 6
CALL MXMA (A,IA,JA, B,IB,JB, C,IC,JC, M,K,N)
```

Example 3 Form the REAL*8 matrix product $C = AB$, where A is a 9-by-6 real matrix stored "upside-down and backwards", i.e., with the row and column subscripts decreasing to the right and bottom, in an array A of dimension 10 by 11, B is a 6-by-8 real matrix stored in an array B of dimension 12 by 13, and C is a 9-by-8 real matrix stored in an array C, of dimension 14 by 15.

```
INTEGER*8 IA,JA,IB,JB,IC,JC,M,K,N
REAL*8    A(10,11),B(12,13),C(14,15)
IA = -1
JA = -10
IB = 1
JB = 12
IC = 1
JC = 14
M = 9
N = 8
K = 6
CALL MXMA (A(M,K),IA,JA, B,IB,JB, C,IC,JC, M,K,N)
```

Specialized Matrix-Vector Multiply — MXV

Purpose

This subprogram computes the matrix-vector product $y = Ax$, where A is an m-by-n matrix, x is an n-vector, and y is an m-vector. The elements of A must be stored in consecutive memory locations in a two-dimensional array of size m by n, and the elements of the x and y must be stored in consecutive memory locations in one-dimensional arrays of size n and m, respectively. SCILIB subprograms SGEMV and MXVA allow more general storage and also admit the transpose of A. SGEMV also admits the ability to add or subtract the product from the original contents of the result vector.

Usage

SCILIB:

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

Input

a Array containing the m-by-n matrix A.

m Number of rows in matrix A and length of vector y, **m** > 0. If **m** $= 0$, the subprogram does not reference **a**, **x**, or **y**.

x Array containing the n-vector x.

n Number of columns in matrix A and length of vector x, **n** > 0. If **n** $= 0$, the subprogram does not reference **a**, **x**, or **y**.

Output

y The result y vector.

Notes

Cray Research, Inc. has declared this subprogram obsolete in release 6.0 of the UNICOS Math and Scientific Library.

If an error in the arguments is detected, the subprograms call error handler XERSCI, which writes an error message onto the standard error file and terminates execution. The standard version of XERSCI (refer to Chapter 9, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are

m ≤ 0, and
n ≤ 0.

FORTRAN Equivalent

Except for the argument error checking, the following FORTRAN subroutine is equivalent to MXVA, and illustrates the meanings of the four increment arguments.

```
      SUBROUTINE MXV (A,M,X,N,Y)
      INTEGER*8 M,N
      REAL*8 A(M,N),X(N),Y(M)
      DO 110 I = 1, M
        Y(I) = 0.0
  110 CONTINUE
      DO 130 J = 1, N
        DO 120 I = 1, M
          Y(I) = Y(I) + A(I,J) * X(J)
  120   CONTINUE
  130 CONTINUE
      RETURN
      END
```

MXV Continued

Example Form the REAL*8 matrix-vector product $y = Ax$, where A is a 9 by 6 real matrix stored in an array A whose dimensions are 9 by 6, x is a real vector 6 elements long stored in an array X of dimension 6, and y is a real vector 9 elements long stored in an array Y of dimension 9.

```
INTEGER*8 M,K,N
REAL*8    A(9,6),X(6),Y(9)
M = 9
N = 6
CALL MXV (A,M,X,N,Y)
```

Generalized Matrix-Vector Multiply — MXVA

Purpose

This subprogram computes the matrix-vector product $y = Ax$, where A is an m-by-n matrix, x is a n-vector, and y is an m-vector. The rows and columns of A may be stored with unit or non-unit stride, effectively allowing A or its transpose to be stored in a two-dimensional array via the storage-association rules of FORTRAN.

SCILIB subprogram SGEMV allows matrix and transposed matrix storage without resorting to storage association, also admitting the ability to add or subtract the product from the original contents of the result vector.

Usage

SCILIB:

```
INTEGER*8 ia, ja, incx, incy, m, n
REAL*8    a(lena), x(lenx), y(leny)
CALL MXVA (a, ia, ja, x, incx, y, incy, m, n)
```

Input

a Array containing the m-by-n matrix A. Typically, **a** will be a two-dimensional array with the rows and columns of A comprising one-dimensional array sections of **a**. Refer to "Notes" for suggested usages. Treating **a** as a one-dimensional array results in

$$\mathbf{lena} = (\mathbf{m}-1)\times|\mathbf{ia}| + (\mathbf{n}-1)\times|\mathbf{ja}| + 1.$$

A_{ij}, $1 \le i \le m$, $1 \le j \le n$, is stored in

$$\mathbf{a}((i-1)\times\mathbf{ia} + (j-1)\times\mathbf{ja} + 1).$$

Note that negative **ia** or **ja** will result in subscript values that lie outside the **a** array as declared above. This need not be an error; refer to "Example 3" for details.

ia Storage increment between successive elements in the same column of matrix A in array **a**. Refer to "Notes" for suggested values.

ja Storage increment between successive elements in the same row of matrix A in array **a**. Refer to "Notes" for suggested values.

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

incx Storage increment between successive elements of vector x in array **x**. 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**, i.e., if x_i is stored in $\mathbf{x}(i)$.

Note that negative **incx** will result in subscript values that lie outside the **x** array as declared above. This need not be an error; refer to "Example 3" for details.

incy Storage increment between successive elements of vector y in array **y**. y_i is stored in $\mathbf{y}((i-1)\times\mathbf{incy} + 1)$. Use **incy** $= 1$ if the vector y is stored contiguously in array **y**, i.e., if y_i is stored in $\mathbf{y}(i)$.

Note that a negative **incy** will result in subscript values that lie outside the **y** array as declared above. This need not be an error; refer to "Example 3" for details.

m Number of rows in matrix A and vector y, $\mathbf{m} > 0$.

	n	Number of columns in matrix A and length of vector x, **n** > 0. **a** or **x**.
Output	**y**	Array of length **leny** = (**m**–1)× |**incy**| + 1 containing the resulting y vector.

Notes

Cray Research, Inc. has declared this subprogram obsolete in release 6.0 of the UNICOS Math and Scientific Library.

Typically, **a** will be a two-dimensional array with rows and columns comprising one-dimensional sections of the array, i.e., one subscript will vary within a row or column of the matrix, and the other will be constant.

If **a** is a two-dimensional array of dimension **lda** by **mda**, and A is stored in untransposed form in **a**, then **ia** = 1 and **ja** = **lda**, while if A is stored in transposed form (A^T is stored), then **ia** = **lda** and **ja** = 1.

If an error in the arguments is detected, the subprograms call error handler XERSCI, which writes an error message onto the standard error file and terminates execution. The standard version of XERSCI (refer to Chapter 9, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are

m ≤ 0,
n ≤ 0,
ia = 0,
ja = 0,
incx = 0, and
incy = 0.

FORTRAN Equivalent

Except for the argument error checking, the following FORTRAN subroutine is equivalent to MXVA, and illustrates the meanings of the four increment arguments.

```
      SUBROUTINE MXVA (A,IA,JA, X,INCX, Y,INCY, M,N)
      INTEGER*8 IA,JA,INCX,INCY,M,N
      REAL*8 A(*),X(*),Y(*)
      DO 110 I = 1, M
        Y((I-1)*INCY+1) = 0.0          ! Y(I) = 0.0
  110 CONTINUE
      DO 130 J = 1, N
        DO 120 I = 1, M
          Y((I-1)*INCY+1) =            ! Y(I) =
     1    Y((I-1)*INCY+1) +            ! Y(I) +
     2    A((I-1)*IA+(J-1)*JA+1) *     ! A(I,J) *
     3    X((J-1)*INCX+1)              ! X(J)
  120   CONTINUE
  130 CONTINUE
      RETURN
      END
```

Example 1 Form the REAL*8 matrix-vector product $y = Ax$, where A is a 9 by 6 real matrix stored in an array A of dimension 10 by 11, x is a real vector 6 elements long stored in an array X of dimension 12, and y is a real vector 9 elements long stored in an array Y, of dimension 13.

```
INTEGER*8 IA,JA,INCX,INCY,M,N
REAL*8    A(10,11),B(12),Y(13)
IA = 1
JA = 10
INCX = 1
INCY = 1
M = 9
N = 6
CALL MXVA (A,IA,JA, X,INCX, Y,INCY, M,N)
```

Example 2 Form the REAL*8 matrix-vector product $y = A^T x$, where A is a 6-by-9 real matrix stored in an array A of dimension 10 by 11, x is a real vector 6 elements long stored in an array X of dimension 12, and y is a real vector 9 elements long stored in an array Y, of dimension 13.

```
INTEGER*8 IA,JA,INCX,INCY,M,N
REAL*8    A(10,11),X(12),Y(13)
IA = 10
JA = 1
INCX = 1
INCY = 1
M = 9
N = 6
CALL MXVA (A,IA,JA, X,INCX, Y,INCY, M,N)
```

Example 3 Form the REAL*8 matrix-vector product $y = Ax$, where A is a 9 by 6 real matrix stored "upside-down and backwards", i.e., with the row and column subscripts decreasing to the right and bottom, in an array A of dimension 10 by 11, x is a real vector 6 elements long stored in an array X of dimension 12, and y is a real vector 9 elements long stored in an array Y, of dimension 13.

```
INTEGER*8 IA,JA,INCX,INCY,M,N
REAL*8    A(10,11),X(12),Y(13)
IA = -1
JA = -10
INCX = 1
INCY = 1
M = 9
N = 6
CALL MXVA (A(M,K),IA,JA, X,INCX, Y,INCY, M,N)
```

SGBMV/CGBMV — 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 may 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

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

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 **SCILIB:**

```
CHARACTER*1 trans
INTEGER*8   m, n, kl, ku, ldab, incx, incy
REAL*8      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
COMPLEX*16  alpha, beta, ab(ldab, n), x(lenx), y(leny)
CALL CGBMV (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, i.e., the number of nonzero diagonals below the principal diagonal in the band, $0 \leq \mathbf{kl} < \mathbf{n}$.

ku The upper bandwidth of A, i.e., the number of nonzero diagonals above the principal diagonal in the band, $0 \leq \mathbf{ku} < \mathbf{n}$.

alpha The scalar α. If $\mathbf{alpha} = 0$, 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 $\mathbf{ab}(\mathbf{ku}+1+i-j, j)$. 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 $\mathbf{ldab} \geq \mathbf{kl}+\mathbf{ku}+1$.

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	$\mathbf{lenx} = (\mathbf{n}-1)\times\lvert\mathbf{incx}\rvert+1$
otherwise	x has m elements	$\mathbf{lenx} = (\mathbf{m}-1)\times\lvert\mathbf{incx}\rvert+1$

incx Increment for the array **x**, $\mathbf{incx} \neq 0$:

$\mathbf{incx} > 0$	x is stored forward in array **x**, i.e., x_i is stored in $\mathbf{x}((i-1)\times\mathbf{incx}+1)$.
$\mathbf{incx} < 0$	x is stored backward in array **x**, i.e., if **trans** = 'N' or 'n', then x_i is stored in $\mathbf{x}((i-\mathbf{n})\times\mathbf{incx}+1)$;

otherwise, x_i is stored in **x**((*i*-**m**)× **incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., 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**, i.e., y_i is stored in **y**((*i*-1)× **incy**+1).
incy < 0	y is stored backward in array **y**, i.e., 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**, i.e., 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) may 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, and
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 **trans** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Form the REAL*8 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*8   M,N,KL,KU,LDAB,INCX,INCY
REAL*8      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*8 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 SGBMV ('TRANSPOSE',M,N,KL,KU,-RHO,AB,LDAB,X,1,0.5,Y,1)
```

SGEMM/CGEMM — 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 may be replaced by A^T or A^*, where A is a k-by-m matrix, and B may 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 may be stored in the result matrix (which is always of size m by n) 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 forms

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

Usage

SCILIB:

```
CHARACTER*1 transa, transb
INTEGER*8   m, n, k, lda, ldb, ldc
REAL*8      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
COMPLEX*16  alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL CGEMM (transa, transb, m, n, k, alpha, a, lda, b, ldb,
            beta, c, ldc)
```

Input

transa Transposition option for A:

'N' or 'n'	Use m-by-k matrix A
'T' or 't'	Use A^T where A is a k-by-m matrix
'C' or 'c'	Use A^* where A is a k-by-m matrix

where A^T is the transpose of A and A^* is the conjugate transpose. In the real subprograms, 'C' and 'c' have the same meaning as 'T' and 't'.

transb Transposition option for B:

'N' or 'n'	Use k-by-n matrix B
'T' or 't'	Use B^T where B is an n-by-k matrix
'C' or 'c'	Use B^* where B is an n-by-k matrix

where B^T is the transpose of B and B^* is the conjugate transpose. In the real subprograms, 'C' and 'c' have the same meaning as 'T' and 't'.

m Number of rows in matrix C, $\mathbf{m} \geq 0$. If $\mathbf{m} = 0$, the subprograms do not reference **a**, **b**, or **c**.

n Number of columns in matrix C, $\mathbf{n} \geq 0$. If $\mathbf{n} = 0$, the subprograms do not reference **a**, **b**, or **c**.

k The *middle* dimension of the matrix multiply, $\mathbf{k} \geq 0$. If $\mathbf{k} = 0$, the subprograms compute $C \leftarrow \beta C$ without referencing **a** or **b**.

alpha The scalar α. If **alpha** = 0, the subprograms compute $C \leftarrow \beta C$ without referencing **a** or **b**.

a Array containing the matrix A, whose size is indicated by **transa**:

'N' or 'n'	A is an m-by-k matrix
otherwise	A is a k-by-m matrix

lda The leading dimension of array **a** as declared in the calling program unit, with **lda** ≥ max(the number of rows of A,1).

b Array containing the matrix B, whose size is indicated by **transb**:

'N' or 'n'	B is a k-by-n matrix
otherwise	B is an n-by-k matrix

ldb The leading dimension of array **b** as declared in the calling program unit, with **ldb** ≥ max(the number of rows of B,1).

beta The scalar β.

c Array containing the m-by-n matrix C. Not used as input if **beta** = 0.

ldc The leading dimension of array **c** as declared in the calling program unit, with **ldc** ≥ max(**m**,1).

Output **c** The updated C matrix replaces the input.

Notes These subprograms conform to specifications of the Level 3 BLAS.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) may 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, and
ldc < max(**m**,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, 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."

SGEMM/CGEMM Continued

Example 1 Form the REAL*8 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*8   M,N,K,LDA,LDB,LDC
REAL*8      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*8 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 SGEMM ('TRAN','NONTRAN',M,N,K,RHO,A,LDA,B,LDB,0.5,C,LDC)
```

Strassen Matrix-Matrix Multiply — SGEMMS/CGEMMS

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 which, under certain circumstances, uses fewer than mnk multiplications and additions. These subprograms are functionally equivalent to the SCILIB Level 3 BLAS subprograms SGEMM and CGEMM, and differ in usage only by the extra character in the subprogram name and the additional argument, **work**. By using Strassen's method, these subprograms may be considerably faster than their SCILIB counterparts. Refer to "Notes" for details.

In addition to computing the matrix-matrix product AB, A may be replaced by A^T or A^*, where A is a k-by-m matrix, and B may 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 may be stored in the result matrix (which is always of size m by n) 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 forms

$$\begin{array}{lll} C \leftarrow \alpha AB + \beta C, & C \leftarrow \alpha A^T B + \beta C, & C \leftarrow \alpha A^* B + \beta C, \\ C \leftarrow \alpha AB^T + \beta C, & C \leftarrow \alpha A^T B^T + \beta C, & C \leftarrow \alpha A^* B^T + \beta C, \\ C \leftarrow \alpha AB^* + \beta C, & C \leftarrow \alpha A^T B^* + \beta C, & C \leftarrow \alpha A^* B^* + \beta C. \end{array}$$

Usage

SCILIB:

```
CHARACTER*1 transa, transb
INTEGER*8   m, n, k, lda, ldb, ldc
REAL*8      alpha, beta, a(lda, *), b(ldb, *), c(ldc, n),
            work(lwork)
CALL SGEMMS (transa, transb, m, n, k, alpha, a, lda, b, ldb,
             beta, c, ldc, work)
CHARACTER*1 transa, transb
INTEGER*8   m, n, k, lda, ldb, ldc
COMPLEX*16  alpha, beta, a(lda, *), b(ldb, *), c(ldc, n),
            work(lwork)
CALL CGEMMS (transa, transb, m, n, k, alpha, a, lda, b, ldb,
             beta, c, ldc, work)
```

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'.

SGEMMS/CGEMMS — Continued

m Number of rows in matrix C, $\mathbf{m} \geq 0$. If $\mathbf{m} = 0$, the subprograms do not reference **a**, **b**, or **c**.

n Number of columns in matrix C, $\mathbf{n} \geq 0$. If $\mathbf{n} = 0$, the subprograms do not reference **a**, **b**, or **c**.

k The *middle* dimension of the matrix multiply, $\mathbf{k} \geq 0$. If $\mathbf{k} = 0$, the subprograms compute $C \leftarrow \beta C$ without referencing **a** or **b**.

alpha The scalar α. If **alpha** $= 0$, the subprograms compute $C \leftarrow \beta C$ without referencing **a** or **b**.

a Array containing the matrix A, whose size is indicated by **transa**:

'N' or 'n' A is an m-by-k matrix
otherwise A is a k-by-m matrix

lda The leading dimension of array **a** as declared in the calling program unit, with **lda** $\geq$ 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** $\geq$ max(**m**,1).

Working Storage

work An array of size **lwork** $= 2.34 \times$ max(**m**,**k**) $\times$ max(**n**,**k**), used for work space.

Output

c The updated C matrix replaces the input.

Notes

Except for the extra character in the subprogram name and the additional working storage argument, these subprograms conform to specifications of the Level 3 BLAS subprograms SGEMM and CGEMM.

Because of their use of Strassen's method, CGEMMS and SGEMMS are asymptotically faster than standard matrix multiply methods such as those employed in the SCILIB routines CGEMM and SGEMM. In practice these particular implementations are faster than their standard counterparts in the following cases:

- If there is a significant bank conflict problem stemming from the combination of TRANSA, TRANSB, LDA, and LDB. (A pleasant side-effect of the data motion supporting Strassen's method is the alleviation of that problem.)
- If $\min(m,n,k) > 200$ for CGEMMS and $\min(m,n,k) > 512$ for SGEMMS. 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. In this first release, the gains in the real case are marginal.

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.

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) may 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, and
ldc < max(**m**,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, 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. WORK is declared large enough to handle all matrices that will fit in the arrays.

```
CHARACTER*1 TRANSA,TRANSB
INTEGER*8   M,N,K,LDA,LDB,LDC
REAL*8      ALPHA,BETA,A(1000,1000),B(1000,1000),
1           C(1000,1000),WORK(2340000)
TRANSA = 'N'
TRANSB = 'N'
M = 900
N = 800
K = 600
ALPHA = 1.0
BETA = 0.0
LDA = 1000
LDB = 1000
LDC = 1000
CALL SGEMMS (TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,
1   C,LDC,WORK)
```

SGEMMS/CGEMMS — Continued

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. WORK is declared large enough to handle all matrices that will fit in the arrays.

```
INTEGER*8  M,N,K,LDA,LDB,LDC
COMPLEX*16 RHO,A(1000,1000),B(1000,1000),C(1000,1000),
1          WORK(2340000)
M = 900
N = 800
K = 600
LDA = 1000
LDB = 1000
LDC = 1000
CALL CGEMMS ('CONJ','NORMAL',M,N,K,RHO,A,LDA,B,LDB,
1   (0.5,0.0),C,LDC,WORK)
```

Matrix-Vector Multiply — SGEMV/CGEMV

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 may 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

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

Usage

SCILIB:

```
CHARACTER*1 trans
INTEGER*8   m, n, lda, incx, incy
REAL*8      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
COMPLEX*16  alpha, beta, a(lda, n), x(lenx), y(leny)
CALL CGEMV (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 $y \leftarrow \alpha A^Tx + \beta y$
'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 **a**, **x**, or **y**.

n Number of 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 containing the m-by-n matrix A.

lda The leading dimension of array **a** as declared in the calling program unit, with **lda** $\geq$ 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 m elements	**lenx** $= (\mathbf{m}-1)\times\lvert\mathbf{incx}\rvert+1$
otherwise	x has n elements	**lenx** $= (\mathbf{n}-1)\times\lvert\mathbf{incx}\rvert+1$

incx Increment for the array **x**, **incx** $\neq$ 0:

incx $>$ 0 x is stored forward in array **x**, i.e.,
x_i is stored in **x**((i-1)$\times$ **incx**+1).

incx $<$ 0 x is stored backward in array **x**, i.e.,
if **trans** = 'N' or 'n', then x_i is stored in **x**((i-**m**)$\times$ **incx**+1);
otherwise, x_i is stored in **x**((i-**n**)$\times$ **incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., 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 n elements	**leny** = (**n**–1)$\times$ |**incy**|+1
otherwise	y has m elements	**leny** = (**m**–1)$\times$ |**incy**|+1

Not used as input if **beta** = 0.

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

incy $>$ 0 y is stored forward in array **y**, i.e.,
y_i is stored in **y**((i-1)$\times$ **incy**+1).

incy $<$ 0 y is stored backward in array **y**, i.e.,
if **trans** = 'N' or 'n', then y_i is stored in **y**((i-**n**)$\times$ **incy**+1);
otherwise, y_i is stored in **y**((i-**m**)$\times$ **incy**+1).

Use **incy** = 1 if the vector y is stored contiguously in array **y**, i.e., 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) may be replaced with a user-supplied version to change the error procedure. Error conditions are

trans $\neq$ 'N' or 'n' or 'T' or 't' or 'C' or 'c',
m $<$ 0,
n $<$ 0,
lda $<$ max(**m**,1),
incx = 0, and
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 **trans** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Form the REAL*8 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*8   M,N,LDA,INCX,INCY
REAL*8      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*8 M,N,LDA
REAL*8    RHO,A(10,10),X(10),Y(10)
M = 9
N = 6
LDA = 10
CALL SGEMV ('TRANSPOSE',M,N,-RHO,A,LDA,X,1,0.5,Y,1)
```

SGER/CGERC/CGERU — Rank-1 Update

Purpose These subprograms compute the rank-1 updates

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

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.

Usage **SCILIB:**

```
INTEGER*8 m, n, lda, incx, incy
REAL*8    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
COMPLEX*16 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*16 alpha, a(lda, n), x(lenx), y(leny)
CALL CGERU (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** $= (\mathbf{m}-1)\times|\mathbf{incx}|+1$ containing the m-vector x.

incx Increment for the array **x**, **incx** $\neq 0$:

incx > 0 x is stored forward in array **x**, i.e., x_i is stored in **x**$((i-1)\times$**incx**$+1)$.
incx < 0 x is stored backward in array **x**, i.e., x_i is stored in **x**$((i-$**m**$)\times$**incx**$+1)$.

Use **incx** $= 1$ if the vector x is stored contiguously in array **x**, i.e., if x_i is stored in **x**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

y Array of length **leny** $= (\mathbf{n}-1)\times|\mathbf{incy}|+1$ containing the n-vector y. y is used in conjugated form by CGERC, and in unconjugated form by the other subprograms. Refer to "Purpose."

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

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

Use **incy** $= 1$ if the vector y is stored contiguously in array **y**, i.e., 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) may 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, and
incy = 0.

Example 1

Apply a REAL*8 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*8 M,N,LDA,INCX,INCY
REAL*8    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*16 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*8  M,N,LDA
COMPLEX*16 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)
```

SMXPY — Matrix-Vector Multiply and Add

Purpose

This subprogram computes the matrix-vector product Ax, and adds the result to another vector y, where A is an m-by-n matrix, x is an n-vector, and y is an m-vector. SCILIB subprogram SGEMV allows more general storage of x and y and also admits scaling, subtraction, and transposing A.

Usage

SCILIB:

```
INTEGER*8 m, n, lda
REAL*8    a(lda, n), x(n), y(m)
CALL SMXPY (m, y, n, lda, x, a)
```

Input

m Number of rows in matrix A and length of vector y, $\mathbf{m} \geq 0$. If $\mathbf{m} = 0$, the subprogram does not reference **a**, **x**, or **y**.

y Array containing the vector y.

n Number of columns in matrix A and length of vector x, $\mathbf{n} \geq 0$. If $\mathbf{n} = 0$, the subprogram does not reference **a**, **x**, or **y**.

lda The leading dimension of array **a** as declared in the calling program unit.

x Array containing the n-vector x.

a Array containing the m-by-n matrix A.

Output

y The updated y vector replaces the input.

Notes

Cray Research, Inc. has declared this subprogram obsolete in release 6.0 of the UNICOS Math and Scientific Library.

If an error in the arguments is detected, the subprograms call error handler XERSCI, which writes an error message onto the standard error file and terminates execution. The standard version of XERSCI (refer to Chapter 9, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are

$\mathbf{m} < 0$,
$\mathbf{n} < 0$, and
$\mathbf{lda} < \mathbf{m}$.

FORTRAN Equivalent

Except for the argument error checking, the following FORTRAN subroutine is equivalent to SMXPY.

```
      SUBROUTINE SMXPY (M,Y,N,LDA,X,A)
      INTEGER*8 M,N,LDA
      REAL*8 A(LDA,N),X(N),Y(M)
      DO 120 J = 1, N
        DO 110 I = 1, M
          Y(I) = A(I,J) * X(J) + Y(I)
  110   CONTINUE
  120 CONTINUE
      RETURN
      END
```

Example Form the REAL*8 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 of dimension 10.

```
INTEGER*8 M,N,LDA
REAL*8    A(10,10),X(10),Y(10)
M = 9
N = 6
LDA = 10
CALL SMXPY (M,Y,N,LDA,X,A)
```

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 may 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:

SSBMV A is a real symmetric band matrix
CHBMV 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.

Matrix Storage

Because it is not necessary to store or operate on the zeros outside the band of A, and since either triangle of A may be obtained from the other, you 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 $kd = 2$:

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

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

SCILIB:

```
CHARACTER*1 uplo
INTEGER*8   n, kd, ldab, incx, incy
REAL*8      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
COMPLEX*16  alpha, beta, ab(ldab, n), x(lenx), y(leny)
CALL CHBMV (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** $\neq$ 0:

incx > 0 x is stored forward in array **x**, i.e.,
x_i is stored in **x**((i–1)× **incx**+1).
incx < 0 x is stored backward in array **x**, i.e.,
x_i is stored in **x**((i–**n**)× **incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., 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** $\neq$ 0:

incy > 0 y is stored forward in array **y**, i.e.,
y_i is stored in **y**((i–1)× **incy**+1).
incy < 0 y is stored backward in array **y**, i.e.,
y_i is stored in **y**((i–**n**)× **incy**+1).

Use **incy** = 1 if the vector y is stored contiguously in array **y**, i.e., 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) may be replaced with a user-supplied version to change the error procedure. Error conditions are

uplo $\neq$ 'L' or 'l' or 'U' or 'u',
n < 0,
kd < 0,
ldab $<$ **kd**+1,
incx = 0, and
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*8 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*8   N,KD,LDAB,INCX,INCY
REAL*8      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*8 N,KD,LDAB
REAL*8    AB(25,100),X(100),Y(100)
N = 75
KD = 15
LDAB = 25
CALL SSBMV ('UPPER', N, KD, 1.0, AB, LDAB, X, 1, 1.0, Y, 1)
```

SSPMV/CHPMV — 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 may 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 A is a real symmetric matrix
CHPMV A is a complex Hermitian matrix

Matrix Storage

Because either triangle of A may 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

11	12	13	14
	22	23	24
		33	34
			44

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 $\mathbf{ap}(i + j \times (j-1)/2)$.

Lower triangular storage. If the lower triangle of A is

11			
21	22		
31	32	33	
41	42	43	44

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 $\mathbf{ap}(i + (j-1) \times (2n-j)/2)$.

Usage **SCILIB:**

```
CHARACTER*1 uplo
INTEGER*8   n, incx, incy
REAL*8      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
COMPLEX*16  alpha, beta, ap(lenap), x(lenx), y(leny)
CALL CHPMV (uplo, n, alpha, ap, x, incx, beta, y, incy)
```

Input

uplo Upper/lower triangular option for A:

'L' or 'l' The lower triangle of A is stored in the packed array.
'U' or 'u' The upper triangle of A is stored in the packed array.

n Number of rows and columns in matrix A, $\mathbf{n} \geq 0$. If $\mathbf{n} = 0$, the subprograms do not reference **ap**, **x**, or **y**.

alpha The scalar α. If $\mathbf{alpha} = 0$, the subprograms compute $y \leftarrow \beta y$ without referencing **ap** or **x**.

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.

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**, i.e., x_i is stored in $\mathbf{x}((i-1) \times \mathbf{incx}+1)$.
$\mathbf{incx} < 0$ x is stored backward in array **x**, i.e., 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**, i.e., if x_i is stored in $\mathbf{x}(i)$. Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

beta The scalar β.

y Array of length $\mathbf{leny} = (\mathbf{n}-1) \times |\mathbf{incy}| + 1$ containing the n-vector y. Not used as input if $\mathbf{beta} = 0$.

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

$\mathbf{incy} > 0$ y is stored forward in array **y**, i.e., y_i is stored in $\mathbf{y}((i-1) \times \mathbf{incy}+1)$.
$\mathbf{incy} < 0$ y is stored backward in array **y**, i.e., y_i is stored in $\mathbf{y}((i-\mathbf{n}) \times \mathbf{incy}+1)$.

Use $\mathbf{incy} = 1$ if the vector y is stored contiguously in array **y**, i.e., if y_i is stored in $\mathbf{y}(i)$. Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

SSPMV/CHPMV Continued

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) may be replaced with a user-supplied version to change the error procedure. Error conditions are

uplo $\neq$ 'L' or 'l' or 'U' or 'u',
n < 0,
incx $= 0$, and
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*8 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*8   N,INCX,INCY
REAL*8      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*16 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*8  N
COMPLEX*16 RHO,AP(55),X(10),Y(10)
N = 9
CALL CHPMV ('LOWER',N,-RHO,AP,X,1,(0.5,0.0),Y,1)
```

Rank-1 Update — SSPR/CHPR

Purpose

These subprograms compute the real symmetric or complex Hermitian rank-1 update

$$A \leftarrow \alpha xx^* + A,$$

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 A is a real symmetric matrix
CHPR A is a complex Hermitian matrix

Matrix Storage

Because either triangle of A may 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

11	12	13	14
	22	23	24
		33	34
			44

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 $\mathbf{ap}(i + j \times (j-1)/2)$.

Lower triangular storage. If the lower triangle of A is

11			
21	22		
31	32	33	
41	42	43	44

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 $\mathbf{ap}(i + (j-1) \times (2n-j)/2)$.

SSPR/CHPR — Continued

Usage **SCILIB:**

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

CHARACTER*1 uplo
INTEGER*8   n, incx
REAL*8      alpha
COMPLEX*16  ap(lenap), x(lenx)
CALL CHPR (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**, i.e., x_i is stored in $\mathbf{x}((i-1)\times\mathbf{incx}+1)$.
$\mathbf{incx} < 0$ x is stored backward in array **x**, i.e., 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**, i.e., 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) may be replaced with a user-supplied version to change the error procedure. Error conditions are

uplo ≠ 'L' or 'l' or 'U' or 'u',
$\mathbf{n} < 0$, and
$\mathbf{incx} = 0$.

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 Apply a REAL*8 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*8   N,INCX
REAL*8      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*16 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*8  N
COMPLEX*16 AP(55),X(10)
N = 9
CALL CHPR ('LOWER',N,-2.0,X,1,AP)
```

SSPR2/CHPR2 — 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 A is a real symmetric matrix
CHPR2 A is a complex Hermitian matrix

Matrix Storage

Because either triangle of A may 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

11	12	13	14
	22	23	24
		33	34
			44

then A is packed column-by-column into an array **ap** as follows:

k	1	2	3	4	5	6	7	8	9	10
ap(k)	11	12	22	13	23	33	14	24	34	44

Upper triangular matrix element a_{ij} is stored in array element **ap**$(i+j\times(j-1)/2)$.

Lower triangular storage. If the lower triangle of A is

11			
21	22		
31	32	33	
41	42	43	44

then A is packed column-by-column into an array **ap** as follows:

k	1	2	3	4	5	6	7	8	9	10
ap(k)	11	21	31	41	22	32	42	33	43	44

Lower triangular matrix element a_{ij} is stored in array element **ap**$(i+(j-1)\times(2n-j)/2)$.

Usage **SCILIB:**

```
CHARACTER*1 uplo
INTEGER*8   n, incx, incy
REAL*8      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
COMPLEX*16  alpha, ap(lenap), x(lenx), y(leny)
CALL CHPR2 (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 $\mathbf{lenx} = (\mathbf{n}-1)\times|\mathbf{incx}|+1$ 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**, i.e., x_i is stored in $\mathbf{x}((i-1)\times\mathbf{incx}+1)$.
$\mathbf{incx} < 0$ x is stored backward in array **x**, i.e., 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**, i.e., if x_i is stored in $\mathbf{x}(i)$. Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

y Array of length $\mathbf{leny} = (\mathbf{n}-1)\times|\mathbf{incy}|+1$ containing the n-vector y.

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

$\mathbf{incy} > 0$ y is stored forward in array **y**, i.e., y_i is stored in $\mathbf{y}((i-1)\times\mathbf{incy}+1)$.
$\mathbf{incy} < 0$ y is stored backward in array **y**, i.e., y_i is stored in $\mathbf{y}((i-\mathbf{n})\times\mathbf{incy}+1)$.

Use $\mathbf{incy} = 1$ if the vector y is stored contiguously in array **y**, i.e., if y_i is stored in $\mathbf{y}(i)$. 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.

SSPR2/CHPR2 Continued

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) may 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, and
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

Apply a REAL*8 symmetric rank-2 update $xy^T + x^T y$ 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*8   N,INCX,INCY
REAL*8      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*16 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*8  N
COMPLEX*16 ALPHA,AP(55),X(10),Y(10)
N = 9
CALL CHPR2 ('LOWER',N,ALPHA,X,1,Y,1,AP)
```

Matrix-Matrix Multiply — SSYMM/CHEMM/CSYMM

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 may be stored in the result matrix (which is always of size m by n) 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 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	A is a real symmetric matrix
CHEMM	A is a complex Hermitian matrix
CSYMM	A is a complex symmetric matrix

Matrix Storage

Because either triangle of A may be obtained from the other, you only need to provide one triangle of A. You may 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

SCILIB:

```
CHARACTER*1 side, uplo
INTEGER*8   m, n, lda, ldb, ldc
REAL*8      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
COMPLEX*16  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*16  alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL CSYMM (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, i.e. compute $C \leftarrow \alpha AB + \beta C$

'R' or 'r' A is the right matrix operand, i.e. 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, $\mathbf{m} \geq 0$. If $\mathbf{m} = 0$, the subprograms do not reference **a**, **b**, or **c**.

n Number of columns in matrix B, $\mathbf{n} \geq 0$. If $\mathbf{n} = 0$, the subprograms do not reference **a**, **b**, or **c**.

alpha The scalar α. If **alpha** $= 0$, the subprograms compute $C \leftarrow \beta C$ without referencing **a** or **b**.

a Array whose upper or lower triangle, as specified by **uplo**, contains the upper or lower triangle of the matrix A. The other triangle of **a** is not referenced. The size of A is indicated by **side**:

'L' or 'l'	A is m by m
'R' or 'r'	A is n by n

lda The leading dimension of array **a** as declared in the calling program unit, with **lda** $\geq$ max(the number of rows of A,1).

b Array containing the m-by-n matrix B.

ldb The leading dimension of array **b** as declared in the calling program unit, with **ldb** $\geq$ max(**m**,1).

beta The scalar β.

c Array containing the m-by-n matrix C. Not used as input if **beta** $= 0$.

ldc The leading dimension of array **c** as declared in the calling program unit, with **ldc** $\geq$ 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) may be replaced with a user-supplied version to change the error procedure. Error conditions are

side $\neq$ 'L' or 'l' or 'R' or 'r',
uplo $\neq$ 'L' or 'l' or 'U' or 'u',
m < 0,
n < 0,
lda too small,
ldb $<$ max(**m**,1), and
ldc $<$ max(**m**,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, for example, by coding the **side** argument as 'LEFT' for 'L' or 'RIGHT' for 'R'. Refer to "Example 2."

Example 1 Form the REAL*8 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*8   M,N,LDA,LDB,LDC
REAL*8      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*16 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*8  M,N,LDA,LDB,LDC
COMPLEX*16 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)
```

SSYMV/CHEMV — 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 may 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	A is a real symmetric matrix
CHEMV	A is a complex Hermitian matrix

Matrix Storage

Because either triangle of A may be obtained from the other, you only need to provide one triangle of A. You may 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

SCILIB:

```
CHARACTER*1 uplo
INTEGER*8   n, lda, incx, incy
REAL*8      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
COMPLEX*16  alpha, beta, a(lda, n), x(lenx), y(leny)
CALL CHEMV (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, **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 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** $\geq$ max(**n**,1).

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

incx Increment for the array **x**, **incx** $\neq$ 0:

incx > 0 x is stored forward in array **x**, i.e.,
x_i is stored in **x**((i-1)$\times$ **incx**+1).
incx < 0 x is stored backward in array **x**, i.e.,
x_i is stored in **x**((i-**n**)$\times$ **incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., 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)$\times$|**incy**|+1 containing the n-vector y. Not used as input if **beta** = 0.

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

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

Use **incy** = 1 if the vector y is stored contiguously in array **y**, i.e., 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) may be replaced with a user-supplied version to change the error procedure. Error conditions are

uplo $\neq$ 'L' or 'l' or 'U' or 'u',
n < 0,
lda $<$ max(**n**,1),
incx = 0, and
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."

SSYMV/CHEMV — Continued

Example 1 Form the REAL*8 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*8   N,LDA,INCX,INCY
REAL*8      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*16 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*8  N,LDA
COMPLEX*16 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)
```

Rank-1 Update — SSYR/CHER

Purpose

These subprograms compute the real symmetric or complex Hermitian rank-1 update

$$A \leftarrow \alpha xx^* + A,$$

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	A is a real symmetric matrix
CHER	A is a complex Hermitian matrix

Matrix Storage

Because either triangle of A may be obtained from the other, these subprograms reference and apply the update to only one triangle of A. You may 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

SCILIB:

```
CHARACTER*1 uplo
INTEGER*8   n, lda, incx
REAL*8      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
COMPLEX*16  a(lda, n), x(lenx)
CALL CHER (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**, $\mathbf{incx} \neq 0$:

$\mathbf{incx} > 0$	x is stored forward in array **x**, i.e., x_i is stored in $\mathbf{x}((i-1)\times\mathbf{incx}+1)$.
$\mathbf{incx} < 0$	x is stored backward in array **x**, i.e., 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**, i.e., if x_i is stored in $\mathbf{x}(i)$. Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

a Array whose upper or lower triangle, as specified by **uplo**, contains the upper or lower triangle of an n-by-n real symmetric or complex Hermitian matrix A. The other triangle of **a** is not referenced.

lda The leading dimension of array **a** as declared in the calling program unit, with **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) may 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), and
incx = 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

Apply a REAL*8 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*8   N,LDA,INCX
REAL*8      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*16 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*8  N,LDA
COMPLEX*16 A(10,10),X(10)
N = 9
LDA = 10
CALL CHER ('LOWER',N,-2.0,X,1,A,LDA)
```

Rank-2 Update — SSYR2/CHER2

Purpose

These subprograms compute the real symmetric or complex Hermitian rank-2 update

$$A \leftarrow \alpha xy^* + \bar{\alpha} yx^* + A,$$

where A is an n-by-n real symmetric or complex Hermitian matrix, α is a complex scalar, $\bar{\alpha}$ is the complex conjugate of α, x and y are real or complex n-vectors, and x^* and y^* are the conjugate transposes of x and y, respectively. (The conjugate of a real scalar is just the scalar, and the conjugate transpose of a real vector is simply the transpose.)

The structure of A is indicated by the name of the subprogram used:

SSYR2 A is a real symmetric matrix
CHER2 A is a complex Hermitian matrix

Matrix Storage

Because either triangle of A may be obtained from the other, these subprograms reference and apply the update to only one triangle of A. You may 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

SCILIB:

```
CHARACTER*1 uplo
INTEGER*8   n, lda, incx, incy
REAL*8      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
COMPLEX*16  alpha, a(lda, n), x(lenx), y(leny)
CALL CHER2 (uplo, n, alpha, x, incx, y, incy, a, lda)
```

Input

uplo Upper/lower triangular option for A:

'L' or 'l' Reference and update only the lower triangle of A.
'U' or 'u' Reference and update only the upper triangle of A.

n Number of rows and columns in matrix A and elements of vectors x and y, **n** ≥ 0. If **n** = 0, the subprograms do not reference **a**, **x**, or **y**.

alpha The scalar α. If **alpha** = 0, the subprograms do not reference **a**, **x**, or **y**.

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

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

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

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., 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**, i.e., y_i is stored in **y**((i–1)× **incy**+1).

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

Use **incy** = 1 if the vector y is stored contiguously in array **y**, i.e., 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) may 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, and
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'.

Example 1

Apply a REAL*8 symmetric rank-2 update $xy^T + x^T y$ 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*8   N,LDA,INCX,INCY
REAL*8      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*16 Hermitian rank-2 update $\alpha xy^* + \overline{\alpha} yx^*$ 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*8  N,LDA
COMPLEX*16 ALPHA,A(10,10),X(10),Y(10)
N = 9
LDA = 10
CALL CHER2 ('LOWER',N,ALPHA,X,1,Y,1,A,LDA)
```

SSYR2K/CHER2K/CSYR2K — 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: $C \leftarrow \alpha AB^T + \bar{\alpha}BA^T + \beta C$ and $C \leftarrow \alpha A^TB + \bar{\alpha}B^TA + \beta C$

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:

Matrix Storage

Because either triangle of C may be obtained from the other, these subprograms reference and apply the update to only one triangle of C. You may 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

SCILIB:

```
CHARACTER*1 uplo, trans
INTEGER*8   n, k, lda, ldb, ldc
REAL*8      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
COMPLEX*16  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*16  alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL CSYR2K (uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
```

Input

uplo Upper/lower triangular storage option for C:

'L' or 'l'	Reference and update only the lower triangle of C
'U' or 'u'	Reference and update only the upper triangle of C

trans Specifies the operation to be performed:

'N' or 'n'	Compute $C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$
'T' or 't'	Compute $C \leftarrow \alpha A^TB + \alpha B^TA + \beta C$
'C' or 'c'	Compute $C \leftarrow \alpha A^*B + \alpha B^*A + \beta C$

'T' and 't' are invalid in subprogram CHER2K, and 'C' and 'c' are invalid in subprogram CSYR2K. In subprogram SSYR2K, 'C' and 'c' have the same meaning as 'T' and 't'.

n Number of rows and columns in matrix C, **n** $\geq$ 0. If **n** = 0, the subprograms do not reference **a**, **b**, or **c**.

k Number of rows or columns in matrices A and B, depending on **trans**; refer to the description of **a** for details. **k** $\geq$ 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** $\geq$ 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** above for details.

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 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** $\geq$ 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) may be replaced with a user-supplied version to change the error procedure. Error conditions are

uplo $\neq$ 'L' or 'l' or 'U' or 'u',
trans $\neq$ 'N' or 'n' or 'T' or 't' or 'C' or 'c',
n $<$ 0,
k $<$ 0,
lda too small,
ldb too small, and
ldc $<$ max(**m**,1).

Also, note that some of the values of **trans** listed above are invalid in subprograms CHER2K and CSYR2K.

SSYR2K/CHER2K/CSYR2K — Continued

Actual character arguments in a subroutine call may be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved, for example, by coding the **uplo** argument as 'LOWER' for 'L' or 'UPPER' for 'U'. Refer to "Example 2."

Example 1 Apply a REAL*8 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*8   N,K,LDA,LDB,LDC
REAL*8      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*16 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*8  N,K,LDA,LDB,LDC
COMPLEX*16 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)
```

Rank-k Update — SSYRK/CHERK

Purpose

These subprograms apply a rank-k update to a real symmetric, complex symmetric, or complex Hermitian matrix; specifically they compute:

$$C \leftarrow \alpha AA^T + \beta C, \quad C \leftarrow \alpha A^T A + \beta C,$$
$$C \leftarrow \alpha AA^* + \beta C, \quad C \leftarrow \alpha A^* A + \beta C,$$

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	C is a real symmetric matrix
CHERK	C is a complex Hermitian matrix
CSYRK	C is a complex symmetric matrix

Matrix Storage

Because either triangle of C may be obtained from the other, these subprograms reference and apply the update to only one triangle of C. You may 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

SCILIB:

```
CHARACTER*1 uplo, trans
INTEGER*8   n, k, lda, ldc
REAL*8      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
COMPLEX*16  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*16  alpha, beta, a(lda, *), c(ldc, n)
CALL CSYRK (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 subprogram CHER2K, and 'C' and 'c' are invalid in subprogram CSYR2K. In subprogram SSYRK, 'C' and 'c' have the same meaning as 'T' and 't'.

n Number of rows and columns in matrix C, **n** $\geq$ 0. If **n** = 0, the subprograms do not reference **a** or **c**.

k Number of rows or columns in matrix A, **k** $\geq$ 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** $\geq$ 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** $\geq$ 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) may be replaced with a user-supplied version to change the error procedure. Error conditions are

uplo $\neq$ 'L' or 'l' or 'U' or 'u',
trans $\neq$ 'N' or 'n' or 'T' or 't' or 'C' or 'c',
n $<$ 0,
k $<$ 0,
lda too small, and
ldc $<$ max(**m**,1).

Also, some values of **trans** listed above are invalid in subprograms CHERK and CSYRK.

Actual character arguments in a subroutine call may be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved, for example, by coding the **uplo** argument as 'LOWER' for 'L' or 'UPPER' for 'U'. Refer to "Example 2."

Example 1 Apply a REAL*8 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*8   N,K,LDA,LDC
REAL*8      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*16 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*8  N,K,LDA,LDC
COMPLEX*16 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)
```

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, $A^T x$, 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

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

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, i.e., with $a_{ij} \neq 0$ only if $0 \leq j-i \leq kd$.

Matrix Storage

Triangular band matrices are stored in a compressed form that takes advantage of knowing the positions of the only elements that may 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

SCILIB:

```
CHARACTER*1 uplo, trans, diag
INTEGER*8   n, kd, ldab, incx
REAL*8      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
COMPLEX*16  ab(ldab, n), x(lenx)
CALL CTBMV (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, i.e., $a_{ii} = 1$, 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** $\geq$ 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** $\geq$ **kd**+1.

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

incx Increment for the array **x**, **incx** $\neq$ 0:

incx > 0 x is stored forward in array **x**, i.e., x_i is stored in **x**((i–1)×**incx**+1).
incx < 0 x is stored backward in array **x**, i.e., x_i is stored in **x**((i–**n**)×**incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., 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) may be replaced with a user-supplied version to change the error procedure. Error conditions are

uplo $\neq$ 'L' or 'l' or 'U' or 'u',
trans $\neq$ 'N' or 'n' or 'T' or 't' or 'C' or 'c',
diag $\neq$ 'N' or 'n' or 'U' or 'u',
n < 0,
kd < 0,
ldab < **kd**+1, and
incx = 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 **trans** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Form the REAL*8 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*8   N,KD,LDAB,INCX
REAL*8      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 $A^T x$, 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*8 N,KD,LDAB
REAL*8    AB(25,100),X(100)
N = 75
KD = 15
LDAB = 25
CALL STBMV ('UPPER','TRANSPOSE','NONUNIT',N,KD,AB,LDAB,X,1)
```

STBSV/CTBSV — 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 is the forward elimination or back substitution step of Gaussian elimination for band matrices. Optionally, A may 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

$$x \leftarrow A^{-1}x, \quad x \leftarrow A^{-T}x, \quad \text{and} \quad x \leftarrow A^{-*}x$$

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 LINPACK band equation solvers. As such, they are intended to supplement but not replace the equation solvers, serving instead as building blocks in constructing optimized linear algebra software. In fact, many of the LINPACK subprograms have been recoded to call these routines.

Matrix Storage

Triangular band matrices are stored in a compressed form that takes advantage of knowing the positions of the only elements that may 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

SCILIB:

```
CHARACTER*1 uplo, trans, diag
INTEGER*8   n, kd, ldab, incx
REAL*8      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
COMPLEX*16  ab(ldab, n), x(lenx)
CALL CTBSV (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, i.e., $a_{ii} = 1$, or not:

'N' or 'n' The diagonal of A is stored in the array
'U' or 'u' The diagonal of A consists of unstored ones

When **diag** is supplied as 'U' or 'u', diagonal elements of A are not referenced, but space must be reserved for them.

n Number of rows and columns in matrix A, $\mathbf{n} \geq 0$. If $\mathbf{n} = 0$, the subprograms do not reference **ab** or **x**.

kd The number of nonzero diagonals above or below the principal diagonal. If **uplo** is supplied as 'U' or 'u', **kd** specifies the number of nonzero diagonals above the principal diagonal. If **uplo** is supplied as 'L' or 'l', **kd** specifies the number of nonzero diagonals below the principal diagonal.

ab Array containing the n-by-n triangular band matrix A in the compressed form described above. The columns of the band of A are stored in the columns of **ab**, and the diagonals of the band of A are stored in the rows of **ab**.

ldab The leading dimension of array **ab** as declared in the calling program unit, with $\mathbf{ldab} \geq \mathbf{kd}+1$.

x Array of length $\mathbf{lenx} = (\mathbf{n}-1)\times|\mathbf{incx}|+1$ 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**, i.e., x_i is stored in $\mathbf{x}((i-1)\times\mathbf{incx}+1)$.
$\mathbf{incx} < 0$ x is stored backward in array **x**, i.e., 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**, i.e., 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 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) may 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, and
incx = 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 **trans** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Perform REAL*8 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*8   N,KD,LDAB,INCX
REAL*8      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*8 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*8 N,KD,LDAB
REAL*8    AB(25,100),X(100)
N = 75
KD = 15
LDAB = 25
CALL STBSV ('UPPER','NONTRANS','NONUNIT',N,KD,AB,LDAB,X,1)
```

STPMV/CTPMV — 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

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

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

11	12	13	14
0	22	23	24
0	0	33	34
0	0	0	44

then A is packed column by column into an array **ap** as follows:

k	1	2	3	4	5	6	7	8	9	10
ap(k)	11	12	22	13	23	33	14	24	34	44

Upper-triangular matrix element a_{ij} is stored in array element **ap**$(i + j \times (j-1)/2)$.

Lower triangular matrix. If A is

11	0	0	0
21	22	0	0
31	32	33	0
41	42	43	44

then A is packed column by column into an array **ap** as follows:

k	1	2	3	4	5	6	7	8	9	10
ap(k)	11	21	31	41	22	32	42	33	43	44

Lower-triangular matrix element a_{ij} is stored in array element **ap**$(i + (j-1) \times (2n-j)/2)$.

Usage

SCILIB:

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

CHARACTER*1 uplo, trans, diag
INTEGER*8   n, incx
COMPLEX*16  ap(lenap), x(lenx)
CALL CTPMV (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, i.e., $a_{ii} = 1$, 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 input vector x.

incx Increment for the array **x**, $\mathbf{incx} \neq 0$:

$\mathbf{incx} > 0$ x is stored forward in array **x**, i.e., x_i is stored in $\mathbf{x}((i-1) \times \mathbf{incx}+1)$.
$\mathbf{incx} < 0$ x is stored backward in array **x**, i.e., 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**, i.e., if x_i is stored in $\mathbf{x}(i)$. Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

Output

x The updated x vector replaces the input.

STPMV/CTPMV Continued

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) may 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, and
incx $= 0$.

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 **trans** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1

Form the REAL*8 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*8   N,INCX
REAL*8      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 $A^T x$, 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*8 N
REAL*8    AP(55),X(10)
N = 6
CALL STPMV ('UPPER','TRANSPOSE','NONUNIT',N,AP,X,1)
```

Solve Triangular System — STPSV/CTPSV

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 may be replaced by A^T, the transpose of A, or by A^*, the conjugate transpose of A. Specifically, these subprograms compute

$$x \leftarrow A^{-1}x, \quad x \leftarrow A^{-T}x, \quad \text{and} \quad x \leftarrow A^{-*}x$$

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 LINPACK linear equation solvers. As such, they are intended to supplement but not replace the equation solvers, serving instead as building blocks in constructing optimized linear algebra software. In fact, many of the LINPACK subprograms have been recoded to call these 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**$(i + j \times (j-1)/2)$.

Lower triangular matrix. If A is

$$\begin{matrix} 11 & 0 & 0 & 0 \\ 21 & 22 & 0 & 0 \\ 31 & 32 & 33 & 0 \\ 41 & 42 & 43 & 44 \end{matrix}$$

then A is packed column by column into an array **ap** as follows:

k	1	2	3	4	5	6	7	8	9	10
ap(k)	11	21	31	41	22	32	42	33	43	44

Lower-triangular matrix element a_{ij} is stored in array element **ap**$(i + (j-1) \times (2n-j)/2)$.

STPSV/CTPSV — Continued

Usage **SCILIB:**

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

CHARACTER*1 uplo, trans, diag
INTEGER*8   n, incx
COMPLEX*16  ap(lenap), x(lenx)
CALL CTPSV (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, i.e., $a_{ii} = 1$, 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** $\geq$ 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 right-hand-side n-vector x.

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

incx > 0 x is stored forward in array **x**, i.e., x_i is stored in **x**((*i*–1)×**incx**+1).
incx < 0 x is stored backward in array **x**, i.e., x_i is stored in **x**((*i*–**n**)×**incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., 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 will cause 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) may 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, and
incx = 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 **trans** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Perform REAL*8 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*8   N,INCX
REAL*8      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*8 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*8 N
REAL*8    AP(5500),X(100)
N = 75
CALL STPSV ('UPPER','NONTRANS','NONUNIT',N,AP,X,1)
```

STRMM/CTRMM — 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 may 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 will not be referenced.

Usage

SCILIB:

```
CHARACTER*1 side, uplo, transa, diag
INTEGER*8   m, n, lda, ldb
REAL*8      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
COMPLEX*16  alpha, a(lda, *), b(ldb, *)
CALL CTRMM (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, i.e., $a_{ii} = 1$, or not:

'N' or 'n'	The diagonal of A is stored in the array
'U' or 'u'	The diagonal of A consists of unstored ones

When **diag** is supplied as 'U' or 'u', the diagonal elements of A are not referenced.

m Number of rows in matrix B, $\mathbf{m} \geq 0$. If $\mathbf{m} = 0$, the subprograms do not reference **a** or **b**.

n Number of columns in matrix B, **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** $\geq$ 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** $\geq$ 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) may be replaced with a user-supplied version to change the error procedure. Error conditions are

side $\neq$ 'L' or 'l' or 'R' or 'r',
uplo $\neq$ 'L' or 'l' or 'U' or 'u',
transa $\neq$ 'N' or 'n' or 'T' or 't' or 'C' or 'c',
diag $\neq$ 'N' or 'n' or 'U' or 'u',
m < 0,
n < 0,
lda too small, and
ldb $<$ max(**m**,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, for example, by coding the **transa** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

STRMM/CTRMM Continued

Example 1 Form the REAL*8 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 will overwrite the input B matrix.

```
CHARACTER*1 SIDE,UPLO,TRANSA,DIAG
INTEGER*8   M,N,LDA,LDB
REAL*8      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 will overwrite the input B matrix.

```
INTEGER*8 M,N,LDA,LDB
REAL*8    Q,A(10,10),B(10,10)
M = 6
N = 8
LDA = 10
LDB = 10
CALL STRMM ('RIGHT','LOWER','TRANS','UNIT',M,N,Q,A,LDA,B,LDB)
```

Matrix-Vector Multiply — STRMV/CTRMV

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

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

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 will not be referenced.

Usage

SCILIB:

```
CHARACTER*1 uplo, trans, diag
INTEGER*8   n, lda, incx
REAL*8      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
COMPLEX*16  a(lda, n), x(lenx)
CALL CTRMV (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 $x \leftarrow A^Tx$
'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, i.e., $a_{ii} = 1$, 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 **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**, i.e., x_i is stored in **x**((i–1)× **incx**+1).
incx < 0 x is stored backward in array **x**, i.e., x_i is stored in **x**((i–**n**)× **incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., 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) may 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), and
incx = 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 **trans** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1

Form the REAL*8 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*8   N,LDA,INCX
REAL*8      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 $A^T x$, 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*8 N,LDA
REAL*8    A(10,10),X(10)
N = 6
LDA = 10
CALL STRMV ('UPPER','TRANSPOSE','NONUNIT',N,A,LDA,X,1)
```

STRSM/CTRSM — 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} may be replaced by A^{-T}, the inverse of the transpose of A, or by A^{-*}, the inverse of the conjugate transpose of A. The resulting matrix solution overwrites the input B matrix. Specifically, these subprograms compute matrix solutions of the forms

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

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 will not be referenced.

Usage

SCILIB:

```
CHARACTER*1 side, uplo, transa, diag
INTEGER*8   m, n, lda, ldb
REAL*8      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
COMPLEX*16  alpha, a(lda, *), b(ldb, *)
CALL CTRSM (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, i.e., $a_{ii} = 1$, 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** $\geq$ 0. If **m** = 0, the subprograms do not reference **a** or **b**.

n Number of columns in matrix B, **n** $\geq$ 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** $\geq$ 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** $\geq$ 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) may be replaced with a user-supplied version to change the error procedure. Error conditions are

side $\neq$ 'L' or 'l' or 'R' or 'r',
uplo $\neq$ 'L' or 'l' or 'U' or 'u',
transa $\neq$ 'N' or 'n' or 'T' or 't' or 'C' or 'c',
diag $\neq$ 'N' or 'n' or 'U' or 'u',
m $<$ 0,
n $<$ 0,
lda too small, and
ldb $<$ max(**m**,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, for example, by coding the **transa** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

STRSM/CTRSM Continued

Example 1 Form the REAL*8 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 will overwrite the input B matrix.

```
CHARACTER*1 SIDE,UPLO,TRANSA,DIAG
INTEGER*8   M,N,LDA,LDB
REAL*8      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 will overwrite the input B matrix.

```
INTEGER*8 M,N,LDA,LDB
REAL*8    Q,A(10,10),B(10,10)
M = 6
N = 8
LDA = 10
LDB = 10
CALL STRSM ('RIGHT','LOWER','TRANS','UNIT',M,N,Q,A,LDA,B,LDB)
```

Solve Triangular System — STRSV/CTRSV

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 may be replaced by A^T, the transpose of A, or by A^*, the conjugate transpose of A. Specifically, these subprograms compute

$$x \leftarrow A^{-1}x, \quad x \leftarrow A^{-T}x, \quad \text{and} \quad x \leftarrow A^{-*}x$$

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 LINPACK linear equation solvers. As such, they are intended to supplement but not replace the equation solvers, serving instead as building blocks in constructing optimized linear algebra software. In fact, many of the LINPACK subprograms have been recoded to call these 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 will not be referenced.

Usage

SCILIB:

```
CHARACTER*1 uplo, trans, diag
INTEGER*8   n, lda, incx
REAL*8      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
COMPLEX*16  a(lda, n), x(lenx)
CALL CTRSV (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, i.e., $a_{ii} = 1$, 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** $\geq$ 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** $\geq$ max(**n**,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** $\neq$ 0:

incx > 0	x is stored forward in array **x**, i.e., x_i is stored in **x**((i–1)× **incx**+1).
incx < 0	x is stored backward in array **x**, i.e., x_i is stored in **x**((i–**n**)× **incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., 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 will cause 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) may be replaced with a user-supplied version to change the error procedure. Error conditions are

uplo $\neq$ 'L' or 'l' or 'U' or 'u',
trans $\neq$ 'N' or 'n' or 'T' or 't' or 'C' or 'c',
diag $\neq$ 'N' or 'n' or 'U' or 'u',
n < 0,
lda < max(**n**,1), and
incx = 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 **trans** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Perform REAL*8 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*8   N,LDA,INCX
REAL*8      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*8 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*8 N,LDA
REAL*8    A(100,100),X(100)
N = 75
LDA = 100
CALL STRSV ('UPPER','NONTRANS','NONUNIT',N,A,LDA,X,1)
```

SXMPY — Vector-Matrix Multiply and Add

Purpose

This subprogram computes the matrix-vector product xA, and adds the result to another vector y, where A is an m-by-n matrix, x is an m-dimensional row vector, and y is an n-dimensional row vector. SCILIB subprogram SGEMV allows more general storage of x and y and also admits scaling, subtraction, and transposing A.

Usage

SCILIB:

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

Input

n Number of columns in matrix A and length of row vector y, **n** ≥ 0. If **n** = 0, the subprogram does not reference **a**, **x**, or **y**.

incy Storage increment between successive elements of vector y in array **y**. y_i is stored in **y**((*i*-1)× **incy** + 1). Use **incy** = 1 if the vector y is stored contiguously in array **y**, i.e., if y_i is stored in **y**(*i*).

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

m Number of rows in matrix A and length of row vector x, **m** ≥ 0. If **m** = 0, the subprogram does not reference **a**, **x**, or **y**.

incx Storage increment between successive elements of vector x in array **x**. x_i is stored in **x**((*i*-1)× **incx** + 1). Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., if x_i is stored in **x**(*i*).

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

lda The leading dimension of array **a** as declared in the calling program unit.

a Array containing the m-by-n matrix A.

Output

y The updated y vector replaces the input.

Notes

Cray research, Inc. has declared this subprogram obsolete in release 6.0 of the UNICOS Math and Scientific Library.

If an error in the arguments is detected, the subprograms call error handler XERSCI, which writes an error message onto the standard error file and terminates execution. The standard version of XERSCI (refer to Chapter 9, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are

m < 0,
n < 0,
lda < **m**,
incx ≤ 0, and
incy ≤ 0.

FORTRAN Equivalent Except for the argument error checking, the following FORTRAN subroutine is equivalent to SXMPY.

```
      SUBROUTINE SXMPY (N,INCY,Y,M,INCX,X,LDA,A)
      INTEGER*8 M,N,LDA,INCX,INCY
      REAL*8 A(LDA,N),X(*),Y(*)
      DO 120 J = 1, M
         DO 110 I = 1, N
            Y((I-1)*INCY+1) =          ! Y(I) =
     1      Y((I-1)*INCY+1) +          ! Y(I) +
     2      X((J-1)*INCX+1) * A(J,I)   ! X(J) * A(J,I)
  110    CONTINUE
  120 CONTINUE
      RETURN
      END
```

Example Form the REAL*8 vector-matrix product $y = y + xA$, 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 9 elements long stored in row 3 of an array X of dimension 10 by 10, and y is a real vector 6 elements long stored in row 7 of an array Y of dimension 10 by 10.

```
INTEGER*8 M,N,LDA,INCX,INCY
REAL*8    A(10,10),X(10,10),Y(10,10)
M = 9
N = 6
LDA = 10
INCX = 10
INCY = 10
CALL SXMPY (N,INCY,Y(7,1),M,INCX,X(3,1),LDA,A)
```

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 SCILIB, 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. If the main program is in FORTRAN, a call traceback is also written onto the standard error file. XERBLA then terminates execution with a nonzero exit status.

You may supply a version of XERBLA that alters this action. Be aware that other subprograms, including many in CONVEX LAPACK, also call XERBLA. All BLAS, VECLIB, SCILIB, 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

SCILIB:

```
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.

4

Linear Equations

Overview

This chapter describes the LINPACK software library included with SCILIB. The most important subprograms in this library have been upgraded by incorporating the Level 2 and Level 3 BLAS and other algorithmic changes. Although SCILIB includes all LINPACK subprograms, only those subprograms optimized for use on CONVEX supercomputers are described in this chapter. Table 4-5 at the end of this chapter lists the LINPACK subprograms that are included in SCILIB but are not documented in the *CONVEX SCILIB User's Guide*. You may find information for these subprograms in the *LINPACK Users' Guide* included in the SCILIB documentation set.

The LAPACK software library included with SCILIB is a comprehensive collection of linear equation solvers and subprograms for other linear algebra computations. This software is documented in the *CONVEX LAPACK User's Guide*. We recommend that you use LAPACK subprograms rather than LINPACK subprograms in new programs. Future optimization efforts will be directed to LAPACK rather than LINPACK.

This chapter explains how to use SCILIB subprograms to solve systems of linear equations. The operations covered are:

- solution of a system of linear equations
- calculation of the inverse of a matrix
- evaluation of the determinant of a matrix

These operations are performed for a variety of types of matrices, including:

- real and complex general dense matrices
- real and complex general band matrices
- real and complex positive definite dense matrices
- real and complex positive definite band matrices
- real and complex general tridiagonal matrices
- real and complex positive definite tridiagonal matrices

Refer to Chapter 6 for software to solve sparse symmetric linear equations.

Chapter Objectives

After you read this chapter you will:

- be familiar with the LINPACK subroutine naming convention
- understand the role of the condition number in solving linear equations
- know how to compute the determinant or inverse of a matrix
- know when not to compute the determinant or inverse of a matrix
- be able to locate documentation for LINPACK subroutines not documented here
- know how to use the described subprograms

What You Need to Know to Use These Subprograms

Subroutine Naming Convention

LINPACK uses a subroutine naming convention that encodes the function of each subroutine into its name. LINPACK subprogram names consist of five letters in the form TXXYY.

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

Table 4-1: LINPACK Naming Convention — Data Type

T	Data Type
S	Single Precision REAL
C	Single Precision COMPLEX

Table 4-2 shows the next two letters in the naming convention which indicate the form of the matrix or its decomposition.

Table 4-2: LINPACK Naming Convention — Form or Decomposition

XX	Form or Decomposition
GE	General
GB	General band
PO	Positive definite
PB	Positive definite band
PP	Positive definite packed
SI	Symmetric indefinite
SP	Symmetric indefinite packed
HI	Hermitian indefinite
HP	Hermitian indefinite packed
TR	Triangular
GT	General tridiagonal
PT	Positive definite tridiagonal
CH	Cholesky decomposition
QR	Orthogonal-triangular decomposition
SV	Singular value decomposition

Table 4-3 lists the final two letters in the naming convention which indicate the computation of a particular subroutine.

Table 4-3: LINPACK Naming Convention — Computation

YY	Subroutine Computation
FA	Factor
CO	Factor and estimate condition
SL	Solve
DI	Determinant and/or inverse and/or inertia
DC	Decompose
UD	Update
DD	Downdate
EX	Exchange

For example, SGBCO factors a general band (GB) matrix and estimates its condition number (CO) using the single precision REAL data type (S). CGEFA calculates the factorization (FA) of a general dense matrix (GE) using the single precision COMPLEX data type (C).

Table 4-4 shows the valid combinations of T, XX, and YY. Each line indicates the allowable T prefixes and YY suffixes for a particular root name XX.

Table 4-4: LINPACK Naming Convention — Subprogram Names

Valid T		XX	Valid YY				
S	C	GE	CO	FA	SL	DI	
S	C	GB	CO	FA	SL	DI	
S	C	PO	CO	FA	SL	DI	
S	C	PB	CO	FA	SL	DI	
S	C	PP	CO	FA	SL	DI	
S	C	SI	CO	FA	SL	DI	
S	C	SP	CO	FA	SL	DI	
	C	HI	CO	FA	SL	DI	
	C	HP	CO	FA	SL	DI	
S	C	TR	CO		SL	DI	
S	C	GT			SL		
S	C	PT			SL		
S	C	CH	DC		UD	DD	EX
S	C	QR	DC	SL			
S	C	SV	DC				

LINPACK is organized so that it is usually necessary to call two subprograms to perform the above operations. One subprogram is called to process the matrix and another is called to process a particular right-hand side. This division of labor significantly reduces computer time when there is a sequence of problems involving the same matrix but different right-hand sides. It also allows you the flexibility to choose between subprograms that are fast but use a less reliable, elementary test for singularity and subprograms that are slightly slower but use a significantly more reliable test involving an estimate of the condition number of the coefficient matrix.

Condition Number

The condition number, $\kappa(A)$, of the coefficient matrix A measures the sensitivity of the solution x of the system of linear equations $Ax = b$ to errors in the matrix A and the right-hand side b. If δA and δb represent the errors in A and b, respectively, and if $\|\ \|$ represents any vector norm and its subordinate matrix norm, the error δx in x that results from solving $(A+\delta A)(x+\delta x) = b+\delta b$ instead of $Ax = b$ is bounded by

$$\frac{\|\delta x\|}{\|x\|} \leq \frac{\kappa(A)}{1 - \|A^{-1}\|\ \|\delta A\|} \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|b\|} \right).$$

A standard result of numerical analysis shows that the roundoff error introduced by the solution process may be modeled by taking $\|\delta A\|/\|A\|$ and $\|\delta b\|/\|b\|$ to be small multiples of the computer's machine epsilon. Computational singularity of A results in $\kappa(A) = \infty$. A more common situation occurs when A is not numerically singular but is ill-conditioned. When a matrix is ill-conditioned, $\kappa(A)$ is large, so small errors in the matrix and right-hand side and small roundoff errors introduced during the solution process itself are magnified greatly in the solution.

Since $1 < \kappa(A) \leq \infty$, it is more convenient to compute the reciprocal condition number, $1/\kappa(A)$, than $\kappa(A)$ itself. The reciprocal condition number has the interpretation that if $1/\kappa(A)$ approximately equals 10^{-d}, elements of x can be expected to have d fewer significant digits of accuracy than the elements of A or b. Consequently, if errors in the coefficient matrix and right-hand side exceed $1/\kappa(A)$, or if $1/\kappa(A)$ is negligible compared to 1.0, then x may have no significant digits at all.

Determinant and Inverse

Subprograms for computing the determinant and inverse of a matrix are provided, although it is almost never necessary to compute either one. While papers and reference books extensively use the notation "$\det(A) \neq 0$" to mean "A is nonsingular," SCILIB includes both more efficient and more reliable subprograms for detecting singularity. Similarly, references frequently use "$A^{-1}b$" to mean "the solution x of the system of linear equations $Ax = b$." Again, it is more efficient and accurate to compute the solution directly than to invert the coefficient matrix and multiply the inverse times the right-hand-side vector. This is true even if there are many systems of equations, all using the same coefficient matrix; the matrix may be factored once and the systems may be solved from the factors just as efficiently—and more accurately—than by matrix multiplication by the inverse.

Supplemental Reading

Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart. *LINPACK Users' Guide.* Philadelphia, PA: SIAM Publications. 1979.

Forsythe, G., and C.B. Moler. *Computer Solution of Linear Algebraic Systems.* Englewood Cliffs, NJ: Prentice-Hall, Inc. 1967.

Subprogram Descriptions

MINV — Invert Matrix or Solve Linear Equations

Purpose

Given a general dense n-by-n matrix A, this subprogram evaluates the determinant of A, optionally solves one or more systems of linear equations $Ax_j=b_j$, and optionally computes A^{-1}.

Computational singularity of A results in $\det(A)=0$. The partial product of pivot elements is computed as A is factored, and A is declared singular if the absolute value of the partial product ever fails to exceed a user-supplied tolerance. If this condition is detected during factorization, the computation is terminated and the "small" partial determinant is returned to indicate its occurrence. A more common situation, however, is that A is not numerically singular but happens to be ill-conditioned. When a matrix is ill-conditioned, small errors in the matrix and right-hand side and small roundoff errors introduced during the solution process itself are magnified greatly in the solution.

Although singular matrices are characterized by having zero determinants, a small nonzero determinant is unrelated to computational singularity or ill-conditioning. Therefore, the stopping criteria is artificial, and this subprogram may give a completely unreliable indication of the singularity of A. The SCILIB subprograms SGECO, SGEDI, and SGESL may be combined to perform the same functions as MINV, while providing a more reliable indication of singularity.

Usage

SCILIB:

```
INTEGER*8 n, ldab, m, job
REAL*8    ab(ldab, n+m), work(2*n), det, tol
CALL MINV (ab, n, ldab, work, det, tol, m, job)
```

Input

ab — Array containing the **n**-by-**n** matrix A in columns 1 through **n**, and $\mathbf{m} \geq 0$ right-hand-side vectors b_j in columns **n**+1 through **n**+**m**.

n — The order of matrix A, $\mathbf{n} > 0$.

ldab — The leading dimension of array **ab** as declared in the calling program unit, with $\mathbf{ldab} \geq \mathbf{n}$.

tol — Lower limit for the partial product of pivot elements, $\mathbf{tol} > 0$. A is considered singular if the magnitude of the partial product of pivot elements ever fails to exceed **tol**.

m — Number of right-hand-side vectors, $\mathbf{m} \geq 0$. If $\mathbf{m} = 0$, no right-hand sides are solved.

job — Option flag:

$\mathbf{job} = 0$	do not compute A^{-1}
$\mathbf{job} \neq 0$	compute A^{-1}

Working Storage

work — An array of size 2**n**, used for work space.

Output

ab If $|\mathbf{det}| > \mathbf{tol}$ on return and **job** $\neq 0$, then A^{-1} overwrites A in columns 1 to **n** of **ab**. If $|\mathbf{det}| \leq \mathbf{tol}$ on return or **job** $= 0$, then columns 1 to **n** of **ab** may have been destroyed.

If $|\mathbf{det}| > \mathbf{tol}$ on return and **m** $\neq 0$, then solution vectors x_j of the systems of linear equations $Ax_j = b_j$ overwrite the right-hand-side vectors in columns **n**+1 to **n**+**m** of **ab**. If $|\mathbf{det}| \leq \mathbf{tol}$ on return or **m** $= 0$, then no solution vectors have been computed.

det The determinant of A if $|\mathbf{det}| > \mathbf{tol}$. Otherwise, **det** is the last partial product computed before the matrix was declared singular and the computation was terminated. Caution: an overflow may be produced in the computation of **det**.

Notes

Cray Research, Inc. has declared this subprogram obsolete in release 6.0 of the UNICOS Math and Scientific Library.

If an error in the arguments is detected, the subprograms call error handler XERSCI, which writes an error message onto the standard error file and terminates execution. The standard version of XERSCI (refer to Chapter 9, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are:

n < 0,
m < 0,
ldab $<$ **n**, and
tol ≤ 0.

It is almost never necessary to compute either the determinant or the inverse of a matrix. While papers and reference books extensively use the notation "det(A) $\neq$ 0" to mean "A is nonsingular," SCILIB includes both more efficient and more reliable subprograms for detecting singularity. Similarly, references frequently use "$A^{-1}b$" to mean "the solution x of the system of linear equations $Ax = b$." Again, it is more efficient and accurate to compute the solution directly than to invert the coefficient matrix and multiply the inverse times the right-hand-side vector. This is true even if there are many systems of equations, all using the same coefficient matrix; the matrix may be factored once, and the systems may be solved from the factors as efficiently, but more accurately, as by matrix multiplication by the inverse.

Example

Invert the 6-by-6 REAL*8 matrix A stored in array AB whose dimensions are 10 by 50 and solve 25 systems of linear equations whose right-hand-sides b_j are stored in columns 7 to 31 of AB. Consider A to be singular if $|\det(A)| < 10^{-10}$.

```
INTEGER*8 N,LDAB,M,JOB
REAL*8    AB(10,50),WORK(20),DET,TOL
N = 6
LDAB = 10
M = 25
TOL = 1.0E-10
JOB = 1
CALL MINV (AB,N,LDAB,WORK,DET,TOL,M,JOB)
IF ( ABS(DET) .LE. TOL ) THEN
  handle singular matrix
END IF
```

OPFILT — Solve Symmetric Toeplitz Linear Equations

Purpose

Given a real symmetric n-by-n Toeplitz matrix A and a right-hand-side vector b, this subprogram solves the system of linear equations $Ax=b$ by means of the Weiner-Levinson algorithm. A matrix A is a symmetric Toeplitz matrix if its elements a_{ij} are given by $a_{ij} = q_{|i-j|+1}$. The following illustrates a 3-by-3 symmetric Toeplitz matrix.

$$\begin{matrix} q_1 & q_2 & q_3 \\ q_2 & q_1 & q_2 \\ q_3 & q_2 & q_1 \end{matrix}$$

OPFILT is designed for use only on matrices which do not require pivoting to maintain numerical stability.

Usage

SCILIB:

```
INTEGER*8 n
REAL*8    x(n), b(n), work(2*n), q(n)
CALL OPFILT (n, x, b, work, q)
```

Input

n The order of matrix A, $n \geq 0$.

b The right-hand-side vector b.

q The vector that generates the A matrix via the relationship $a_{ij} = q_{|i-j|+1}$.

Working Storage

work An array of size **2n**, used for work space.

Output

x The solution vector x.

Notes

If an error in the arguments is detected, the subprograms call error handler XERSCI, which writes an error message onto the standard error file and terminates execution. The standard version of XERSCI (refer to Chapter 9, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are:

$$n < 0.$$

An overflow or divide by zero may be produced if the matrix is not suitable for solution with the Weiner-Levinson algorithm.

Example

Solve the 6-by-6 REAL*8 symmetric Toeplitz matrix system $Ax = b$ where A is represented by array Q of dimension 10, and x and b are stored in arrays X and B, also of dimension 10 by 10, respectively.

```
INTEGER*8 N
REAL*8    X(10),B(10),WORK(20),Q(10)
N = 6
CALL OPFILT (N,X,B,WORK,Q)
```

Purpose

These subprograms compute the triangular factorization and estimate the condition number of a general nonsymmetric n-by-n band matrix A stored in a two-dimensional array. 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 subprograms for band matrices use less storage than the subprograms for full matrices if $2kl + ku < n$.

Tridiagonal matrices are the special case $kl = ku = 1$. They can be handled more efficiently by the subprograms SGTSL and CGTSL. SCILIB also contains subprograms designed to handle positive definite band matrices. These subprograms are documented elsewhere in this chapter.

Specifically, given A, these subprograms determine an upper-triangular band matrix U, and a matrix L that is the product of elementary lower-triangular band matrices and permutation matrices such that

$$A = LU$$

and compute an estimate of $\kappa(A)$, the condition number of A. Refer to "Condition Number" in the introduction to this chapter for a discussion of $\kappa(A)$. When a matrix is ill-conditioned, $\kappa(A)$ is large, so small errors in the matrix and right-hand side and small roundoff errors introduced during the solution process itself are magnified greatly in the solution.

Since $1 < \kappa(A) \leq \infty$, these subprograms actually compute the reciprocal condition number, $1/\kappa(A)$. The reciprocal condition number has the interpretation that if $1/\kappa(A)$ approximately equals 10^{-d}, elements of x can be expected to have d fewer significant digits of accuracy than the elements of A or b. Consequently, if errors in the coefficient matrix and right-hand side exceed $1/\kappa(A)$, or if $1/\kappa(A)$ is negligible compared to 1.0, then x may have no significant digits at all.

A set of companion subprograms computes the triangular factorization of a general band matrix without estimating its condition number. These companion subprograms are faster but provide a less reliable indication of singularity.

The triangular factors may be used to solve a system of linear equations, $Ax = b$, by successively solving $L(Ux) = b$. The determinant of A can be computed as $\det(A) = \det(L)\det(U)$. These operations are performed by a set of companion SCILIB subprograms whose names depend on the data type:

Data Type	Estimate Condition	Factor	Solve	Determinant
REAL*8	SGBCO	SGBFA	SGBSL	SGBDI
COMPLEX*16	CGBCO	CGBFA	CGBSL	CGBDI

The inverse of A will usually be a full n-by-n matrix that cannot be stored in the band storage of A. Therefore, no direct provision is made for computing A^{-1}. Calculations formulated in terms of matrix inverses are invariably more efficient when expressed in terms of the solution of sets of linear equations.

Matrix Storage

Because it is not necessary to store or operate on the zeros outside the band of A, you need only provide the elements within the band of A. Compared to storing the entire matrix, this can save memory if $2kl + ku + 1 < n$.

The following example illustrates the storage of general band matrices. Consider the following matrix A of order $n = 9$ and lower and upper bandwidths $kl = 2$ and $ku = 3$, respectively:

11	12	13	14	0	0	0	0	0
21	22	23	24	25	0	0	0	0
31	32	33	34	35	36	0	0	0
0	42	43	44	45	46	47	0	0
0	0	53	54	55	56	57	58	0
0	0	0	64	65	66	67	68	69
0	0	0	0	75	76	77	78	79
0	0	0	0	0	86	87	88	89
0	0	0	0	0	0	97	98	99

When Gaussian elimination is performed on a general band matrix, pivoting introduces nonzero elements outside the band. L can be stored with a lower bandwidth of kl, but U requires an upper bandwidth of $kl + ku$. You must, therefore, provide storage for the extra kl diagonals. This is done by presenting the original matrix to the subprogram in an array large enough to satisfy the additional storage requirements. Thus, for the above matrix, A is given in an array **ab** with at least $2kl + ku + 1 = 8$ rows and $n = 9$ columns as follows:

*	*	*	*	*	+	+	+	+
*	*	*	*	+	+	+	+	+
*	*	*	14	25	36	47	58	69
*	*	13	24	35	46	57	68	79
*	12	23	34	45	56	67	78	89
11	22	33	44	55	66	77	88	99
21	32	43	54	65	76	87	98	*
31	42	53	64	75	86	97	*	*

The asterisks in the $(kl + ku)$-by-$(kl + ku)$ triangle at the upper left corner and in the ku-by-ku triangle at the lower right corner represent elements of **ab** that are not referenced, and the plus signs in the first kl rows indicate elements that may be filled in during the factorization. Thus, if a_{ij} is an element within the band of A, then it is stored in $\mathbf{ab}(kl + ku + 1 + i - j, j)$. 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 $kl + ku + 1$ of **ab**.

Usage

SCILIB:

```
INTEGER*8 ldab, n, kl, ku, ipvt(n)
REAL*8    ab(ldab, n), rcond, work(n)
CALL SGBCO (ab, ldab, n, kl, ku, ipvt, rcond, work)

INTEGER*8    ldab, n, kl, ku, ipvt(n)
COMPLEX*16 ab(ldab, n), work(n)
REAL*8       rcond
CALL CGBCO (ab, ldab, n, kl, ku, ipvt, rcond, work)
```

Input

ab Array containing the **n**-by-**n** band matrix A in the compressed form described above. If a_{ij} is in the band, it is stored in **ab**(**kl**+**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 **kl**+1 through **2kl**+**ku**+1. The first **kl** rows are used for work space and output.

ldab The leading dimension of array **ab** as declared in the calling program unit, with **ldab** ≥ **2kl**+**ku**+1.

n The order of matrix A, **n** > 0.

kl The lower bandwidth of A, i.e., the number of nonzero diagonals below the principal diagonal in the band, 0 ≤ **kl** < **n**.

ku The upper bandwidth of A, i.e., the number of nonzero diagonals above the principal diagonal in the band, 0 ≤ **ku** < **n**. These subprograms are more efficient if **kl** ≤ **ku**. This usually can be arranged since the factors used by these subprograms can be used to solve either $Ax = b$ or $A^*x = b$.

Working Storage

work An array of size **n**, used for work space.

Output

ab The triangular factors replace the input matrix. **ab** must be preserved between the condition number estimation call and any solve or determinant call.

ipvt The pivot information necessary to construct the permutations in the lower-triangular factor, L. **ipvt** must be preserved between the condition number estimation call and any solve or determinant call.

rcond An estimate of the reciprocal condition number, $1/\kappa(A)$. If **rcond** is small enough so that the logical expression

1.0 + rcond .EQ. 1.0

is true, then A can be regarded as singular to working precision.

Notes

These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

Example

Factor and estimate the reciprocal condition number of the 9-by-9 REAL*8 general band matrix A whose lower bandwidth is 2 and whose upper bandwidth is 3. A is stored as illustrated above in array AB whose dimensions are 8 by 10.

```
INTEGER*8 LDAB,N,KL,KU,IPVT(10)
REAL*8    AB(8,10),RCOND,WORK(10)
LDAB = 8
N = 9
KL = 2
KU = 3
CALL SGBCO (AB,LDAB,N,KL,KU,IPVT,RCOND,WORK)
IF ( 1.0 + RCOND .EQ. 1.0 ) THEN
  handle singular matrix
END IF
```

SGBDI/CGBDI — Determinant

Purpose

Given the triangular factorization of a general n-by-n band matrix A, these subprograms evaluate the determinant of A. No provision is made to compute A^{-1} since it will usually be a full n-by-n matrix that cannot be stored in the band storage of A. Moreover, it is almost never necessary to compute the inverse of a matrix. Mathematical references frequently use "$A^{-1}b$" to mean "the solution x of the system of linear equations $Ax = b$." It is more efficient and accurate to compute the solution directly than to invert the coefficient matrix and multiply the inverse times the right-hand-side vector. This is true even if there are many systems of equations, all using the same coefficient matrix; the matrix may be factored once and the systems may be solved from the factors just as efficiently, and more accurately, than by matrix multiplication by the inverse.

Specifically, given an n-by-n upper-triangular band matrix U, and a matrix L which is the product of elementary lower-triangular band matrices and permutation matrices such that

$$A = LU,$$

the subprograms compute

$$\det(A) = \det(L)\det(U).$$

The triangular factorization of the coefficient matrix may be computed by either of two companion subprograms. One computes only the factorization, using an elementary test for singularity of the coefficient matrix; it is slightly faster. The other not only computes the factorization, but also estimates the condition number of the matrix. This takes a little more time, but is considerably more reliable. The names of the companion subprograms depend on the data type:

Data Type	Estimate Condition	Factor	Determinant
REAL*8	SGBCO	SGBFA	SGBDI
COMPLEX*16	CGBCO	CGBFA	CGBDI

The companion subprograms are documented elsewhere in this chapter.

Usage

SCILIB:

```
INTEGER*8 ldab, n, kl, ku, ipvt(n)
REAL*8    ab(ldab, n), det(2)
CALL SGBDI (ab, ldab, n, kl, ku, ipvt, det)

INTEGER*8    ldab, n, kl, ku, ipvt(n)
COMPLEX*16 ab(ldab, n), det(2)
CALL CGBDI (ab, ldab, n, kl, ku, ipvt, det)
```

Input

ab Array containing the triangular factors of the **n**-by-**n** general band matrix A as computed by the companion factorization or condition number estimation subprogram. **ab** must have been preserved between the factorization or condition number call and the determinant call.

ldab The leading dimension of array **ab** as declared in the calling program unit, with **ldab** ≥ **2kl+ku+1**.

n The order of matrix A, **n** ≥ **0**.

kl The lower bandwidth of A, i.e., the number of nonzero diagonals below the principal diagonal in the band, $0 \leq$ **kl** $<$ **n**.

ku The upper bandwidth of A, i.e., the number of nonzero diagonals above the principal diagonal in the band, $0 \leq$ **ku** $<$ **n**.

ipvt The pivot information necessary to construct the permutations in the lower-triangular factor, L. **ipvt** must have been preserved between the factorization or condition number call and the determinant call.

Output

det The determinant of A, in the form $\det(A) =$ **det(1)** $\times 10^{\text{det(2)}}$. This expression may underflow or overflow if evaluated; on the CONVEX supercomputer, underflows automatically flush to zero, but overflows normally terminate execution. For REAL*8 and COMPLEX*16, overflow cannot occur if **det(2)** ≤ 306. If evaluation is safe, an efficient way to do it is with the statement

$$\det(A) = \text{det(1) * 10.0 ** INT(det(2))}$$

The value stored in **det(2)** is an integer in REAL or COMPLEX form. **det(1)** is normalized so that either **det(1)** $= 0$ or $1 \leq |Re(\text{det(1)}| + |Im(\text{det(1)}| < 10$, where $Re(z)$ and $Im(z)$ are the real and imaginary parts of z; $Re(z) = z$ and $Im(z) = 0$ if z is real.

Notes

These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

It is almost never necessary to compute the determinant of a matrix. While it is true that papers and reference books make extensive use of the notation "$\det(A) \neq 0$" to mean "A is nonsingular," SCILIB includes more efficient and more reliable subprograms for detecting singularity.

Example

Compute the determinant of a 9-by-9 REAL*8 general band matrix A whose lower bandwidth is 2 and whose upper bandwidth is 3. A is stored in array AB whose dimensions are 8 by 10. The less reliable, but slightly faster, factorization subprogram is used to factor the coefficient matrix.

```
INTEGER*8 LDAB,N,KL,KU,IPVT(10),IER
REAL*8    AB(8,10),DET(2),DETA
LDAB = 8
N = 9
KL = 2
KU = 3
CALL SGBFA (AB,LDAB,N,KL,KU,IPVT,IER)
IF ( IER .EQ. 0 ) THEN
  CALL SGBDI (AB,LDAB,N,KL,KU,IPVT,DET)
  IF ( DET(1) .EQ. 0.0 ) THEN
    DETA = 0.0
  ELSE IF ( DET(2) .LE. 306 ) THEN
    DETA = DET(1) * 10.0 ** INT(DET(2))
  ELSE
    the determinant of A is too large to evaluate
    without overflow
  END IF
ELSE
  DETA = 0.0
END IF
```

SGBFA/CGBFA — Band LU Factorization

Purpose

These subprograms compute the triangular factorization of a general nonsymmetric n-by-n band matrix A stored in a two-dimensional array. 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 $m = kl + ku + 1$ is the total bandwidth. The subprograms for band matrices use less storage than the subprograms for full matrices if $2kl + ku < n$.

Tridiagonal matrices are the special case $kl = ku = 1$. They can be handled more efficiently by the subprograms SGTSL or CGTSL. SCILIB also contains subprograms designed to handle positive definite band matrices. These subprograms are documented elsewhere in this chapter.

Specifically, given A, these subprograms determine an upper-triangular band matrix U, and a matrix L that is the product of elementary lower triangular band matrices and permutation matrices such that

$$A = LU.$$

Computational singularity of A results in one or more zero diagonal elements of U. This condition is detected during factorization, and a status response is returned to indicate its occurrence. A more common situation, however, is that A is not numerically singular but happens to be ill-conditioned. When a matrix is ill-conditioned, small errors in the matrix and right-hand side and small roundoff errors introduced during the solution process itself are magnified greatly in the solution. A set of companion subprograms computes the triangular factorization of a general band matrix and also estimates its condition number. These companion subprograms provide a more reliable indication of singularity. The small amount of additional time they require is usually worthwhile, especially when developing a program or encountering stability or convergence problems.

The triangular factors may be used to solve a system of linear equations, $Ax = b$, by successively solving $L(Ux) = b$. The determinant of A can be computed as $\det(A) = \det(L)\det(U)$. These operations are performed by a set of companion SCILIB subprograms whose names depend on the data type:

Data Type	Factor	Estimate Condition	Solve	Determinant
REAL*8	SGBFA	SGBCO	SGBSL	SGBDI
COMPLEX*16	CGBFA	CGBCO	CGBSL	CGBDI

The companion subprograms are documented elsewhere in this chapter.

The inverse of A will usually be a full n-by-n matrix that cannot be stored in the band storage of A. Therefore, no direct provision is made for computing A^{-1}. Calculations formulated in terms of matrix inverses are invariably more efficient when expressed in terms of the solution of sets of linear equations.

Matrix Storage

Because it is not necessary to store or operate on the zeros outside the band of A, you need only provide the elements within the band of A. Compared to storing the entire matrix, this can save memory if $2kl + ku + 1 < n$.

The following example illustrates the storage of general band matrices. Consider the following matrix A of order $n = 9$ and lower and upper bandwidths $kl = 2$ and $ku = 3$, respectively:

11	12	13	14	0	0	0	0	0
21	22	23	24	25	0	0	0	0
31	32	33	34	35	36	0	0	0
0	42	43	44	45	46	47	0	0
0	0	53	54	55	56	57	58	0
0	0	0	64	65	66	67	68	69
0	0	0	0	75	76	77	78	79
0	0	0	0	0	86	87	88	89
0	0	0	0	0	0	97	98	99

When Gaussian elimination is performed on a general band matrix, pivoting introduces nonzero elements outside the band. L can be stored with a lower bandwidth of kl, but U requires an upper bandwidth of $kl + ku$. You must, therefore, provide storage for the extra kl diagonals. This is done by presenting the original matrix to the subprogram in an array large enough to satisfy the additional storage requirements. Thus, for the above matrix, A is given in an array **ab** with at least $2kl + ku + 1 = 8$ rows and $n = 9$ columns as follows:

*	*	*	*	*	+	+	+	+
*	*	*	*	+	+	+	+	+
*	*	*	14	25	36	47	58	69
*	*	13	24	35	46	57	68	79
*	12	23	34	45	56	67	78	89
11	22	33	44	55	66	77	88	99
21	32	43	54	65	76	87	98	*
31	42	53	64	75	86	97	*	*

The asterisks in the $(kl + ku)$-by-$(kl + ku)$ triangle at the upper left corner and in the ku-by-ku triangle at the lower right corner represent elements of **ab** that are not referenced, and the plus signs in the first kl rows indicate elements that may be filled in during the factorization. Thus, if a_{ij} is an element within the band of A, then it is stored in $\mathbf{ab}(kl + ku + 1 + i - j, j)$. 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 $kl + ku + 1$ of **ab**.

Usage

SCILIB:

```
INTEGER*8 ldab, n, kl, ku, ipvt(n), ier
REAL*8    ab(ldab, n)
CALL SGBFA (ab, ldab, n, kl, ku, ipvt, ier)

INTEGER*8    ldab, n, kl, ku, ipvt(n), ier
COMPLEX*16 ab(ldab, n)
CALL CGBFA (ab, ldab, n, kl, ku, ipvt, ier)
```

SGBFA/CGBFA — Continued

Input

ab Array containing the **n**-by-**n** band matrix A in the compressed form described above. If a_{ij} is in the band, it is stored in **ab**(**kl**+**ku**+1+i-j,j). Columns of A are stored in the columns of **ab**, and the diagonals of A are stored in rows **kl**+1 through **2kl**+**ku**+1. The first **kl** rows are used for work space and output.

ldab The leading dimension of array **ab** as declared in the calling program unit, with **ldab** ≥ **2kl**+**ku**+1.

n The order of matrix A, **n** > **0**.

kl The lower bandwidth of A, i.e., the number of nonzero diagonals below the principal diagonal in the band, **0** ≤ **kl** < **n**.

ku The upper bandwidth of A, i.e., the number of nonzero diagonals above the principal diagonal in the band, **0** ≤ **ku** < **n**. These subprograms are more efficient if **kl** ≤ **ku**. This usually can be arranged because factors used by these subprograms can be used to solve either $Ax = b$ or $A^*x = b$.

Output

ab The triangular factors replace the input matrix. **ab** must be preserved between the factorization call and any solve or determinant call.

ipvt The pivot information necessary to construct the permutations in the lower triangular factor, L. **ipvt** must be preserved between the factorization call and any solve or determinant call.

ier Status response:

ier = 0	Normal return.
ier = k ≠ 0	if u_{kk}=0. (u_{kk} is the k-th element on the diagonal of upper triangular matrix U). Technically, this is not an error condition for these subprograms, but it does indicate that A is computationally singular and that a division by zero will occur if the factorization is used to solve a system of linear equations.

Notes

These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

Example

Factor the 9-by-9 REAL*8 general band matrix A whose lower bandwidth is 2 and whose upper bandwidth is 3. A is stored, as illustrated above, in array AB whose dimensions are 8 by 10.

```
INTEGER*8 LDAB,N,KL,KU,IPVT(10),IER
REAL*8    AB(8,10)
LDAB = 8
N = 9
KL = 2
KU = 3
CALL SGBFA (AB,LDAB,N,KL,KU,IPVT,IER)
IF ( IER .NE. 0 ) THEN
  handle singular matrix
END IF
```

Solve Band Linear Equations — SGBSL/CGBSL

Purpose

Given the triangular factorization of a general n-by-n band matrix A, and a right-hand-side n-vector b, these subprograms solve the system of linear equations $Ax = b$. Optionally, these subprograms will solve the system $A^*x = b$, where A^* is the conjugate transpose of A (the conjugate transpose of a real matrix is simply the transpose). Specifically, given an n-by-n upper-triangular band matrix U, and a matrix L that is the product of elementary lower-triangular band matrices and permutation matrices such that

$$A = LU,$$

and an n-vector b, to find x satisfying $Ax = b$, the subprograms successively solve

$$Lw = b$$

and

$$Ux = w,$$

while to solve $A^*x = b$, the subprograms successively solve

$$U^*v = b$$

and

$$L^*x = v.$$

Triangular factors of the coefficient matrix may be computed by either of two companion subprograms. One computes only the factorization, using an elementary test for singularity of the coefficient matrix; it is slightly the faster. The other not only computes the factorization, but also estimates the condition number of the matrix. This takes a little more time, but is considerably more reliable. Names of companion subprograms depend on the data type:

Data Type	Estimate Condition	Factor	Solve
REAL*8	SGBCO	SGBFA	SGBSL
COMPLEX*16	CGBCO	CGBFA	CGBSL

The companion subprograms are documented elsewhere in this chapter.

Usage

SCILIB:

```
INTEGER*8 ldab, n, kl, ku, ipvt(n), job
REAL*8    ab(ldab, n), b(n)
CALL SGBSL (ab, ldab, n, kl, ku, ipvt, b, job)

INTEGER*8    ldab, n, kl, ku, ipvt(n), job
COMPLEX*16 ab(ldab, n), b(n)
CALL CGBSL (ab, ldab, n, kl, ku, ipvt, b, job)
```

Input

ab Array containing the triangular factors of the **n**-by-**n** general band matrix A as computed by the companion factorization or condition number estimation subprogram. **ab** must have been preserved between the factorization or condition number call and the solve call.

ldab The leading dimension of array **ab** as declared in the calling program unit, with **ldab** ≥ **2kl+ku+1**.

n The order of matrix A, **n** ≥ **0**.

kl The lower bandwidth of A, i.e., the number of nonzero diagonals below the principal diagonal in the band, $0 \leq$ **kl** $<$ **n**.

ku The upper bandwidth of A, i.e., the number of nonzero diagonals above the principal diagonal in the band, $0 \leq$ **ku** $<$ **n**.

ipvt Pivot information necessary to construct permutations in the lower-triangular factor, L. **ipvt** must have been preserved between the factorization or condition number estimation call and the solve call.

b The right-hand-side vector b.

job Option flag:

job $= 0$ solve $Ax = b$
job $\neq 0$ solve $A^*x = b$

Output

b The solution vector x overwrites the right-hand-side vector b.

Notes

These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

Example

Solve a system of linear equations $Ax = b$, where A is a 9-by-9 REAL*8 general band matrix whose lower bandwidth is 2 and whose upper bandwidth is 3. A is stored in array AB whose dimensions are 8 by 10. b is a vector 9 elements long stored in an array B of dimension 10. The more robust, but slightly slower, condition number estimation subprogram is used to factor the coefficient matrix.

```
INTEGER*8 LDAB,N,KL,KU,IPVT(10),JOB
REAL*8    AB(8,10),B(10),RCOND,WORK(10)
LDAB = 8
N = 9
KL = 2
KU = 3
JOB = 0
CALL SGBCO (AB,LDAB,N,KL,KU,IPVT,RCOND,WORK)
IF ( 1.0 + RCOND .NE. 1.0 ) THEN
  CALL SGBSL (AB,LDAB,N,KL,KU,IPVT,B,JOB)
ELSE
  handle singular matrix
END IF
```

Estimate Condition — SGECO/CGECO

Purpose

These subprograms compute the triangular factorization and estimate the condition number of a general dense n-by-n matrix A. Specifically, given A, these subprograms determine an n-by-n permutation matrix P, an n-by-n unit lower-triangular matrix L, and an n-by-n upper-triangular matrix U, such that

$$PA = LU$$

and compute an estimate of $\kappa(A)$, the condition number of A. Refer to "Condition Number" in the introduction to this chapter for a discussion of $\kappa(A)$. When a matrix is ill-conditioned, $\kappa(A)$ is large, so small errors in the matrix and right-hand side and small roundoff errors introduced during the solution process itself are magnified greatly in the solution.

Since $1 < \kappa(A) \leq \infty$, these subprograms actually compute the reciprocal condition number, $1/\kappa(A)$. The reciprocal condition number has the interpretation that if $1/\kappa(A)$ approximately equals 10^{-d}, elements of x can be expected to have d fewer significant digits of accuracy than the elements of A or b. Consequently, if errors in the coefficient matrix and right-hand side exceed $1/\kappa(A)$, or if $1/\kappa(A)$ is negligible compared to 1.0, then x may have no significant digits at all.

A set of companion subprograms computes the triangular factorization of a matrix without estimating its condition number. These companion subprograms are faster but provide a less reliable indication of singularity.

The triangular factors may be used to solve a system of linear equations, $Ax = b$, by successively solving $L(Ux) = Pb$. The determinant of A can be computed as $\det(A) = \det(P) \times \det(L) \times \det(U)$. The inverse of A may be formed as $A^{-1} = U^{-1}L^{-1}P$. These operations are performed by a set of companion SCILIB subprograms whose names depend on the data type:

Data Type	Estimate Condition	Factor	Solve	Determinant or inverse
REAL*8	SGECO	SGEFA	SGESL	SGEDI
COMPLEX*16	CGECO	CGEFA	CGESL	CGEDI

The companion subprograms are documented elsewhere in this chapter.

Usage

SCILIB:

```
INTEGER*8 lda, n, ipvt(n)
REAL*8    a(lda, n), rcond, work(n)
CALL SGECO (a, lda, n, ipvt, rcond, work)

INTEGER*8    lda, n, ipvt(n)
COMPLEX*16 a(lda, n), work(n)
REAL*8       rcond
CALL CGECO (a, lda, n, ipvt, rcond, work)
```

Input

a Array containing the **n**-by-**n** matrix A.

lda The leading dimension of array **a** as declared in the calling program unit, with **lda** $\geq$ max(**n**,1).

n The order of matrix A, **n** $\geq$ 0.

Working storage

work An array of size **n**, used for work space.

Output

a The triangular factors replace the input matrix: the strict lower triangle of **a** contains the strict lower triangle of L and the upper triangle of **a** contains U. **a** must be preserved between the condition number estimation call and any solve, determinant, or inverse call.

ipvt The pivot information necessary to construct the permutation matrix P. **ipvt** must be preserved between the condition number estimation call and any solve, determinant, or inverse call.

rcond An estimate of the reciprocal condition, $1/\kappa(A)$. If **rcond** is small enough so that the logical expression

1.0 + rcond .EQ. 1.0

is true, then A can be regarded as singular to working precision. If **rcond** is zero, then the companion subprograms for solving and computing the inverse may divide by zero.

Notes

These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

The triangular factors are stored in a different format from the format used by the standard LINPACK subprograms, but are compatible with the SCILIB factorization, solve, and determinant and inverse subprograms.

Example

Factor the 6-by-6 REAL*8 matrix A stored in array **A** whose dimensions are 10 by 10 and estimate its reciprocal condition number.

```
INTEGER*8 LDA,N,IPVT(10)
REAL*8    A(10,10),RCOND,WORK(10)
LDA = 10
N = 6
CALL SGECO (A,LDA,N,IPVT,RCOND,WORK)
IF ( 1.0 + RCOND .EQ. 1.0 ) THEN
  handle singular matrix
END IF
```

Purpose

Given the triangular factorization of a general dense n-by-n coefficient matrix A, these subprograms evaluate the determinant of A and/or compute A^{-1}. Specifically, given an n-by-n permutation matrix P, an n-by-n unit lower-triangular matrix L, and a nonsingular n-by-n upper-triangular matrix U, such that

$$PA = LU,$$

the subprograms compute

$$\det(A) = \det(P) \times \det(L) \times \det(U)$$

and/or

$$A^{-1} = U^{-1}L^{-1}P.$$

The triangular factors of the coefficient matrix may be computed by either of two companion subprograms. One computes only the factorization, using an elementary test for singularity of the coefficient matrix; it is slightly faster. The other not only computes the factorization, but also estimates the condition number of the matrix. This process takes a little more time, but is considerably more reliable, especially when A^{-1} is desired. The names of the companion subprograms depend on the data type:

Data Type	Estimate Condition	Factor	Determinant or inverse
REAL*8	SGECO	SGEFA	SGEDI
COMPLEX*16	CGECO	CGEFA	CGEDI

The companion subprograms are documented elsewhere in this chapter.

Usage

SCILIB:

```
INTEGER*8 lda, n, ipvt(n), job
REAL*8    a(lda, n), det(2), work(n)
CALL SGEDI (a, lda, n, ipvt, det, work, job)

INTEGER*8    lda, n, ipvt(n), job
COMPLEX*16 a(lda, n), det(2), work(n)
CALL CGEDI (a, lda, n, ipvt, det, work, job)
```

Input

a Array containing the triangular factors L and U of the **n**-by-**n** coefficient matrix A as computed by the companion factorization or condition number estimation subprogram. **a** must have been preserved between the factorization or condition number call and the determinant or inverse call.

lda The leading dimension of array **a** as declared in the calling program unit, with **lda** ≥ max(**n**,1).

n The order of matrix A, **n** ≥ 0.

ipvt The pivot information necessary to construct the permutation matrix P as computed by the companion factorization or condition number estimation subprogram. **ipvt** must have been preserved between the factorization or condition number call and the determinant or inverse call.

job Option flag:

job = 1 compute only A^{-1}
job = 10 compute only det(A)
job = 11 compute both A^{-1} and det(A)

Working storage

work An array of size **n**, used for work space if A^{-1} is requested.

Output

a Unchanged if A^{-1} is not requested. Otherwise, A^{-1} overwrites the triangular factors of the coefficient matrix.

det Not referenced if the determinant is not requested. Otherwise, the determinant of A, in the form $\det(A) = \mathbf{det(1)} \times 10^{\mathbf{det(2)}}$. This expression may underflow or overflow if evaluated; on the CONVEX supercomputer, underflows automatically flush to zero, but overflows normally terminate execution. For REAL*8 and COMPLEX*16, overflow cannot occur if **det(2)** ≤ 306. If evaluation is safe, an efficient way to do it is with the statement

det(A) = **det(1) * 10.0 ** INT(det(2))**

Refer to "Example 2."

The value stored in **det(2)** is an integer in REAL or COMPLEX form. **det(1)** is normalized so that either **det(1)** = 0 or $1 \le |Re(\mathbf{det(1)})| + |Im(\mathbf{det(1)})| < 10$, where $Re(z)$ and $Im(z)$ are the real and imaginary parts of z; $Re(z) = z$ and $Im(z) = 0$ if z is real.

Notes

These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

It is almost never necessary to compute either the determinant or the inverse of a matrix. While papers and reference books extensively use the notation "det(A) ≠ 0" to mean "A is nonsingular," SCILIB includes both more efficient and more reliable subprograms for detecting singularity. Similarly, references frequently use "$A^{-1}b$" to mean "the solution x of the system of linear equations $Ax = b$." Again, it is more efficient and accurate to compute the solution directly than to invert the coefficient matrix and multiply the inverse times the right-hand-side vector. This is true even if there are many systems of equations, all using the same coefficient matrix; the matrix may be factored once and the systems may be solved from the factors just as efficiently, and more accurately, than by matrix multiplication by the inverse.

Example 1 Compute only the inverse of a 6-by-6 REAL*8 matrix A stored in array A whose dimensions are 10 by 10. The more robust, but slightly slower condition number estimation subprogram is used to factor the coefficient matrix.

```
INTEGER*8 LDA,N,IPVT(10),JOB
REAL*8    A(10,10),DET(2),RCOND,WORK(10)
LDA = 10
N = 6
JOB = 1
CALL SGECO (A,LDA,N,IPVT,RCOND,WORK)
IF ( 1.0 + RCOND .NE. 1.0 ) THEN
  CALL SGEDI (A,LDA,N,IPVT,DET,WORK,JOB)
ELSE
  handle singular matrix
END IF
```

If the coefficient matrix A is determined to be nonsingular, A^{-1} overwrites the coefficient matrix A in array A.

Example 2 Compute only the determinant of a 6-by-6 REAL*8 matrix A stored in array A whose dimensions are 10 by 10. The less reliable, but slightly faster factorization subprogram is used to factor the coefficient matrix.

```
INTEGER*8 LDA,N,IPVT(10),IER,JOB
REAL*8    A(10,10),DET(2),DETA,WORK(10)
LDA = 10
N = 6
JOB = 10
CALL SGEFA (A,LDA,N,IPVT,IER)
IF ( IER .EQ. 0 ) THEN
  CALL SGEDI (A,LDA,N,IPVT,DET,WORK,JOB)
  IF ( DET(1) .EQ. 0.0 ) THEN
    DETA = 0.0
  ELSE IF ( DET(2) .LE. 306 ) THEN
    DETA = DET(1) * 10.0 ** INT(DET(2))
  ELSE
    the determinant of A is too large to evaluate
    without overflow
  END IF
ELSE
  DETA = 0.0
END IF
```

Purpose

These subprograms compute the triangular factorization of a general dense n by n matrix A. Specifically, given A, these subprograms determine an n-by-n permutation matrix P, an n-by-n unit lower-triangular matrix L, and an n-by-n upper-triangular matrix U, such that

$$PA = LU.$$

Computational singularity of A results in one or more zero diagonal elements of U. This condition is detected during factorization, and a status response is returned to indicate its occurrence. A more common situation, however, is that A is not numerically singular but happens to be ill-conditioned. When a matrix is ill-conditioned, small errors in the matrix and right-hand side and small roundoff errors introduced during the solution process itself are magnified greatly in the solution. A set of companion subprograms computes the triangular factorization of a matrix and also estimates its condition number. These companion subprograms provide a more reliable indication of singularity. The small amount of additional time they require is usually worthwhile, especially when developing a program or encountering stability or convergence problems.

The triangular factors may be used to solve a system of linear equations, $Ax = b$, by successively solving $L(Ux) = Pb$. The determinant of A can be computed as $\det(A) = \det(P) \times \det(L) \times \det(U)$. The inverse of A may be formed as $A^{-1} = U^{-1}L^{-1}P$. These operations are performed by a set of companion SCILIB subprograms whose names depend on the data type:

Data Type	Factor	Estimate Condition	Solve	Determinant or inverse
REAL*8	SGEFA	SGECO	SGESL	SGEDI
COMPLEX*16	CGEFA	CGECO	CGESL	CGEDI

The companion subprograms are documented elsewhere in this chapter.

Usage

SCILIB:

```
INTEGER*8 lda, n, ipvt(n), ier
REAL*8    a(lda, n)
CALL SGEFA (a, lda, n, ipvt, ier)

INTEGER*8    lda, n, ipvt(n), ier
COMPLEX*16 a(lda, n)
CALL CGEFA (a, lda, n, ipvt, ier)
```

Input

a Array containing the **n**-by-**n** matrix A.

lda The leading dimension of array **a** as declared in the calling program unit, with **lda** ≥ max(**n**,1).

n The order of matrix A, **n** ≥ 0.

Output

a The triangular factors replace the input matrix; the strict lower triangle of **a** contains the strict lower triangle of L and the upper triangle of **a** contains U. **a** must be preserved between the factorization call and any solve, determinant, or inverse call.

ipvt The pivot information necessary to construct the permutation matrix P. **ipvt** must be preserved between the factorization call and any solve, determinant, or inverse call.

ier Status response:

ier = 0	Normal return.
ier = k ≠ 0	if u_{kk}=0. (u_{kk} is the k-th element on the diagonal of upper triangular matrix U). Technically, this is not an error condition for these subprograms, but it does indicate that A is computationally singular and that a division by zero will occur if the factorization is used to solve a system of linear equations or to compute the matrix inverse.

Notes These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

The triangular factors are stored in a different format from the format used by the standard LINPACK subprograms, but are compatible with the SCILIB condition number estimation, solve, and determinant and inverse subprograms.

Example Factor the 6-by-6 REAL*8 matrix A stored in array A whose dimensions are 10 by 10.

```
INTEGER*8 LDA,N,IPVT(10),IER
REAL*8    A(10,10)
LDA = 10
N = 6
CALL SGEFA (A,LDA,N,IPVT,IER)
IF ( IER .NE. 0 ) THEN
  handle singular matrix
END IF
```

SGESL/CGESL — Solve Linear Equations

Purpose

Given the triangular factorization of a general dense n-by-n coefficient matrix A, and a right-hand-side n-vector b, these subprograms solve the system of linear equations $Ax = b$. Optionally, these subprograms will solve the system $A^*x = b$, where A^* is the conjugate transpose of A (the conjugate transpose of a real matrix is simply the transpose). Specifically, given an n-by-n permutation matrix P, an n-by-n unit lower-triangular matrix L, and a nonsingular n-by-n upper-triangular matrix U, such that

$$PA = LU,$$

and an n-vector b, to find x satisfying $Ax = b$, the subprograms compute

$$v = Pb,$$

then successively solve

$$Lw = v$$

and

$$Ux = w,$$

To solve $A^*x = b$, the subprograms successively solve

$$U^*v = b$$

and

$$L^*w = v,$$

and then compute

$$x = P^*w.$$

The triangular factors of the coefficient matrix may be computed by either of two companion subprograms. One computes only the factorization, using an elementary test for singularity of the coefficient matrix; it is slightly faster. The other not only computes the factorization, but also estimates the condition number of the matrix. This process takes a little more time, but is considerably more reliable. The names of the companion subprograms depend on the data type:

Data Type	Estimate Condition	Factor	Solve
REAL*8	SGECO	SGEFA	SGESL
COMPLEX*16	CGECO	CGEFA	CGESL

The companion subprograms are documented elsewhere in this chapter.

Usage

SCILIB:

```
INTEGER*8 lda, n, ipvt(n), job
REAL*8    a(lda, n), b(n)
CALL SGESL (a, lda, n, ipvt, b, job)

INTEGER*8    lda, n, ipvt(n), job
COMPLEX*16 a(lda, n), b(n)
CALL CGESL (a, lda, n, ipvt, b, job)
```

Input

a Array containing the triangular factors L and U of the **n**-by-**n** coefficient matrix A as computed by the companion factorization or condition number estimation subprogram. **a** must have been preserved between the factorization or condition number call and the solve call.

lda The leading dimension of array **a** as declared in the calling program unit, with **lda** ≥ max(**n**,1).

n The order of matrix A, **n** ≥ 0.

ipvt The pivot information necessary to construct the permutation matrix P as computed by the companion factorization or condition number estimation subprogram. **ipvt** must have been preserved between the factorization or condition number call and the solve call.

b The right-hand-side vector b.

job Option flag:

job = 0 solve $Ax = b$
job ≠ 0 solve $A^*x = b$

Output

b The solution vector x overwrites the right-hand-side vector b.

Notes

These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

Example 1

Solve a system of linear equations $Ax = b$, where A is a 6-by-6 REAL*8 matrix stored in array A whose dimensions are 10 by 10, and b is a vector 6 elements long stored in an array B of dimension 10. The more robust, but slightly slower condition number estimation subprogram is used to factor the coefficient matrix.

```
INTEGER*8 LDA,N,IPVT(10),JOB
REAL*8    A(10,10),B(10),RCOND,WORK(10)
LDA = 10
N = 6
JOB = 0
CALL SGECO (A,LDA,N,IPVT,RCOND,WORK)
IF ( 1.0 + RCOND .NE. 1.0 ) THEN
  CALL SGESL (A,LDA,N,IPVT,B,JOB)
ELSE
  handle singular matrix
END IF
```

If the coefficient matrix A is determined to be nonsingular, the solution vector x overwrites the right-hand-side b in array **b**.

Example 2 Solve a system of linear equations $A^T x = b$, where A is a 6-by-6 REAL*8 matrix stored in array A whose dimensions are 10 by 10, and b is a vector 6 elements long stored in an array B of dimension 10. The less reliable, but slightly faster, factorization subprogram is used to factor the coefficient matrix.

```
INTEGER*8 LDA,N,IPVT(10),IER,JOB
REAL*8    A(10,10),B(10)
LDA = 10
N = 6
JOB = 1
CALL SGEFA (A,LDA,N,IPVT,IER)
IF ( IER .EQ. 0 ) THEN
  CALL SGESL (A,LDA,N,IPVT,B,JOB)
ELSE
  handle singular matrix
END IF
```

If the coefficient matrix A is determined to be nonsingular, the solution vector x overwrites the right-hand-side b in array **b**.

Solve Tridiagonal Linear Equations — SGTSL/CGTSL

Purpose

Given an n-by-n tridiagonal matrix A, and a right-hand-side n-vector b, these subprograms solve the system of linear equations $Ax = b$. A tridiagonal matrix $A = \{a_{ij}\}$ is a matrix whose nonzero elements lie only on the principal diagonal ($i = j$), the subdiagonal ($i = j+1$), and the superdiagonal ($i = j-1$) of the matrix.

Matrix Storage

The following example illustrates the storage of general tridiagonal matrices. Consider the following tridiagonal matrix of order $n = 7$:

11	12	0	0	0	0	0
21	22	23	0	0	0	0
0	32	33	34	0	0	0
0	0	43	44	45	0	0
0	0	0	54	55	56	0
0	0	0	0	65	66	67
0	0	0	0	0	76	77

The subdiagonal is stored in array **dl**, the principal diagonal is stored in array **d**, and the superdiagonal is stored in array **du**, as follows:

i	dl(i)	d(i)	du(i)
1	*	11	12
2	21	22	23
3	32	33	34
4	43	44	45
5	54	55	56
6	65	66	67
7	76	77	*

The asterisks represent elements whose initial contents are not used.

Usage

SCILIB:

```
INTEGER*8 n, ier
REAL*8    dl(n), d(n), du(n), b(n)
CALL SGTSL (n, dl, d, du, b, ier)

INTEGER*8    n, ier
COMPLEX*16 dl(n), d(n), du(n), b(n)
CALL CGTSL (n, dl, d, du, b, ier)
```

Input

n The order of matrix A, **n** > 0.

dl Array containing the subdiagonal of the tridiagonal matrix, **dl**(i) = $a_{i,i-1}$, $i = 2, 3, \ldots,$ **n**. On return, **dl** is destroyed, including **dl**(1).

d Array containing the principal diagonal of the tridiagonal matrix, **d**(i) = a_{ii}, $i = 1, 2, \ldots,$ **n**. On return, **d** is destroyed.

du Array containing the superdiagonal of the tridiagonal matrix, **du**(i) = $a_{i,i+1}$, $i = 1, 2, \ldots,$ **n**-1. On return, **du** is destroyed, including **du**(**n**).

b The right-hand-side vector b.

SGTSL/CGTSL — Continued

Output

b The solution vector x overwrites the right-hand-side vector b if **ier** $= 0$ is returned.

ier Status response:

ier $= 0$	Normal return.
ier $= k \neq 0$	if the k-th element of the diagonal becomes zero.

Notes

These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

Example

Solve a system of linear equations $Ax = b$, where A is a 7-by-7 REAL*8 tridiagonal matrix. The subdiagonal of A is stored in array DL, the principal diagonal is stored in array D, and the superdiagonal is stored in array DU. b is a vector 7 elements long stored in an array B.

```
INTEGER*8 N,IER
REAL*8    DL(10),D(10),DU(10),B(10)
N = 7
CALL SGTSL (N,DL,D,DU,B,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
END IF
```

Purpose

These subprograms compute the Cholesky factorization and estimate the condition number of an n-by-n positive definite band matrix A stored in a two-dimensional array. A matrix A is positive definite if and only if it is Hermitian; that is, A is equal to A^*, its conjugate transpose, and the quadratic form x^*Ax is positive for all nonzero vectors x. (The conjugate transpose of a real matrix or vector is simply the transpose.)

A positive definite band matrix is a positive definite matrix whose nonzero elements all are fairly near the principal diagonal. Specifically, $a_{ij} = 0$ if $|i-j| > kd$ for some integer kd. The smallest such kd for a given matrix is called the half bandwidth, and $2kd+1$ is called the total bandwidth.

Tridiagonal matrices are the special case $kd = 1$. They can be handled more efficiently by the subprograms SPTSL and CPTSL.

Specifically, given A, these subprograms determine an n-by-n upper-triangular band matrix R, such that

$$A = R^*R$$

and compute an estimate of $\kappa(A)$, the condition number of A. Refer to "Condition Number" in the introduction to this chapter for a discussion of $\kappa(A)$. When a matrix is ill-conditioned, $\kappa(A)$ is large, so small errors in the matrix and right-hand side and small roundoff errors introduced during the solution process itself are magnified greatly in the solution.

Since $1 < \kappa(A) \leq \infty$, these subprograms actually compute the reciprocal condition number, $1/\kappa(A)$. The reciprocal condition number has the interpretation that if $1/\kappa(A)$ approximately equals 10^{-d}, elements of x can be expected to have d fewer significant digits of accuracy than the elements of A or b. Consequently, if errors in the coefficient matrix and right-hand side exceed $1/\kappa(A)$, or if $1/\kappa(A)$ is negligible compared to 1.0, then x may have no significant digits at all.

A set of companion subprograms computes the Cholesky factorization of a matrix without estimating its condition number. These companion subprograms are faster but provide a less reliable indication of singularity.

The triangular factors may be used to solve a system of linear equations, $Ax = b$, by successively solving $R^*(Rx) = b$. The determinant of A can be computed as $\det(A) = \det(R)^2$. These operations are performed by a set of companion SCILIB subprograms whose names depend on the data type:

Data Type	Estimate Condition	Factor	Solve	Determinant
REAL*8	SPBCO	SPBFA	SPBSL	SPBDI
COMPLEX*16	CPBCO	CPBFA	CPBSL	CPBDI

The inverse of A will usually be a full n-by-n matrix, which cannot be stored in the band storage of A. Therefore, no direct provision is made for computing A^{-1}. Calculations formulated in terms of matrix inverses are invariably more efficient when expressed in terms of the solution of sets of linear equations.

SPBCO/CPBCO — Continued

Matrix Storage

Because it is not necessary to store or operate on the zeros outside the band of A, and since the Cholesky factorization of A may be computed from either triangle of A, you need only provide the band within the upper triangle. 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 triangle.

The following examples illustrate the storage of positive definite band matrices. Consider the following matrix A of order $n = 7$ and half bandwidth $kd = 2$:

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

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**.

Usage

SCILIB:

```
INTEGER*8 ldab, n, kd, ier
REAL*8    ab(ldab, n), rcond, work(n)
CALL SPBCO (ab, ldab, n, kd, rcond, work, ier)

INTEGER*8    ldab, n, kd, ier
COMPLEX*16 ab(ldab, n), work(n)
REAL*8       rcond
CALL CPBCO (ab, ldab, n, kd, rcond, work, ier)
```

Input

ab Array containing the upper triangle of the **n**-by-**n** positive definite band matrix A in the compressed form described above. If $0 \le j-i \le kd$, then a_{ij} is stored in **ab**$(kd+1+i-j,j)$. Columns of the upper triangle of A are stored in the columns of **ab**, and diagonals of the upper triangle 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** $\ge$ **kd**+1.

n The order of matrix A, **n** > 0.

kd The half bandwidth of A, i.e., the number of diagonals above the principal diagonal in the band, $0 \le$ **kd** $<$ **n**.

Working storage

work An array of size **n**, used for work space.

Output

ab The Cholesky factor R replaces the input matrix. The factorization is not complete if **ier** is nonzero. **ab** must be preserved between the condition number estimation call and any solve or determinant call.

rcond An estimate of the reciprocal condition number, $1/\kappa(A)$, if **ier** is zero; unchanged from its input value if **ier** is nonzero. If **ier** is zero and **rcond** is so small that the logical expression

1.0 + rcond .EQ. 1.0

is true, A can be regarded as singular to working precision.

ier Status response:

ier = 0	Normal return—factorization complete.
ier = k ≠ 0	The leading submatrix of order k is not computationally positive definite, possibly because of roundoff error.

Notes

These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

Example

Factor the 7-by-7 REAL*8 positive definite band matrix A whose half bandwidth is 2 and whose upper triangle is stored in the upper triangle of array AB whose dimensions are 5 by 10, and estimate its reciprocal condition number.

```
INTEGER*8 LDAB,N,KD,IER
REAL*8    AB(5,10),RCOND,WORK(10)
LDAB = 5
N = 7
M = 2
CALL SPBCO (AB,LDAB,N,KD,RCOND,WORK,IER)
IF ( IER .NE. 0 ) THEN
  handle indefinite matrix
ELSE IF ( 1.0 + RCOND .EQ. 1.0 ) THEN
  handle singular matrix
END IF
```

SPBDI/CPBDI — Determinant

Purpose

Given the Cholesky factorization of an n-by-n positive definite band matrix A, these subprograms evaluate the determinant of A. No provision is made to compute A^{-1} because it will usually be a full n-by-n matrix, which cannot be stored in the band storage of A. Moreover, it is almost never necessary to compute the inverse of a matrix. Mathematical references frequently use "$A^{-1}b$" to mean "the solution x of the system of linear equations $Ax = b$." It is more efficient and accurate to compute the solution directly than to invert the coefficient matrix and multiply the inverse times the right-hand-side vector. This is true even if there are many systems of equations, all using the same coefficient matrix; the matrix may be factored once and the systems may be solved from the factors just as efficiently, and more accurately, than by matrix multiplication by the inverse.

Specifically, given an n-by-n upper-triangular band matrix R, such that

$$A = R^*R,$$

where R^* is the conjugate transpose of R, the subprograms compute

$$\det(A) = \det(R)^2.$$

The Cholesky factorization of the coefficient matrix may be computed by either of two companion subprograms. One computes only the factorization, using an elementary test for singularity of the coefficient matrix; it is slightly faster. The other not only computes the factorization, but also estimates the condition number of the matrix. This process takes a little more time, but is considerably more reliable. The names of the companion subprograms depend on the data type:

Data Type	Estimate Condition	Factor	Determinant
REAL*8	SPBCO	SPBFA	SPBDI
COMPLEX*16	CPBCO	CPBFA	CPBDI

The companion subprograms are documented elsewhere in this chapter.

Usage

SCILIB:

```
INTEGER*8 ldab, n, kd
REAL*8    ab(ldab, n), det(2)
CALL SPBDI (ab, ldab, n, kd, det)

INTEGER*8    ldab, n, kd
COMPLEX*16 ab(ldab, n)
REAL*8       det(2)
CALL CPBDI (ab, ldab, n, kd, det)
```

Input

ab Array containing the Cholesky factor R of the **n**-by-**n** positive definite band matrix A as computed by the companion factorization or condition number estimation subprogram. **ab** must have been preserved between the factorization or condition number call and the determinant call.

ldab The leading dimension of array **ab** as declared in the calling program unit, with **ldab** ≥ **n**+1.

n The order of matrix A, **n** ≥ 0.

kd The half bandwidth of A, i.e., the number of diagonals above the principal diagonal in the band, $0 \leq$ **kd** $<$ **n**.

Output

det The determinant of A, in the form $\det(A) =$ **det**(1) $\times 10^{\text{det}(2)}$. This expression may underflow or overflow if evaluated; on the CONVEX supercomputer, underflows automatically flush to zero, but overflows normally terminate execution. For REAL*8 and COMPLEX*16, overflow cannot occur if **det(2)** $\leq$ 306. If evaluation is safe, an efficient way to do it is with the statement

$$\det(A) = \textbf{det(1)} * \textbf{10.0} ** \textbf{INT(det(2))}$$

The value stored in **det(2)** is an integer in REAL form. **det(1)** is normalized so that **det(1)** $= 0$ or $1 \leq$ **det(1)** < 10.

Notes These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

It is almost never necessary to compute the determinant of a matrix. While it is true that papers and reference books make extensive use of the notation "$\det(A) \neq 0$" to mean "A is nonsingular," SCILIB includes both more efficient and more reliable subprograms for detecting singularity.

Example Compute the determinant of a 7-by-7 REAL*8 matrix A whose half bandwidth is 2 and whose upper triangle is stored in the upper triangle of array AB whose dimensions are 5 by 10. The less reliable, but slightly faster factorization subprogram is used to factor the coefficient matrix.

```
INTEGER*8 LDAB,N,KD,IER
REAL*8    AB(5,10),DET(2),DETA
LDAB = 5
N = 7
KD = 2
CALL SPBFA (AB,LDAB,N,KD,IER)
IF ( IER .EQ. 0 ) THEN
  CALL SPBDI (AB,LDAB,N,KD,DET)
  IF ( DET(1) .EQ. 0.0 ) THEN
    DETA = 0.0
  ELSE IF ( DET(2) .LE. 306 ) THEN
    DETA = DET(1) * 10.0 ** INT(DET(2))
  ELSE
    the determinant of A is too large to evaluate
    without overflow
  END IF
ELSE
  DETA = 0.0
END IF
```

Purpose

These subprograms compute Cholesky factorization of an n-by-n positive definite band matrix A stored in a two-dimensional array. A matrix A is positive definite if and only if it is Hermitian; that is, A is equal to A^*, its conjugate transpose, and the quadratic form x^*Ax is positive for all nonzero vectors x. (The conjugate transpose of a real matrix or vector is simply the transpose.)

A positive definite band matrix is a positive definite matrix whose nonzero elements all are fairly near the principal diagonal. Specifically, $a_{ij} = 0$ if $|i-j| > kd$ for some integer kd. The smallest such kd for a given matrix is called the half bandwidth, and $2m+1$ is called the total bandwidth.

Tridiagonal matrices are the special case $kd = 1$. They can be handled more efficiently by the subprograms SPTSL and CPTSL.

Specifically, given A, these subprograms determine an n-by-n upper-triangular band matrix R, such that

$$A = R^*R.$$

Computational singularity of A results in one or more zero diagonal elements of R, or, more frequently, in the loss of positive definiteness as evidenced by a negative diagonal element. This condition is detected during factorization, and a status response is returned to indicate its occurrence. A more common situation, however, is that A is not numerically singular but is ill-conditioned. When a matrix is ill-conditioned, small errors in the matrix and right-hand side and small roundoff errors introduced during the solution process itself are magnified greatly in the solution. A set of companion subprograms computes Cholesky factorization of a matrix and estimates its condition number. These companion subprograms provide a more reliable indication of singularity. The small amount of additional time they require is usually worthwhile, especially when developing a program or encountering stability or convergence problems.

The triangular factors may be used to solve a system of linear equations, $Ax = b$, by successively solving $R^*(Rx) = b$. The determinant of A can be computed as $\det(A) = \det(R)^2$. These operations are performed by a set of companion SCILIB subprograms whose names depend on the data type:

Data Type	Factor	Estimate Condition	Solve	Determinant
REAL*8	SPBFA	SPBCO	SPBSL	SPBDI
COMPLEX*16	CPBFA	CPBCO	CPBSL	CPBDI

The companion subprograms are documented elsewhere in this chapter.

The inverse of A will usually be a full n-by-n matrix, which cannot be stored in the band storage of A. Therefore, no direct provision is made for computing A^{-1}. Calculations formulated in terms of matrix inverses are invariably more efficient when expressed in terms of the solution of sets of linear equations.

Matrix Storage

Because it is not necessary to store or operate on the zeros outside the band of A, and since the Cholesky factorization of A may be computed from either triangle of A, you need only provide the band within the upper triangle. 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 triangle.

The following examples illustrate the storage of positive definite band matrices. Consider the following matrix A of order $n = 7$ and half bandwidth $kd = 2$:

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

The upper triangle of A is stored in an array **ab** with at least $kd+1 = 3$ rows and 7 columns:

*	*	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**.

Usage

SCILIB:

```
INTEGER*8 ldab, n, kd, ier
REAL*8    ab(ldab, n)
CALL SPBFA (ab, ldab, n, kd, ier)

INTEGER*8    ldab, n, kd, ier
COMPLEX*16 ab(ldab, n)
CALL CPBFA (ab, ldab, n, kd, ier)
```

Input

ab Array containing the upper triangle of the **n**-by-**n** positive definite band matrix A in the compressed form described above. If $0 \le j-i \le kd$, then a_{ij} is stored in **ab**$(\mathbf{kd}+1+i-j,j)$. 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**.

ldab The leading dimension of array **ab** as declared in the calling program unit, with **ldab** $\ge$ **kd**+1.

n The order of matrix A, **n** > 0.

kd The half bandwidth of A, i.e., the number of diagonals above the principal diagonal in the band, $0 \le$ **kd** $<$ **n**.

SPBFA/CPBFA — Continued

Output

ab The Cholesky factor R replaces the input matrix. The factorization is not complete if **ier** is nonzero. **ab** must be preserved between the condition number estimation call and any solve or determinant call.

ier Status response:

ier = 0	Normal return—factorization complete.
ier = k ≠ 0	The leading submatrix of order k is not computationally positive definite, possibly because of roundoff error.

Notes

These subprograms are usage-compatible with the standard LINPACK subprograms with the same names.

Example

Factor the 7-by-7 REAL*8 positive definite band matrix A whose half bandwidth is 2 and whose upper triangle is stored in the upper triangle of array AB whose dimensions are 5 by 10, and estimate its reciprocal condition number.

```
INTEGER*8 LDAB,N,KD,IER
REAL*8    AB(5,10)
LDAB = 5
N = 7
KD = 2
CALL SPBFA (AB,LDAB,N,KD,IER)
IF ( IER .NE. 0 ) THEN
  handle singular or indefinite matrix
END IF
```

Purpose

Given the Cholesky factorization of an n-by-n positive definite band matrix A, and a right-hand-side n-vector b, these subprograms solve the system of linear equations $Ax = b$. Specifically, given an n-by-n upper-triangular band matrix R, such that

$$A = R^*R,$$

where R^* is the conjugate transpose of R, and an n-vector b, to find x satisfying $Ax = b$, the subprograms successively solve

$$R^*w = b$$

and

$$Rx = w.$$

Cholesky factorization of the coefficient matrix may be computed by either of two companion subprograms. One computes only the factorization, using an elementary test for singularity of the coefficient matrix; it is slightly faster. The other not only computes factorization, but also estimates the condition number of the matrix. This process takes a little more time, but is considerably more reliable. The names of the companion subprograms depend on the data type:

Data Type	Estimate Condition	Factor	Solve
REAL*8	SPBCO	SPBFA	SPBSL
COMPLEX*16	CPBCO	CPBFA	CPBSL

The companion subprograms are documented elsewhere in this chapter.

Usage

SCILIB:

```
INTEGER*8 ldab, n, kd
REAL*8    ab(ldab, n), b(n)
CALL SPBSL (ab, ldab, n, kd, b)

INTEGER*8    ldab, n, kd
COMPLEX*16 ab(ldab, n), b(n)
CALL CPBSL (ab, ldab, n, kd, b)
```

Input

ab Array containing the Cholesky factor R of the **n**-by-**n** positive definite band matrix A as computed by the companion factorization or condition number estimation subprogram. **ab** must have been preserved between the factorization or condition number call and the solve call.

ldab The leading dimension of array **ab** as declared in the calling program unit, with **ldab** ≥ **kd**+1.

n The order of matrix A, **n** ≥ 0.

kd The half bandwidth of A, i.e., the number of diagonals above the principal diagonal in the band, 0 ≤ **kd** < **n**.

b The right-hand-side vector b.

Output

b The solution vector x overwrites the right-hand-side vector b.

Notes

These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

SPBSL/CPBSL — Continued

Example Solve a system of linear equations $Ax = b$, where A is a 7-by-7 REAL*8 positive definite band matrix with half bandwidth 2. The upper triangle of A is stored in array AB whose dimensions are 5 by 10. b is a vector 7 elements long stored in an array B of dimension 10. The more robust, but slightly slower condition number estimation subprogram is used to factor the coefficient matrix.

```
INTEGER*8 LDAB,N,KD,IER
REAL*8    AB(5,10),B(10),RCOND,WORK(10)
LDAB = 5
N = 7
KD = 2
CALL SPBCO (AB,LDAB,N,KD,RCOND,WORK,IER)
IF ( IER .NE. 0 ) THEN
  handle indefinite matrix
ELSE IF ( 1.0 + RCOND .NE. 1.0 ) THEN
  CALL SPBSL (AB,LDAB,N,KD,B)
ELSE
  handle singular matrix
END IF
```

If the coefficient matrix A is determined to be positive definite and nonsingular, the solution vector x overwrites the right-hand-side b in array B.

Purpose

These subprograms compute Cholesky factorization and estimate the condition number of an n-by-n positive definite matrix A stored in a two-dimensional array and estimate its condition number. A matrix A is positive definite if and only if it is Hermitian; that is, A is equal to A^*, its conjugate transpose, and the quadratic form x^*Ax is positive for all nonzero vectors x. (The conjugate transpose of a real matrix or vector is simply the transpose.)

Specifically, given A, these subprograms determine an n-by-n upper-triangular matrix R, such that

$$A = R^*R$$

and compute an estimate of $\kappa(A)$, the condition number of A. Refer to "Condition Number" in the introduction to this chapter for a discussion of $\kappa(A)$. When a matrix is ill-conditioned, $\kappa(A)$ is large, so small errors in the matrix and right-hand side and small roundoff errors introduced during the solution process itself are magnified greatly in the solution.

Since $1 < \kappa(A) \leq \infty$, these subprograms actually compute the reciprocal condition number, $1/\kappa(A)$. The reciprocal condition number has the interpretation that if $1/\kappa(A)$ approximately equals 10^{-d}, elements of x can be expected to have d fewer significant digits of accuracy than the elements of A or b. Consequently, if errors in the coefficient matrix and right-hand side exceed $1/\kappa(A)$, or if $1/\kappa(A)$ is negligible compared to 1.0, then x may have no significant digits at all.

A set of companion subprograms computes Cholesky factorization of a matrix without estimating its condition number. These companion subprograms are faster but provide a less reliable indication of singularity.

The triangular factors may be used to solve a system of linear equations, $Ax = b$, by successively solving $R^*(Rx) = b$. The determinant of A can be computed as $\det(A) = \det(R)^2$. The inverse of A may be formed as $A^{-1} = R^{-1}R^{-*}$, where R^{-*} is the conjugate transpose of the inverse of R. These operations are performed by a set of companion SCILIB subprograms whose names depend on the data type:

Data Type	Estimate Condition	Factor	Solve	Determinant or inverse
REAL*8	SPOCO	SPOFA	SPOSL	SPODI
COMPLEX*16	CPOCO	CPOFA	CPOSL	CPODI

The companion subprograms are documented elsewhere in this chapter.

Matrix Storage

Because the Cholesky factorization of A may be computed from either triangle of A, you need only provide the upper triangle. Provide it in a two-dimensional array large enough to hold the entire array. The lower triangle of the array is not referenced.

Usage

SCILIB:

```
INTEGER*8 lda, n, ier
REAL*8    a(lda, n), rcond, work(n)
CALL SPOCO (a, lda, n, rcond, work, ier)
```

SPOCO/CPOCO — Continued

```
INTEGER*8   lda, n, ier
COMPLEX*16  a(lda, n), work(n)
REAL*8      rcond
CALL CPOCO (a, lda, n, rcond, work, ier)
```

Input

a Array containing the diagonal and upper triangle of the **n**-by-**n** positive definite matrix A. The elements in the strict lower triangle of **a** are not referenced.

lda The leading dimension of array **a** as declared in the calling program unit, with **lda** ≥ max(**n**,1).

n The order of matrix A, **n** ≥ 0.

Working storage

work An array of size **n**, used for work space.

Output

a The Cholesky factor R replaces the input matrix A in the upper triangle of **a**. The strict lower triangle of **a** is unchanged. The factorization is not complete if **ier** is nonzero. **a** must be preserved between the condition number estimation call and any solve, determinant, or inverse call.

rcond An estimate of the reciprocal condition number, $1/\kappa(A)$, if **ier** is zero; unchanged from its input value if **ier** is nonzero. If **ier** is zero and **rcond** is so small that the logical expression

1.0 + rcond .EQ. 1.0

is true, then A can be regarded as singular to working precision.

ier Status response:

ier = 0	Normal return—factorization complete.
ier = k ≠ 0	The leading submatrix of order k is not computationally positive definite, possibly because of roundoff error.

Notes

These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

Example

Factor the 6-by-6 REAL*8 positive definite matrix A whose upper triangle is stored in the upper triangle of array A whose dimensions are 10 by 10, and estimate its reciprocal condition number.

```
INTEGER*8 LDA,N,IER
REAL*8    A(10,10),RCOND,WORK(10)
LDA = 10
N = 6
CALL SPOCO (A,LDA,N,RCOND,WORK,IER)
IF ( IER .NE. 0 ) THEN
  handle indefinite matrix
ELSE IF ( 1.0 + RCOND .EQ. 1.0 ) THEN
  handle singular matrix
END IF
```

Purpose

Given the Cholesky factorization of an n-by-n positive definite coefficient matrix A, these subprograms evaluate the determinant of A and/or compute A^{-1}. Specifically, given an n-by-n upper-triangular matrix R, such that

$$A = R^*R,$$

where R^* is the conjugate transpose of R, the subprograms compute

$$\det(A) = \det(R)^2$$

and/or

$$A^{-1} = R^{-1}R^{-*}$$

where R^{-*} is the conjugate transpose of the inverse of R.

The Cholesky factorization of the coefficient matrix may be computed by either of two companion subprograms. One computes only the factorization, using an elementary test for singularity of the coefficient matrix; it is slightly faster. The other not only computes factorization, but also estimates the condition number of the matrix. This process takes a little more time, but is considerably more reliable, especially when A^{-1} is desired. The names of the companion subprograms depend on the data type:

Data Type	Estimate Condition	Factor	Determinant or inverse
REAL*8	SPOCO	SPOFA	SPODI
COMPLEX*16	CPOCO	CPOFA	CPODI

The companion subprograms are documented elsewhere in this chapter.

Usage

SCILIB:

```
INTEGER*8 lda, n, job
REAL*8    a(lda, n), det(2)
CALL SPODI (a, lda, n, det, job)
INTEGER*8 lda, n, job
COMPLEX*16 a(lda, n)
REAL*8    det(2)
CALL CPODI (a, lda, n, det, job)
```

Input

a Array containing the Cholesky factor R of the **n**-by-**n** positive definite coefficient matrix A in its upper triangle, as computed by the companion factorization or condition number estimation subprogram. The upper triangle of **a** must have been preserved between the factorization or condition number call and the determinant or inverse call.

lda The leading dimension of array **a** as declared in the calling program unit, with **lda** ≥ max(**n**,1).

n The order of matrix A, **n** ≥ 0.

job Option flag:

job = 1 compute only A^{-1}
job = 10 compute only $\det(A)$
job = 11 compute both A^{-1} and $\det(A)$

Output

a Unchanged if A^{-1} is not requested. Otherwise, the upper triangle of A^{-1} overwrites the Cholesky factor of the coefficient matrix. The strict lower triangle of **a** is never changed.

det Not referenced if the determinant is not requested. Otherwise, the determinant of A, in the form $\det(A) = \mathbf{det}(1) \times 10^{\mathbf{det}(2)}$. This expression may underflow or overflow if evaluated; on the CONVEX supercomputer, underflows automatically flush to zero, but overflows normally terminate execution. For REAL*8 and COMPLEX*16, overflow cannot occur if $\mathbf{det}(2) \leq 306$. If evaluation is safe, an efficient way to do it is with the statement

$$\det(A) = \mathbf{det(1)} * \mathbf{10.0} ** \mathbf{INT(det(2))}$$

Refer to "Example 2."

The value stored in **det(2)** is an integer in REAL form. **det(1)** is normalized so that $\mathbf{det}(1) = 0$ or $1 \leq \mathbf{det}(1) < 10$.

Notes

These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

It is almost never necessary to compute either the determinant or the inverse of a matrix. While papers and reference books extensively use the notation "$\det(A) \neq 0$" to mean "A is nonsingular," SCILIB includes more efficient and more reliable subprograms for detecting singularity. Similarly, references frequently use "$A^{-1}b$" to mean "the solution x of the system of linear equations $Ax = b$." Again, it is more efficient and accurate to compute the solution directly than to invert the coefficient matrix and multiply the inverse times the right-hand-side vector. This is true even if there are many systems of equations, all using the same coefficient matrix; the matrix may be factored once and the systems may be solved from the factors just as efficiently, and more accurately, than by matrix multiplication by the inverse.

Example 1 Compute only the inverse of a 6-by-6 REAL*8 positive definite matrix A whose upper triangle is stored in the upper triangle of array A whose dimensions are 10 by 10. The more robust, but slightly slower condition number estimation subprogram is used to factor the coefficient matrix.

```
INTEGER*8 LDA,N,IER,JOB
REAL*8    A(10,10),DET(2),RCOND,WORK(10)
LDA = 10
N = 6
JOB = 1
CALL SPOCO (A,LDA,N,RCOND,WORK,IER)
IF ( IER .NE. 0 ) THEN
  handle indefinite matrix
ELSE IF ( 1.0 + RCOND .NE. 1.0 ) THEN
  CALL SPODI (A,LDA,N,DET,JOB)
ELSE
  handle singular matrix
END IF
```

If the coefficient matrix A is determined to be nonsingular, A^{-1} overwrites the coefficient matrix A in array **a**.

Example 2 Compute only the determinant of a 6-by-6 REAL*8 matrix A stored in array A whose dimensions are 10 by 10. The less reliable, but slightly faster factorization subprogram is used to factor the coefficient matrix.

```
INTEGER*8 LDA,N,IER,JOB
REAL*8    A(10,10),DET(2),DETA
LDA = 10
N = 6
JOB = 10
CALL SPOFA (A,LDA,N,IER)
IF ( IER .EQ. 0 ) THEN
  CALL SPODI (A,LDA,N,DET,JOB)
  IF ( DET(1) .EQ. 0.0 ) THEN
    DETA = 0.0
  ELSE IF ( DET(2) .LE. 306 ) THEN
    DETA = DET(1) * 10.0 ** INT(DET(2))
  ELSE
    the determinant of A is too large to evaluate
    without overflow
  END IF
ELSE
  DETA = 0.0
END IF
```

SPOFA/CPOFA — Cholesky Factorization

Purpose

These subprograms compute the Cholesky factorization of an n-by-n positive definite matrix A stored in a two-dimensional array. A matrix A is positive definite if and only if it is Hermitian; that is, A is equal to A^*, its conjugate transpose, and the quadratic form x^*Ax is positive for all nonzero vectors x. (The conjugate transpose of a real matrix or vector is simply the transpose.) Specifically, given A, these subprograms determine an n-by-n upper-triangular matrix R, such that

$$A = R^*R.$$

Computational singularity of A results in one or more zero diagonal elements of R, or, more frequently, in the loss of positive definiteness as evidenced by a negative diagonal element. This condition is detected during the factorization, and a status response is returned to indicate its occurrence. A more common situation, however, is that A is not numerically singular but happens to be ill-conditioned. When a matrix is ill-conditioned, small errors in the matrix and right-hand side and small roundoff errors introduced during the solution process itself are magnified greatly in the solution. A set of companion subprograms computes the Cholesky factorization of a matrix and also estimates its condition number. These companion subprograms provide a more reliable indication of singularity. The small amount of additional time they require is usually worthwhile, especially when developing a program or encountering stability or convergence problems.

The triangular factors may be used to solve a system of linear equations, $Ax = b$, by successively solving $R^*(Rx) = b$. The determinant of A can be computed as $\det(A) = \det(R)^2$. The inverse of A may be formed as $A^{-1} = R^{-1}R^{-*}$, where R^{-*} is the conjugate transpose of the inverse of R. These operations are performed by a set of companion SCILIB subprograms whose names depend on the data type:

Data Type	Factor	Estimate Condition	Solve	Determinant or inverse
REAL*8	SPOFA	SPOCO	SPOSL	SPODI
COMPLEX*16	CPOFA	CPOCO	CPOSL	CPODI

The companion subprograms are documented elsewhere in this chapter.

Matrix Storage

Because the Cholesky factorization of A may be computed from either triangle of A, you need only provide the upper triangle. Provide it in a two-dimensional array large enough to hold the entire array. The lower triangle of the array is not referenced.

Usage

SCILIB:

```
INTEGER*8 lda, n, ier
REAL*8    a(lda, n)
CALL SPOFA (a, lda, n, ier)

INTEGER*8    lda, n, ier
COMPLEX*16 a(lda, n)
CALL CPOFA (a, lda, n, ier)
```

Input

a Array containing the diagonal and upper triangle of the **n**-by-**n** positive definite matrix A. The elements of the strict lower triangle are not referenced.

lda The leading dimension of array **a** as declared in the calling program unit, with **lda** ≥ max(**n**,1).

n The order of matrix A, **n** $\geq$ 0.

Output

a The Cholesky factor R replaces the input matrix A in the upper triangle of **a**. The strict lower triangle of **a** is unchanged. The factorization is not complete if **ier** is nonzero. **a** must be preserved between the factorization call and any solve, determinant, or inverse call.

ier Status response:

ier = 0	Normal return—factorization complete.
ier = k $\neq$ 0	The leading submatrix of order k is not computationally positive definite, possibly because of roundoff error.

Notes These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

Example Factor the 6-by-6 REAL*8 positive definite matrix A whose upper triangle is stored in the upper triangle of array A whose dimensions are 10 by 10.

```
INTEGER*8 LDA,N,IER
REAL*8    A(10,10)
LDA = 10
N = 6
CALL SPOFA (A,LDA,N,IER)
IF ( IER .NE. 0 ) THEN
  handle indefinite matrix
END IF
```

SPOSL/CPOSL — Solve Linear Equations

Purpose

Given the Cholesky factorization of an n-by-n positive definite coefficient matrix A, and a right-hand-side n-vector b, these subprograms solve the system of linear equations $Ax = b$. Specifically, given an n-by-n upper-triangular matrix R, such that

$$A = R^*R,$$

where R^* is the conjugate transpose of R, and an n-vector b, to find x satisfying $Ax = b$, the subprograms successively solve

$$R^*w = b$$

and

$$Rx = w.$$

The Cholesky factorization of the coefficient matrix may be computed by either of two companion subprograms. One computes only the factorization, using an elementary test for singularity of the coefficient matrix; it is slightly faster. The other not only computes the factorization, but also estimates the condition number of the matrix. This process takes a little more time, but is considerably more reliable. The names of the companion subprograms depend on the data type:

Data Type	Estimate condition	Factor	Solve
REAL*8	SPOCO	SPOFA	SPOSL
COMPLEX*16	CPOCO	CPOFA	CPOSL

The companion subprograms are documented elsewhere in this chapter.

Usage

SCILIB:

```
INTEGER*8 lda, n
REAL*8    a(lda, n), b(n)
CALL SPOSL (a, lda, n, b)

INTEGER*8    lda, n
COMPLEX*16 a(lda, n), b(n)
CALL CPOSL (a, lda, n, b)
```

Input

a Array containing the Cholesky factor R of the **n**-by-**n** positive definite coefficient matrix A in its upper triangle, as computed by the companion factorization or condition number estimation subprogram. The upper triangle of **a** must have been preserved between the factorization or condition number call and the solve call.

lda The leading dimension of array **a** as declared in the calling program unit, with **lda** ≥ max(**n**,1).

n The order of matrix A, **n** ≥ 0.

b The right-hand-side vector b.

Output

b The solution vector x overwrites the right-hand-side vector b.

Notes

These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

Example Solve a system of linear equations $Ax = b$, where A is a 6-by-6 REAL*8 positive definite matrix whose upper triangle is stored in array A whose dimensions are 10 by 10, and where b is a vector 6 elements long stored in an array B of dimension 10. The more robust, but slightly slower condition number estimation subprogram is used to factor the coefficient matrix.

```
INTEGER*8 LDA,N,IER
REAL*8    A(10,10),B(10),RCOND,WORK(10)
LDA = 10
N = 6
CALL SPOCO (A,LDA,N,RCOND,WORK,IER)
IF ( IER .NE. 0 ) THEN
  handle indefinite matrix
ELSE IF ( 1.0 + RCOND .NE. 1.0 ) THEN
  CALL SPOSL (A,LDA,N,B)
ELSE
  handle singular matrix
END IF
```

If the coefficient matrix A is determined to be positive definite and nonsingular, the solution vector x overwrites the right-hand-side b in array b.

SPTSL/CPTSL Solve Positive Definite Tridiagonal Linear Equations

Purpose

Given an n-by-n positive definite tridiagonal matrix A, and a right-hand-side n-vector b, these subprograms solve the system of linear equations $Ax = b$. A matrix A is positive definite if and only if it is Hermitian; that is, A is equal to A^*, its conjugate transpose, and the quadratic form x^*Ax is positive for all nonzero vectors x. (The conjugate transpose of a real matrix or vector is simply the transpose.)

A positive definite tridiagonal matrix is a positive definite matrix $A = \{a_{ij}\}$ whose nonzero elements lie only on the principal diagonal ($i = j$), the subdiagonal ($i = j+1$), and the superdiagonal ($i = j-1$) of the matrix. Because of conjugate symmetry, the principal diagonal is always real, and the subdiagonal and superdiagonal are complex conjugates of each other. Thus, it is not necessary to store both the subdiagonal and the superdiagonal.

Matrix Storage

The following example illustrates the storage of a real symmetric or complex Hermitian tridiagonal matrix. Consider the following symmetric tridiagonal matrix of order $n = 7$

11	12	0	0	0	0	0
12	22	23	0	0	0	0
0	23	33	34	0	0	0
0	0	34	44	45	0	0
0	0	0	45	55	56	0
0	0	0	0	56	66	67
0	0	0	0	0	67	77

then the principal diagonal is stored in array **d** and the superdiagonal is stored in array **e** as follows:

i	d(i)	e(i)
1	11	12
2	22	23
3	33	34
4	44	45
5	55	56
6	66	67
7	77	*

The asterisk represents an element that is not referenced.

Usage

SCILIB:

```
INTEGER*8 n
REAL*8    d(n), e(n-1), b(n)
CALL SPTSL (n, d, e, b)

INTEGER*8    n
COMPLEX*16 d(n), e(n-1), b(n)
CALL CPTSL (n, d, e, b)
```

Input

n The order of matrix A, **n** > 0.

d Array containing the principal diagonal of the tridiagonal matrix, d(i) = a_{ii}, $i = 1, 2, \ldots,$ **n**. For CPTSL and ZPTSL, only the real parts of **d** are used. On return, **d** is destroyed.

	e	Array containing the superdiagonal of the tridiagonal matrix, $\mathbf{e}(i) = a_{i,i+1}$, $i = 1, 2, \ldots, \mathbf{n}-1$.
	b	The right-hand-side vector b.
Output	**b**	The solution vector x overwrites the right-hand-side vector b.

Notes

These subprograms are usage-compatible with the standard LINPACK subprograms with the same names.

Caution is necessary since these subprograms do not detect error conditions. An inaccurate solution may be computed or a division by zero may occur if the matrix is indefinite or singular.

Example

Solve a system of linear equations $Ax = b$, where A is a 7-by-7 REAL*8 positive definite tridiagonal matrix. The principal diagonal of A is stored in array D, and the superdiagonal is stored in array E. b is a vector 7 elements long stored in an array B.

```
INTEGER*8 N
REAL*8    D(10),E(10),B(10)
N = 7
CALL SPTSL (N,D,E,B)
```

LINPACK Subprograms not in the *CONVEX SCILIB User's Guide*

Although SCILIB includes all LINPACK subprograms, the following nonoptimized subprograms are not documented in the *CONVEX SCILIB User's Guide*. The *LINPACK Users' Guide*, included in the SCILIB documentation set, documents these subprograms.

Table 4-5: LINPACK Subprograms not in the *SCILIB User's Guide*

Name	Function
SCHDC	Cholesky Decomposition of a Symmetric Matrix
CCHDC	Cholesky Decomposition of a Hermitian Matrix
SCHDD	Recompute the Cholesky Decomposition of a Downdated Symmetric Matrix
CCHDD	Recompute the Cholesky Decomposition of a Downdated Hermitian Matrix
SCHEX	Recompute the Cholesky Decomposition of a Permuted Symmetric Matrix
CCHEX	Recompute the Cholesky Decomposition of a Permuted Hermitian Matrix
SCHUD	Recompute the Cholesky Decomposition of a Updated Symmetric Matrix
CCHUD	Recompute the Cholesky Decomposition of a Updated Hermitian Matrix
CHICO	Factor a Hermitian Indefinite Matrix and Estimate its Condition Number
CHIDI	Determinant, Inverse, and Inertia of a Hermitian Indefinite Matrix
CHIFA	Factor a Hermitian Indefinite Matrix
CHISL	Solve Linear Equations with a Hermitian Indefinite Matrix
CHPCO	Factor a Hermitian Indefinite Packed Matrix and Estimate its Condition Number
CHPDI	Determinant, Inverse, and Inertia of a Hermitian Indefinite Packed Matrix
CHPFA	Factor a Hermitian Indefinite Packed Matrix
CHPSL	Solve Linear Equations with a Hermitian Indefinite Packed Matrix
SPPCO	Factor a Positive Definite Packed Matrix and Estimate its Condition Number
CPPCO	Factor a Positive Definite Packed Matrix and Estimate its Condition Number
SPPDI	Determinant and Inverse of a Positive Definite Packed Matrix
CPPDI	Determinant and Inverse of a Positive Definite Packed Matrix
SPPFA	Factor a Positive Definite Packed Matrix
CPPFA	Factor a Positive Definite Packed Matrix
SPPSL	Solve Linear Equations with a Positive Definite Packed Matrix
CPPSL	Solve Linear Equations with a Positive Definite Packed Matrix
SQRDC	*QR* Decomposition of a General Rectangular Matrix
CQRDC	*QR* Decomposition of a General Rectangular Matrix
SQRSL	Solve Linear Equations using the *QR* Decomposition
CQRSL	Solve Linear Equations using the *QR* Decomposition
SSICO	Factor a Symmetric Indefinite Matrix and Estimate its Condition Number
CSICO	Factor a Symmetric Indefinite Matrix and Estimate its Condition Number
SSIDI	Determinant, Inverse, and Inertia of a Symmetric Indefinite Matrix
CSIDI	Determinant, Inverse, and Inertia of a Symmetric Indefinite Matrix
SSIFA	Factor a Symmetric Indefinite Matrix
CSIFA	Factor a Symmetric Indefinite Matrix
SSISL	Solve Linear Equations with a Symmetric Indefinite Matrix
CSISL	Solve Linear Equations with a Symmetric Indefinite Matrix

Continued **LINPACK Subprograms not in the *SCILIB User's Guide***

Name	Function
SSPCO	Factor a Symmetric Indefinite Packed Matrix and Estimate its Condition Number
CSPCO	Factor a Symmetric Indefinite Packed Matrix and Estimate its Condition Number
SSPDI	Determinant, Inverse, and Inertia of a Symmetric Indefinite Packed Matrix
CSPDI	Determinant, Inverse, and Inertia of a Symmetric Indefinite Packed Matrix
SSPFA	Factor a Symmetric Indefinite Packed Matrix
CSPFA	Factor a Symmetric Indefinite Packed Matrix
SSPSL	Solve Linear Equations with a Symmetric Indefinite Packed Matrix
CSPSL	Solve Linear Equations with a Symmetric Indefinite Packed Matrix
SSVDC	Singular Value Decomposition of a General Rectangular Matrix
CSVDC	Singular Value Decomposition of a General Rectangular Matrix
STRCO	Estimate the Condition Number of a Triangular Matrix
CTRCO	Estimate the Condition Number of a Triangular Matrix
STRDI	Determinant and Inverse of a Triangular Matrix
CTRDI	Determinant and Inverse of a Triangular Matrix
STRSL	Solve Linear Equations with a Triangular Matrix
CTRSL	Solve Linear Equations with a Triangular Matrix

5

Eigenvalues and Eigenvectors

Overview

This chapter describes the EISPACK library included with SCILIB. Some subprograms in this library have been upgraded by incorporating Level 2 and Level 3 BLAS and other algorithmic improvements. Although all EISPACK subprograms are included in SCILIB, only upgraded ones are described in this chapter. Table 5-1 at the end of this chapter lists the subprograms that are included in SCILIB but not documented in the *CONVEX SCILIB User's Guide*. You may find information for these subprograms in the *EISPACK Guide* and the *EISPACK Guide Extension* included in the SCILIB documentation set.

The LAPACK software library included with SCILIB is a comprehensive collection of eigenvalue and eigenvector solvers and subprograms for other linear algebra computations. This software is documented in the *CONVEX LAPACK User's Guide*. We recommend that you use LAPACK subprograms rather than EISPACK subprograms in new programs. Future optimization efforts will be directed to LAPACK rather than EISPACK.

This chapter explains how to use SCILIB subprograms to compute eigenvalues or eigenvalues and eigenvectors of matrices. The operations covered are:

- dense Hermitian eigenproblems, $Ax = \lambda x$, with $A = A^*$
- dense general eigenproblems, $Ax = \lambda x$, for arbitrary A
- dense generalized eigenproblems, $Ax = \lambda Bx$
- banded eigenproblems, $Ax = \lambda x$

Refer to Chapter 7 for software to compute the eigenvalues or eigenvectors of a real, symmetric, sparse, ordinary or generalized eigenproblem.

Chapter Objectives

After reading this chapter you will:

- know which version of EISPACK is included in the SCILIB library
- know how to use the described subprograms

What You Need to Know to Use These Subprograms

EISPACK exists in single- and double-precision versions. Only the single-precision (64-bit) version is included in SCILIB.

Supplemental Reading

Garbow, B.S., *et al.* "Matrix Eigensystem Routines—EISPACK Guide Extension." *Lecture Notes in Computer Science*, Vol. 51. New York: Springer-Verlag. 1977.

Parlett, B.N. *The Symmetric Eigenproblem.* Englewood Cliffs, NJ: Prentice-Hall, Inc. 1980.

Smith, B.T., *et al.* "Matrix Eigensystem Routines—EISPACK Guide." *Lecture Notes in Computer Science*, Vol. 6, 2nd edition. New York: Springer-Verlag. 1976.

Wilkinson, J.H. *The Algebraic Eigenproblem.* New York: Oxford University Press. 1965.

Subprogram Descriptions

Eigenvalues and Eigenvectors of a Real Symmetric Matrix — RS

Purpose

This subprogram computes eigenvalues or eigenvalues and eigenvectors of a full real symmetric n-by-n matrix A. Specifically, given A, this subprogram determines n scalars, λ_i, $i = 1, 2, \ldots, n$, for which there exist corresponding nonzero vectors, x_i, such that

$$Ax_i = \lambda_i x_i.$$

Optionally, the x_i also may be computed.

Matrix Storage

Because the upper triangle of A may be obtained from the lower triangle, you need only provide the lower triangle of A, in a two-dimensional array large enough to hold the entire matrix. The upper triangle of the array is not referenced.

Usage

SCILIB:

```
INTEGER*8 ldax, n, job, ier
REAL*8    a(ldax, n), w(n), x(ldax, n), work1(n), work2(n)
CALL RS (ldax, n, a, w, job, x, work1, work2, ier)
```

Input

ldax The leading dimension of arrays **a** and **x** as declared in the calling program unit, with **ldax** ≥ max(**n**,1).

n The order of matrix A, **n** ≥ 0.

a Array containing the diagonal and lower triangle of the **n**-by-**n** matrix A. Elements in the strict upper triangle are not referenced.

job Option flag:

job = 0 compute eigenvalues only
job ≠ 0 compute eigenvalues and eigenvectors

Working storage

work1 Array of size **n**, used for work space.

work2 Array of size **n**, used for work space.

Output

a The lower triangle is destroyed if **job** = 0. Not modified if **job** ≠ 0.

w The eigenvalues λ_i of A in ascending order if **ier** = 0 is returned.

x Not referenced if eigenvectors are not requested. In this case, **x** can be a dummy variable. Otherwise, eigenvectors of A if **ier** = 0 is returned. The j-th column of **x** contains the eigenvector x_j of A corresponding to the eigenvalue in **w**(j), $j = 1, 2, \ldots,$ **n**. Eigenvectors are normalized to have Euclidean length = 1.

ier Status response:

ier = 0	Normal return.
ier = k, $1 \le k \le$ **n**	if calculation of the k-th eigenvalue failed to converge. **w**(1), **w**(2), ..., **w**(k-1) are eigenvalues, but are not necessarily the smallest and are not necessarily sorted. If eigenvectors are requested, the first k-1 columns of **x** are eigenvectors corresponding to the first k-1 elements of **w**.
ier = 10**n**	if **n** > **ldax**. No eigenvalues or eigenvectors are returned.

RS — Continued

Notes

This subprogram is usage-compatible with the standard single-precision EISPACK subprogram with the same name. It calls EISPACK subprograms TRED1 and TQLRAT or TRED2 and TQL2, which are documented elsewhere in this chapter.

Example 1

Compute eigenvalues of a 6-by-6 REAL*8 symmetric matrix A whose diagonal and lower triangle are stored in array A whose dimensions are 10 by 10. Eigenvalues are stored in array W of dimension 10.

```
INTEGER*8 LDA,N,JOB,IER
REAL*8    A(10,10),W(10),X,WORK1(10),WORK2(10)
LDA = 10
N = 6
JOB = 0
CALL RS (LDA,N,A,W,JOB,X,WORK1,WORK2,IER)
IF ( IER .NE. 0 ) THEN
  handle convergence failure
END IF
```

Example 2

Compute eigenvalues and eigenvectors of a 6-by-6 REAL*8 symmetric matrix A whose diagonal and lower triangle are stored in array A whose dimensions are 10 by 10. Eigenvalues are stored in array W of dimension 10; eigenvectors are stored in the first six columns of array X of dimension 10 by 10.

```
INTEGER*8 LDAX,N,JOB,IER
REAL*8    A(10,10),W(10),X(10,10),WORK1(10),WORK2(10)
LDAX = 10
N = 6
JOB = 1
CALL RS (LDAX,N,A,W,JOB,X,WORK1,WORK2,IER)
IF ( IER .NE. 0 ) THEN
  handle convergence failure
END IF
```

Eigenvalues and Eigenvectors of a Real Symmetric Matrix — TQL2

Purpose

This subprogram computes eigenvalues and eigenvectors of a tridiagonal real symmetric n-by-n matrix. Eigenvalues and eigenvectors of a full real symmetric matrix can also be computed by this subprogram if TRED2 has been used to reduce the full matrix to tridiagonal form.

Specifically, given a tridiagonal real symmetric matrix A or output of TRED2 applied to a full real symmetric matrix A, this subprogram determines scalars, λ_i, $i = 1, 2, \ldots, n$, and nonzero vectors, x_i, $i = 1, 2, \ldots, n$, such that

$$Ax_i = \lambda_i x_i.$$

Matrix Storage

The following example illustrates the storage of symmetric tridiagonal matrices. Consider the following symmetric tridiagonal matrix of order $n = 7$:

11	21	0	0	0	0	0
21	22	32	0	0	0	0
0	32	33	43	0	0	0
0	0	43	44	54	0	0
0	0	0	54	55	65	0
0	0	0	0	65	66	76
0	0	0	0	0	76	77

The subdiagonal is stored in array **e**, and the principal diagonal is stored in array **d**, as follows:

i	e(i)	d(i)
1	*	11
2	21	22
3	32	33
4	43	44
5	54	55
6	65	66
7	76	77

The asterisk represents an element whose initial contents are not used.

Usage

SCILIB:

```
INTEGER*8 ldx, n, ier
REAL*8    d(n), e(n), x(ldx, n)
CALL TQL2 (ldx, n, d, e, x, ier)
```

Input

ldx The leading dimension of array **x** as declared in the calling program unit, with **ldx** ≥ max(**n**,1).

n The order of matrix A, **n** ≥ 0.

d Array containing the diagonal elements of the **n**-by-**n** symmetric tridiagonal matrix A.

e Array containing the subdiagonal elements of A in elements **e**(2) through **e**(**n**). **e**(1) is not used as input.

x If A is a tridiagonal matrix, **x** must be initialized to the **n**-by-**n** identity matrix. If A is a full matrix, **x** contains the transformation matrix produced by TRED2 in reducing the full matrix to tridiagonal form.

TQL2 — Continued

Output

d Eigenvalues, λ_i, $i = 1, 2, ..., n$, of A overwrite the input if **ier** = 0 is returned. Eigenvalues have been sorted into ascending order.

e Destroyed.

x Eigenvectors of A if **ier** = 0 is returned. The j-th column of **x** is the eigenvector x_j of A, corresponding to the eigenvalue in **d**(j), $j = 1, 2, ...,$ **n**. Eigenvectors are normalized to have Euclidean length = 1.

ier Status response:

ier = 0	Normal return.
ier = k, $1 \le k \le$ **n**	if calculation of the k-th eigenvalue failed to converge. **d**(1), **d**(2), ..., **d**(k-1) are eigenvalues, but are not necessarily the smallest and are not necessarily sorted. The first k-1 columns of **x** are eigenvectors corresponding to the first k-1 elements of **d**.

Notes This subprogram is usage compatible with the standard single-precision EISPACK subprogram with the same name.

Example 1 Compute eigenvalues and eigenvectors of a 6-by-6 tridiagonal REAL*8 symmetric matrix A whose diagonal and lower subdiagonal are stored in arrays D and E of dimension 10. Eigenvalues are returned in array D; eigenvectors are placed in the first six columns of array X of dimension 10 by 10.

```
INTEGER*8 LDX,N,IER
REAL*8    D(10),E(10),X(10,10)
LDX = 10
N = 6
DO J = 1, N
  DO I = 1, N
    X(I,J) = 0.0
  END DO
  X(J,J) = 1.0
END DO
CALL TQL2 (LDX,N,D,E,X,IER)
IF ( IER .NE. 0 ) THEN
  handle convergence failure
END IF
```

Example 2 Compute eigenvalues and eigenvectors of a 6-by-6 REAL*8 symmetric matrix A whose diagonal and lower triangle are stored in array A whose dimensions are 10 by 10. Eigenvalues are stored in array W of dimension 10; eigenvectors are stored in the first six columns of array X of dimension 10 by 10. (Compare with "Example 2" in the description of RS.)

```
INTEGER*8 LDAX,N,IER
REAL*8    A(10,10),W(10),X(10,10),WORK(10)
LDAX = 10
N = 6
CALL TRED2 (LDAX,N,A,W,WORK,X)
CALL TQL2 (LDAX,N,W,WORK,X,IER)
IF ( IER .NE. 0 ) THEN
  handle convergence failure
END IF
```

Eigenvalues of a Real Symmetric Matrix — TQLRAT

Purpose

This subprogram computes the eigenvalues of a tridiagonal real symmetric n-by-n matrix. Eigenvalues of a full real symmetric matrix can also be computed by this subprogram if TRED1 has been used to reduce the full matrix to tridiagonal form.

Specifically, given a tridiagonal real symmetric matrix A or output of TRED1 applied to a full real symmetric matrix A, this subprogram determines scalars, λ_i, $i = 1, 2, \ldots, n$, for which there exist corresponding nonzero vectors, x_i, $i = 1, 2, \ldots, n$, such that

$$Ax_i = \lambda_i x_i.$$

Matrix Storage

The following example illustrates the storage of symmetric tridiagonal matrices. Consider the following symmetric tridiagonal matrix of order $n = 7$:

11	21	0	0	0	0	0
21	22	32	0	0	0	0
0	32	33	43	0	0	0
0	0	43	44	54	0	0
0	0	0	54	55	65	0
0	0	0	0	65	66	76
0	0	0	0	0	76	77

The squares of the subdiagonal elements are stored in array **e2**, and the principal diagonal is stored in array **d**, as follows:

i	**e2**(i)	**d**(i)
1	*	11
2	21^2	22
3	32^2	33
4	43^2	44
5	54^2	55
6	65^2	66
7	76^2	77

The asterisk represents an element whose initial contents are not used.

Usage

SCILIB:

```
INTEGER*8 n, ier
REAL*8    d(n), e2(n)
CALL TQLRAT (n, d, e2, ier)
```

Input

n The order of matrix A, **n** ≥ 0.

d Array containing diagonal elements of the **n**-by-**n** symmetric tridiagonal matrix A.

e2 Array containing squares of subdiagonal elements of A in elements **e2**(2) through **e2**(**n**). **e2**(1) is not used as input.

Output

d Eigenvalues, λ_i, $i = 1, 2, \ldots, n$, of A overwrite the input if **ier** = 0 is returned. Eigenvalues have been sorted into ascending order. ρ

e2 Destroyed.

ier Status response:

ier = 0	Normal return.
ier = $k \neq 0$	If calculation of the k-th eigenvalue failed to converge. **d**(1), **d**(2), ..., **d**(k-1) are eigenvalues, but are not necessarily the smallest ones.

Notes This subprogram is usage-compatible with the standard single-precision EISPACK subprogram with the same name.

Example 1 Compute eigenvalues of a 6-by-6 tridiagonal REAL*8 symmetric matrix A whose diagonal is stored in array D of dimension 10. Squares of the lower subdiagonal elements of A are stored in array E2, also of dimension 10. The eigenvalues are returned in array D.

```
INTEGER*8 N,IER
REAL*8    D(10),E2(10)
N = 6
CALL TQLRAT (N,D,E2,IER)
IF ( IER .NE. 0 ) THEN
  handle convergence failure
END IF
```

Example 2 Compute eigenvalues of a 6-by-6 REAL*8 symmetric matrix A whose diagonal and lower triangle are stored in array A whose dimensions are 10 by 10. Eigenvalues are stored in array W of dimension 10. (Compare with "Example 1" in the description of RS.)

```
INTEGER*8 LDA,N,IER
REAL*8    A(10,10),W(10),WORK1(10),WORK2(10)
LDA = 10
N = 6
CALL TRED1 (LDA,N,A,W,WORK1,WORK2)
CALL TQLRAT (N,W,WORK2,IER)
IF ( IER .NE. 0 ) THEN
  handle convergence failure
END IF
```

Reduce Real Symmetric Matrix to Tridiagonal Form — TRED1

Purpose

This subprogram uses orthogonal-similarity transformations to reduce a full real symmetric n-by-n matrix A to symmetric tridiagonal form without accumulating reduction transformations. The reduced form may be passed to subprogram TQLRAT, documented elsewhere in this chapter, to find the eigenvalues of A.

Specifically, given A, this subprogram determines an n-by-n tridiagonal matrix T that is orthogonally similar to A, i.e., such that there exists an n-by-n orthogonal matrix Q for which

$$Q^T A Q = T.$$

Matrix Storage

Because the upper triangle of A may be obtained from the lower triangle, you need only provide the lower triangle of A, in a two-dimensional array large enough to hold the entire matrix. The upper triangle of the array is not referenced.

Usage

SCILIB:

```
INTEGER*8 lda, n
REAL*8    a(lda, n), d(n), e(n), e2(n)
CALL TRED1 (lda, n, a, d, e, e2)
```

Input

lda The leading dimension of array **a** as declared in the calling program unit, with **lda** ≥ max(**n**,1).

n The order of matrix A, **n** ≥ 0.

a Array containing the diagonal and lower triangle of the **n**-by-**n** matrix A. Elements in the strict upper triangle are not referenced.

Output

a The diagonal and lower triangle are destroyed.

d Array containing diagonal elements of the tridiagonal matrix T.

e Array containing the subdiagonal elements of T in elements **e**(2) through **e**(**n**). **e**(1) = 0.

e2 Array containing squares of subdiagonal elements of T in elements **e**(2) through **e**(**n**). **e2**(1) = 0.

Notes

This subprogram is usage-compatible with the standard single-precision EISPACK subprogram with the same name.

Output arrays **e** and **e2** are redundant. Some EISPACK subprograms that can be used following TRED1 require **e** as input and some require **e2**.

Example 1

Reduce the 6-by-6 REAL*8 symmetric matrix A whose diagonal and lower triangle are stored in array **A** whose dimensions are 10 by 10 to tridiagonal form.

```
INTEGER*8 LDA,N
REAL*8    A(10,10),D(10),E(10),E2(10)
LDA = 10
N = 6
CALL TRED1 (LDA,N,A,D,E,E2)
```

TRED1 Continued

Example 2 Compute eigenvalues of a 6-by-6 REAL*8 symmetric matrix A whose diagonal and lower triangle are stored in array A whose dimensions are 10 by 10. Eigenvalues will be stored in array W of dimension 10. (Compare with "Example 1" in the description of RS.)

```
INTEGER*8 LDA,N,IER
REAL*8    A(10,10),W(10),WORK1(10),WORK2(10)
LDA = 10
N = 6
CALL TRED1 (LDA,N,A,W,WORK1,WORK2)
CALL TQLRAT (N,W,WORK2,IER)
IF ( IER .NE. 0 ) THEN
  handle convergence failure
END IF
```

Reduce Real Symmetric Matrix to Tridiagonal Form — TRED2

Purpose

This subprogram uses orthogonal similarity transformations to reduce a full real symmetric n-by-n matrix A to symmetric tridiagonal form and accumulates reduction transformations. This reduced form and the transformation matrix may be passed to subprogram TQL2, documented elsewhere in this chapter, to find eigenvalues and eigenvectors of A.

Specifically, given A, this subprogram determines an n-by-n orthogonal matrix X and an n-by-n symmetric tridiagonal matrix T such that

$$X^T AX = T.$$

Matrix Storage

Because the upper triangle of A may be obtained from the lower triangle, you need only provide the lower triangle of A, in a two-dimensional array large enough to hold the entire matrix. The upper triangle of the array is not referenced.

Usage

SCILIB:

```
INTEGER*8 ldax, n
REAL*8    a(ldax, n), d(n), e(n), x(ldax, n)
CALL TRED2 (ldax, n, a, d, e, x)
```

Input

ldax The leading dimension of arrays **a** and **x** as declared in the calling program unit, with **ldax** ≥ max(**n**,1).

n The order of matrix A, **n** ≥ 0.

a Array containing the diagonal and lower triangle of the **n**-by-**n** matrix A. Elements in the strict upper triangle are not referenced.

Output

d Array containing diagonal elements of the tridiagonal matrix T.

e Array containing subdiagonal elements of T in elements **e**(2) through **e**(**n**). **e**(1) = 0.

x The transformation matrix X that reduces A to tridiagonal form.

Notes

This subprogram is usage-compatible with the standard single-precision EISPACK subprogram with the same name.

Example 1

Reduce the 6-by-6 REAL*8 symmetric matrix A whose diagonal and lower triangle are stored in array A whose dimensions are 10 by 10 to tridiagonal form and accumulate the transformation matrix.

```
INTEGER*8 LDAX,N
REAL*8    A(10,10),D(10),E(10),X(10,10)
LDAX = 10
N = 6
CALL TRED2 (LDAX,N,A,D,E,X)
```

TRED2 Continued

Example 2 Compute eigenvalues and eigenvectors of a 6-by-6 REAL*8 symmetric matrix *A* whose diagonal and lower triangle are stored in array A whose dimensions are 10 by 10. Eigenvalues will be stored in array W of dimension 10; eigenvectors will be stored in the first six columns of array X of dimension 10 by 10. (Compare with "Example 2" in the description of RS.)

```
INTEGER*8 LDAX,N,IER
REAL*8    A(10,10),W(10),X(10,10),WORK(10)
LDAX = 10
N = 6
CALL TRED2 (LDAX,N,A,W,WORK,X)
CALL TQL2 (LDAX,N,W,WORK,X,IER)
IF ( IER .NE. 0 ) THEN
  handle convergence failure
END IF
```

EISPACK Subprograms not in the *CONVEX SCILIB User's Guide*

Although SCILIB includes all EISPACK subprograms, the following nonoptimized routines are not documented in the *CONVEX SCILIB User's Guide*. The *EISPACK Guide* and the *EISPACK Guide Extension*, included in the SCILIB documentation set, document these subprograms.

Table 5-1: EISPACK Subprograms not in the *SCILIB User's Guide*

Name	Function
BAKVEC	Back Transform Eigenvectors following FIGI
BALANC	Balance a Real General Matrix
BALBAK	Back Transform Eigenvectors following BALANC
BANDR	Reduce a Real Symmetric Band Matrix to Real Symmetric Tridiagonal Form
BANDV	Determine Some Eigenvectors of a Real Symmetric Band Matrix
BISECT	Determine Some Eigenvectors of a Real Symmetric Tridiagonal Matrix
BQR	Determine Some Eigenvalues of a Real Symmetric Band Matrix
CBABK2	Back Transform Eigenvectors following CBAL
CBAL	Balance a Complex General Matrix
CG	Determine Eigenvalues/vectors of a Complex General Matrix
CH	Determine Eigenvalues/vectors of a Complex Hermitian Matrix
CINVIT	Determine Some Eigenvectors of a Complex Upper Hessenberg Matrix
COMBAK	Back Transform Eigenvectors following COMHES
COMHES	Reduce a Complex General Matrix to Complex Upper Hessenberg Form
COMLR	Determine the Eigenvalues of a Complex Upper Hessenberg Matrix
COMLR2	Determine the Eigenvalues/vectors of a Complex Hessenberg Matrix
COMQR	Determine the Eigenvalues of a Complex Upper Hessenberg Matrix
COMQR2	Determine the Eigenvalues/vectors of a Complex Upper Hessenberg Matrix
CORTB	Back Transform Eigenvectors following CORTH
CORTH	Reduce a Complex General Matrix to Complex Upper Hessenberg Form
ELMBAK	Back Transform Eigenvectors following ELMHES
ELMHES	Reduce a Real General Matrix to Real Upper Hessenberg Form
ELTRAN	Accumulate the Transformations in the Reduction by ELMHES
FIGI	Transform a Real Non-symmetric Tridiagonal Matrix to Real Symmetric Form
FIGI2	Transform a Real Non-symmetric Tridiagonal Matrix to Real Symmetric Form
HQR	Determine the Eigenvalues of a Real Upper Hessenberg Matrix
HQR2	Determine the Eigenvalues/vectors of a Real Upper Hessenberg Matrix
HTRIB3	Back Transform Eigenvectors following HTRID3
HTRIBK	Back Transform Eigenvectors following HTRIDI
HTRID3	Reduce a Complex Hermitian Matrix to Real Symmetric Tridiagonal Form
HTRIDI	Reduce a Complex Hermitian Matrix to Real Symmetric Tridiagonal Form
IMTQL1	Determine the Eigenvalues of a Real Symmetric Tridiagonal Matrix
IMTQL2	Determine the Eigenvalues/vectors of a Real Symmetric Tridiagonal Matrix
IMTQLV	Determine the Eigenvalues of a Real Symmetric Tridiagonal Matrix
INVIT	Determine Some Eigenvectors of a Real Upper Hessenberg Matrix
MINFIT	Solve a Least Squares Problem with a Real Rectangular Coefficient Matrix

EISPACK Subprograms not in the *CONVEX SCILIB User's Guide* Continued

Name	Function
ORTBAK	Back Transform Eigenvectors following ORTHES
ORTHES	Reduce a Real General Matrix to Real Upper Hessenberg Form
ORTRAN	Accumulate the Transformations in the Reduction by ORTHES
QZHES	Partially Reduce a Real General Generalized Eigenproblem
QZIT	Complete the Reduction of a Real General Generalized Eigenproblem
QZVAL	Determine the Eigenvalues of a Reduced Real General Generalized Eigenproblem
QZVEC	Determine the Eigenvectors of a Reduced Real General Generalized Eigenproblem
RATQR	Determine Some Extreme Eigenvalues of a Real Symmetric Tridiagonal Matrix
REBAK	Back Transform Eigenvectors following REDUC or REDUC2
REBAKB	Back Transform Eigenvectors following REDUC2
REDUC	Reduce a Real Symmetric Generalized Eigenproblem to Standard Form
REDUC2	Reduce a Real Symmetric Generalized Eigenproblem to Standard Form
RG	Determine the Eigenvalues/vectors of a Real General Matrix
RGG	Determine the Eigenvalues/vectors of a Real General Generalized Eigenproblem
RSB	Determine the Eigenvalues/vectors of a Real Symmetric Band Matrix
RSG	Determine the Eigenvalues/vectors of a Real Symmetric Generalized Eigenproblem
RSGAB	Determine the Eigenvalues/vectors of a Real Symmetric Generalized Eigenproblem
RSGBA	Determine the Eigenvalues/vectors of a Real Symmetric Generalized Eigenproblem
RSM	Determine All Eigenvalues and Some Eigenvectors of a Real Symmetric Matrix
RSP	Determine the Eigenvalues/vectors of a Real Symmetric Packed Matrix
RST	Determine the Eigenvalues/vectors of a Real Symmetric Tridiagonal Matrix
RT	Determine the Eigenvalues/vectors of a Real Tridiagonal Matrix
SVD	Compute the Singular Value Decomposition of a Real Rectangular Matrix
TINVIT	Determine Some Eigenvectors of a Real Symmetric Tridiagonal Matrix
TQL1	Determine the Eigenvalues of a Real Symmetric Tridiagonal Matrix
TRBAK1	Back Transform Eigenvectors following TRED1
TRBAK3	Back Transform Eigenvectors following TRED3
TRED3	Reduce a Real Symmetric Matrix to Real Symmetric Tridiagonal Form
TRIDIB	Determine Some Eigenvalues of a Real Symmetric Tridiagonal Matrix
TSTURM	Determine Some Eigenvalues/vectors of a Real Symmetric Tridiagonal Matrix

6

Fast Fourier Transforms

Overview

This chapter explains how to use the SCILIB Fast Fourier Transform (FFT) subprograms. The operations covered are

- one-dimensional complex-to-complex FFT subprograms
- one-dimensional real-to-complex and complex-to-real FFT subprograms
- one-dimensional simultaneous complex-to-complex FFT subprograms
- one-dimensional simultaneous real-to-complex and complex-to-real FFT subprograms

Chapter Objectives

After reading this chapter you will

- understand the SCILIB FFT subprogram restrictions
- know how to augment subprograms with zero-value data points
- know how to use the described subprograms

What You Need to Know to Use These Subprograms

Strictly speaking, an FFT is not a type of transform but a class of algorithms for efficiently computing the discrete Fourier transform (DFT). Although the DFT is defined for any number of data points, the SCILIB FFT subprograms restrict the number of points to certain forms. For single one-dimensional transforms, the number of points must be a power of two:

$$l = 2^p, \quad p \geq 0.$$

For simultaneous transforms, the number of points in each direction must be a product of powers of two, three, and five:

$$l = 2^p \, 3^q \, 5^r, \quad p,q,r \geq 0.$$

While these restrictions limit the utility of the subprograms, the gain in speed is enormous. You can frequently adapt your data set to the SCILIB FFT subprograms by augmenting it with enough zero-value data points to reach the next acceptable number of points. Doing so slightly changes the problem, which may or may not be important, depending on the problem. For example, adding zero-value points to a time series changes the implied sampling frequency, but adding zero-value points to data sets before using FFT subprograms to compute convolutions does not change the result.

Supplemental Reading

Brigham, E.O. *The Fast Fourier Transform.* Englewood Cliffs, NJ: Prentice-Hall, Inc. 1974.

Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, Inc. 1975.

Subprogram Descriptions

One-Dimensional Complex-to-Complex FFT — CFFT2

Purpose

Given an array of complex data, this subprogram computes the one-dimensional unscaled forward or inverse discrete Fourier transform using a radix 2 fast Fourier transform (FFT) algorithm.

The one-dimensional unscaled forward discrete Fourier transform of $z(n)$, for $n = 1, 2, ..., l$, is defined by

$$Z(m) = \sum_{n=1}^{l} z(n)\, e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, ..., l$ and $i = \sqrt{-1}$.

Alternatively, the one-dimensional unscaled inverse discrete Fourier transform of $Z(m)$, for $m = 1, 2, ..., l$, is defined by

$$z(n) = \sum_{m=1}^{l} Z(m)\, e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, ..., l$.

CFFT2 requires that l be a power of 2, i.e., of the form $l = 2^p$, where $p \geq 0$.

Usage

Because it is common to use one data set length repetitively, this subprogram has a separate initialization call so that the setup can be performed only once for each different transform size. You will, therefore, always have at least two **CALL** statements to the FFT subprogram, using the same working storage array. Refer to "Example."

SCILIB:

```
INTEGER*8    init, isgn, l
COMPLEX*16   zin(l), zout(l)
REAL*8       work(5*l)
CALL CFFT2 (init, isgn, l, zin, work, zout)
```

Input

init Initialization flag:

init ≠ 0 initialize **work** for subsequent transforms of length **l**.
init = 0 compute transform.

isgn Operation flag: if **init** = 0,

isgn < 0 compute unscaled forward transform.
isgn > 0 compute unscaled inverse transform.

The sign of the exponential is the same as the sign of **isgn**.

l Number of data points, of the form $\mathbf{l} = 2^p$, with $p \geq 0$.

zin Array of data to be transformed. Not used if **init** ≠ 0. **zin** may be equivalenced to **work**, in which case the input values may be overwritten.

Working Storage

work If **init** ≠ 0, **work** is initialized for computing transforms of length **l**.

If **init** = 0, **work** must have been initialized by a previous call with this value of **l** in which **init** ≠ 0.

CFFT2 Continued

Output **zout** Array of transformed data if **init** = 0. Not used if **init** ≠ 0.

Notes It is usual to scale the inverse transform by multiplying the summation by 1/l so that the inverse transform of the forward transform of a data set returns the original data. This subprogram omits this scaling, meaning that the inverse transform of the forward transform of a data set is the original data multiplied by l.

If an error in the arguments is detected, the subprograms call error handler XERSCI, which writes an error message onto the standard error file and terminates execution. The standard version of XERSCI (refer to Chapter 9, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are:

init = 0 and **isgn** = 0;
l not a power of 2.

Example Compute the forward discrete Fourier transform of two COMPLEX*16 data sets of length 1024. The length of working storage is 5*1024 = 5120.

```
INTEGER*8  INIT,ISGN,L
COMPLEX*16 ZIN1(1024),ZOUT1(1024),ZIN2(1024),ZOUT2(1024)
REAL*8     WORK(5120)
L = 1024
INIT = 1
CALL CFFT2 (INIT,ISGN,L,ZIN1,WORK,ZOUT1) ! INITIALIZE
INIT = 0
ISGN = -1
CALL CFFT2 (INIT,ISGN,L,ZIN1,WORK,ZOUT1) ! FIRST TRANSFORM
CALL CFFT2 (INIT,ISGN,L,ZIN2,WORK,ZOUT2) ! SECOND TRANSFORM
```

Simultaneous One-Dimensional FFT — CFTFAX/CFFTMLT

Purpose

Given a number of sets of one-dimensional complex data with real and imaginary parts in separate real arrays, subroutine CFFTMLT computes all of their one-dimensional forward or inverse discrete Fourier transforms using a radix 2-3-5 fast Fourier transform (FFT) algorithm.

The one-dimensional forward discrete Fourier transform of a complex set of data $z(n)$, for $n = 1, 2, \ldots, l$, is defined by

$$Z(m) = \sum_{n=1}^{l} z(n)\, e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, \ldots, l$ and $i = \sqrt{-1}$.

Alternatively, the one-dimensional scaled inverse discrete Fourier transform of $Z(m)$, for $m = 1, 2, \ldots, l$, is defined by

$$z(n) = \frac{1}{l} \sum_{m=1}^{l} Z(m)\, e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, \ldots, l$.

This subprogram performs forward or scaled inverse transform operations simultaneously on a number of data sets. It requires that the length l of the data sets be a product of powers of 2, 3, and 5, i.e., of the form

$$l = 2^p\ 3^q\ 5^r,$$

where $p, q, r \geq 0$. Refer to "Notes" for a partial list of permissible values of l.

The complex data, z or Z, are stored with real and imaginary parts in separate real arrays, **x** and **y**, respectively.

Usage

Because it is common to use one data set length repetitively, subroutine CFFTMLT has a separate initialization subprogram, CFTFAX, so that the setup can be performed only once for each different transform size. You will, therefore, always have at least one **CALL CFTFAX** statement and at least one **CALL CFFTMLT** statement, using the same working storage arrays. Refer to "Example."

SCILIB:

```
INTEGER*8 work3(19), incl, incn, l, n, isgn
REAL*8    x(lenxy), y(lenxy), work1(4*l*n), work2(2*l)
```

Initialization call:

```
CALL CFTFAX (l, work3, work2)
```

Transform call:

```
CALL CFFTMLT (x, y, work1, work2, work3, incl, incn, l, n, isgn)
```

CFTFAX/CFFTMLT — Continued

Input

x and y Arrays containing **n** data sets, each consisting of **l** data points, to be transformed. Typically, **x** and **y** will be two- or three-dimensional arrays with each data set being a one-dimensional array section. Refer to "Notes" for suggested usages.

Treating **x** and **y** as one-dimensional arrays results in

$$\mathbf{lenxy} = (\mathbf{l}-1)\times\mathbf{incl} + (\mathbf{n}-1)\times\mathbf{incn} + 1.$$

The real and imaginary parts of the i-th data point of the j-th data set, $1 \le i \le \mathbf{l}$, $1 \le j \le \mathbf{n}$, are stored in

$$\mathbf{x}((i-1)\times\mathbf{incl} + (j-1)\times\mathbf{incn} + 1)$$

and

$$\mathbf{y}((i-1)\times\mathbf{incl} + (j-1)\times\mathbf{incn} + 1),$$

respectively.

incl Storage increment between successive elements of the same data set, **incl** $>$ 0. Use **incl** $= 1$ if each data set is stored contiguously in **x** and **y**.

incn Storage increment between corresponding data points of successive data sets, **incn** > 0.

l Number of data points in each data set, of the form $\mathbf{l} = 2^p\ 3^q\ 5^r$, with $p,q,r \ge 0$.

n The number of data sets, **n** > 0.

isgn Operation flag:

isgn $=$ -1 compute forward transform.
isgn $=$ +1 compute inverse transform.

The sign of the exponential is the same as the sign of **isgn**.

Working Storage

work1 Array used for work space.

work2 Array, initialized by CFTFAX for use as work space in CFFTMLT.

work3 Array, initialized by CFTFAX for use as work space in CFFTMLT. CFTFAX returns **work3(1)** $= -99$ if **l** is not factorable as specified above.

Output

x and y The transformed data replaces the input.

Notes

Typically, **x** and **y** will be two- or three-dimensional arrays with each data set being a one-dimensional section of the arrays, i.e., all but one subscript will be constant within a data set.

If **x** and **y** are two-dimensional arrays of dimension **ldxy** by **mdxy**, and if the data sets are stored in the columns of **x** and **y**, then **l** ≤ **ldxy**, **n** ≤ **mdxy**, **incl** = 1, and **incn** = **ldxy**. For example:

CALL CFFTMLT (x, y, work1, work2, work3, 1, ldxy, l, n, isgn)

If **x** and **y** are two-dimensional arrays as above and data sets are stored in rows of **x** and **y**, **l** ≤ **mdxy**, **n** ≤ **ldxy**, **incl** = **ldxy**, and **incn** = 1. For example:

CALL CFFTMLT (x, y, work1, work2, work3, ldxy, 1, l, n, isgn)

The subprograms generally are faster if the data sets are the rows of the arrays, so that **incn** = 1, rather than the columns. Otherwise, it is generally better to have an odd value of **incn**.

If **x** and **y** are three-dimensional arrays of dimension **ldxy** by **mdxy**-by-**ndxy**, then **incl** and **incn** will usually be 1, **ldxy**, or **ldxy**×**mdxy**, depending on which of the subscripts of the three-dimensional array varies within a data set, which subscript varies between data sets, and which remains constant. Specifically, if the subscript that varies within a data set is the

1st subscript, use **incl** = 1.
2nd subscript, use **incl** = **ldxy**.
3rd subscript, use **incl** = **ldxy**×**mdxy**.

Similarly, if the subscript that varies between data sets is the

1st subscript, use **incn** = 1.
2nd subscript, use **incn** = **ldxy**.
3rd subscript, use **incn** = **ldxy**×**mdxy**.

If an error in the arguments is detected, the subprograms call error handler XERSCI, which writes an error message onto the standard error file and terminates execution. The standard version of XERSCI (refer to Chapter 9, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are

work3(1) = –99,
incl ≤ 0,
incn ≤ 0,
l not of the form $2^p\ 3^q\ 5^r$ for $p,q,r \geq 0$,
n < 0, and
isgn ≠ ± 1.

The following list indicates some of the permissible values of **l**. Although **l** can be any value of the form $2^p\ 3^q\ 5^r$ where $p,q,r \geq 0$, this list only shows values not exceeding 1000:

1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, 32, 36, 40, 45, 48, 50, 54, 60, 64, 72, 75, 80, 81, 90, 96, 100, 108, 120, 125, 128, 135, 144, 150, 160, 162, 180, 192, 200, 216, 225, 240, 243, 250, 256, 270, 288, 300, 320, 324, 360, 375, 384, 400, 405, 432, 450, 480, 486, 500, 512, 540, 576, 600, 625, 640, 648, 675, 720, 729, 750, 768, 800, 810, 864, 900, 960, 972, and 1000.

CFTFAX/CFFTMLT Continued

Example 1 Compute the forward discrete Fourier transform of 256 complex data sets of length 1024. Real and imaginary parts of data sets are stored as columns of arrays X and Y whose dimensions are 1025 by 256.

```
INTEGER*8 WORK3(19),INCL,INCN,L,N,ISGN
REAL*8    X(1025,256),Y(1025,256),WORK1(1048576),WORK2(512)
L = 1024
INCL = 1
N = 256
INCN = 1025
ISGN = -1
CALL CFTFAX (L,WORK3,WORK2)
IF ( WORK3(1) .EQ. -99 ) THEN
  handle error condition
END IF
CALL CFFTMLT (X,Y,WORK1,WORK2,WORK3,INCL,INCN,L,N,ISGN)
```

Example 2 Compute the inverse discrete Fourier transform of 1024 complex data sets of length 256. Real and imaginary parts of data sets are stored as rows of arrays X and Y whose dimensions are 1025 by 256.

```
INTEGER*8 WORK3(19),INCL,INCN,L,N,ISGN
REAL*8    X(1025,256),Y(1025,256),WORK1(1048576),WORK2(512)
L = 256
INCL = 1025
N = 1024
INCN = 1
ISGN = 1
CALL CFTFAX (L,WORK3,WORK2)
IF ( WORK3(1) .EQ. -99 ) THEN
  handle error condition
END IF
CALL CFFTMLT (X,Y,WORK1,WORK2,WORK3,INCL,INCN,L,N,ISGN)
```

Complex-to-Real One-Dimensional FFT — CRFFT2

Purpose

A complex sequence $Z(m)$, for $m = 1, 2, ..., l$, is conjugate-symmetric about $Z(l/2+1)$ if

$$Im(Z(1)) = Im(Z(l/2 + 1)) = 0$$

and

$$Z(l/2+1 + m) = \overline{Z}(l/2+1 - m), \quad m = 1, 2, ..., l/2-1,$$

where $\overline{Z}$ is the complex conjugate of Z.

Given an array of conjugate-symmetric complex data, this subprogram computes the one-dimensional, complex-to-real, unscaled forward or unscaled inverse, discrete Fourier transform using a radix 2 fast Fourier transform (FFT) algorithm optimized for real output.

The one-dimensional unscaled forward discrete Fourier transform of a data set, $z(n)$, for $n = 1, 2, ..., l$, is defined by

$$Z(m) = \sum_{n=1}^{l} z(n)\, e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, ..., l$ and $i = \sqrt{-1}$.

Alternatively, the one-dimensional unscaled inverse discrete Fourier transform of the data set $Z(m)$, for $m = 1, 2, ..., l$, is defined by

$$z(n) = \sum_{m=1}^{l} Z(m)\, e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, ..., l$.

This subprogram requires that l be a power of 2, i.e., of the form $l = 2^p$, where $p \geq 0$.

Usage

Because it is common to use one data set length repetitively, this subprogram has a separate initialization call so that the setup can be performed only once for each different transform size. Therefore, you will always have at least two **CALL** statements to the FFT subprogram, using the same working storage array.

SCILIB:

```
INTEGER*8    init, isgn, l
COMPLEX*16   zin(l/2+1)
REAL*8       work(3*l+4), xout(l)
CALL CRFFT2 (init, isgn, l, zin, work, xout)
```

Input

init Initialization flag:

init $\neq$ 0 initialize **work** for subsequent transforms of length l.
init = 0 compute transform.

isgn Operation flag: if **init** = 0,

isgn < 0 compute forward transform.
isgn > 0 compute unscaled inverse transform.

The sign of the exponential is the same as the sign of **isgn**.

CRFFT2 Continued

	l	Number of data points, of the form $\mathbf{l} = 2^p$, with $p \geq 0$.
	zin	Array containing the first **l**/2+1 elements of the data set to be transformed. Not used if **init** ≠ 0.
Working Storage	**work**	If **init** ≠ 0, **work** is initialized for computing transforms of length **l**. If **init** = 0, **work** must have been initialized by a previous call with this value of **l** in which **init** ≠ 0.
Output	**xout**	Array of transformed data if **init** = 0. Not used if **init** ≠ 0.

Notes

It is usual to scale the inverse transform by multiplying the summation by 1/**l** so that the inverse transform of the forward transform of a data set returns the original data. This subprogram omits this scaling, meaning that the inverse transform of the forward transform of a data set is the original data multiplied by **l**.

If an error in the arguments is detected, the subprograms call error handler XERSCI, which writes an error message onto the standard error file and terminates execution. The standard version of XERSCI (refer to Chapter 9, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are:

init = 0 and **isgn** = 0;
l not a power of 2.

Example

Compute the forward discrete Fourier transform of two conjugate-symmetric COMPLEX*16 data sets of length 1024. Only the first 513 elements of the input data sets are stored. The length of working storage is 3*1024+4 = 3076.

```
INTEGER*8  INIT,ISGN,L
COMPLEX*16 ZIN1(513),ZIN2(513)
REAL*8     WORK(3076),XOUT1(1024),XOUT2(1024)
L = 1024
INIT = 1
CALL CRFFT2 (INIT,ISGN,L,ZIN1,WORK,XOUT1)  ! INITIALIZE
INIT = 0
ISGN = -1
CALL CRFFT2 (INIT,ISGN,L,ZIN1,WORK,XOUT1)  ! FIRST TRANSFORM
CALL CRFFT2 (INIT,ISGN,L,ZIN2,WORK,XOUT2)  ! SECOND TRANSFORM
```

Real-to-Complex One-Dimensional FFT — RCFFT2

Purpose

Given an array of real data, this subprogram computes the one-dimensional, real-to-complex, unscaled forward or inverse, discrete Fourier transform using a radix 2 fast Fourier transform (FFT) algorithm optimized for real input.

The one-dimensional unscaled forward discrete Fourier transform of a data set, $z(n)$, for $n = 1, 2, \ldots, l$, is defined by

$$Z(m) = \sum_{n=1}^{l} z(n)\, e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, \ldots, l$ and $i = \sqrt{-1}$.

When the sequence $z(n)$ is real, the sequence $Z(m)$ is conjugate-symmetric about $Z(l/2+1)$, i.e.,

$$Im(Z(1)) = Im(Z(l/2 + 1)) = 0$$

and

$$Z(l/2+1 + m) = \overline{Z}(l/2+1 - m), \quad m = 1, 2, \ldots, l/2-1,$$

where $\overline{Z}$ is the complex conjugate of Z.

This subprogram actually computes twice the above quantity:

$$Z(m) = 2 \sum_{n=1}^{l} z(n)\, e^{-2\pi i(m-1)(n-1)/l}$$

Alternatively, the one-dimensional unscaled inverse discrete Fourier transform of the data set $Z(m)$, for $m = 1, 2, \ldots, l$, is defined by

$$z(n) = \sum_{m=1}^{l} Z(m)\, e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, \ldots, l$.

Again, twice the above quantity is what is actually computed:

$$z(n) = 2 \sum_{m=1}^{l} Z(m)\, e^{+2\pi i(m-1)(n-1)/l}$$

This subprogram requires that l be a power of 2, i.e., of the form $l = 2^p$, where $p \geq 0$.

Usage

Because it is common to use one data set length repetitively, this subprogram has a separate initialization call so that the setup can be performed only once for each different transform size. Therefore, you will always have at least two **CALL** statements to the FFT subprogram, using the same working storage array.

SCILIB:

```
INTEGER*8    init, isgn, l
REAL*8       xin(l), work(3*l+4)
COMPLEX*16   zout(l/2+1)
CALL RCFFT2 (init, isgn, l, xin, work, zout)
```

RCFFT2 · Continued

Input

init Initialization flag:
init $\neq$ 0 initialize **work** for subsequent transforms of length **l**.
init = 0 compute transform.

isgn Operation flag: if **init** = 0,
isgn $<$ 0 compute forward transform.
isgn $>$ 0 compute unscaled inverse transform.

The sign of the exponential is the same as the sign of **isgn**.

l Number of data points, of the form $l = 2^p$, with $p \geq 0$.

xin Array containing the data set to be transformed. Not used if **init** $\neq$ 0.

Working Storage

work If **init** $\neq$ 0, **work** is initialized for computing transforms of length **l**.

If **init** = 0, **work** must have been initialized by a previous call with this value of **l** in which **init** $\neq$ 0.

Output

zout Array containing the first l/2+1 elements of transformed data if **init** = 0. Not used if **init** $\neq$ 0.

Notes

It is usual to scale the inverse transform by multiplying the summation by 1/l so that the inverse transform of the forward transform of a data set returns the original data. This subprogram replaces this scaling with scaling both forward and inverse transforms by a factor of 2, meaning that the inverse transform of the forward transform of a data set does not return the original data.

If an error in the arguments is detected, the subprograms call error handler XERSCI, which writes an error message onto the standard error file and terminates execution. The standard version of XERSCI (refer to Chapter 9, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are

isgn = 0, and
l not a power of 2.

Example

Compute the forward discrete Fourier transform of two REAL*8 data sets of length 1024. Only the first 513 elements of the transformed data are computed. The length of working storage is 3*1024+4 = 3076.

```
INTEGER*8  INIT,ISGN,L
REAL*8     XIN1(1024),XIN2(1024),WORK(3076)
COMPLEX*16 ZOUT1(513),ZOUT2(513)
L = 1024
INIT = 1
CALL RCFFT2 (INIT,ISGN,L,XIN1,WORK,ZOUT1)  ! INITIALIZE
INIT = 0
ISGN = -1
CALL RCFFT2 (INIT,ISGN,L,XIN1,WORK,ZOUT1)  ! FIRST TRANSFORM
CALL RCFFT2 (INIT,ISGN,L,XIN2,WORK,ZOUT2)  ! SECOND TRANSFORM
```

Purpose

Given a number of one-dimensional real data sets, subroutine RFFTMLT computes the nonredundant portion of all of their one-dimensional forward real-to-complex discrete Fourier transforms using a radix 2-3-5 fast Fourier transform (FFT) algorithm optimized for real input. Alternatively, given the nonredundant parts of a number of conjugate-symmetric one-dimensional complex data sets, subroutine RFFTMLT computes the inverse complex-to-real discrete Fourier transform using a radix 2-3-5 FFT algorithm optimized for real output.

The multiple forward, scaled, real-to-complex Fourier transform of a real set of data $z(n)$, for $n = 1, 2, ..., l$, is defined by:

$$Z(m) = \sum_{n=1}^{l} \frac{1}{l} z(n)\, e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, ..., l$, where $i = \sqrt{-1}$ and
$Z(1,2,...,l)$ is one of the real input data sets.
The sequence $Z(m)$ is conjugate-symmetric about $Z(l/2+1)$, i.e.,

$$Im(Z(1)) = Im(Z(l/2 + 1)) = 0$$

and

$$Z(l/2+1 + m) = \overline{Z}(l/2+1 - m), \quad m = 1, 2, ..., l/2-1,$$

where $\overline{Z}$ is the complex conjugate of Z. Therefore, the nonredundant part consists of the first $l/2+1$ elements of Z, which is all of Z that is computed or stored.

Alternatively, if $Z(m)$, for $m = 1, 2, ..., l$, is a conjugate-symmetric complex data set, the multiple backward, unscaled complex-to-real Fourier transform of $Z(m)$ is defined by:

$$z(n) = \sum_{m=1}^{l} Z(m)\, e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, ..., l$. Only the nonredundant part of Z is used.

This subprogram performs forward or inverse transform operations simultaneously on a number of data sets. It requires that the length l of the data sets be a product of powers of 2, 3, and 5, i.e., of the form:

$$l = 2^p\, 3^q\, 5^r,$$

where $p, q, r \geq 0$, and where either $l = 1$ or l is even. Refer to "Notes" for a partial list of permissible values of l.

FFTFAX/RFFTMLT Continued

Usage

Because it is common to use one data set length repetitively, subroutine RFFTMLT has a separate initialization subprogram, FFTFAX, so that the setup can be performed only once for each different transform size. You will, therefore, always have at least one **CALL FFTFAX** statement and at least one **CALL RFFTMLT** statement, using the same working storage arrays. Refer to "Example."

SCILIB:

```
INTEGER*8 work3(19), incl, incn, l, n, isgn
REAL*8    x(lenx), work1(2*l*n), work2(2*l)
```

Initialization call:

CALL FFTFAX (l, work3, work2)

Transform call:

CALL RFFTMLT (x, work1, work2, work3, incl, incn, l, n, isgn)

Input

x Array containing **n** one-dimensional data sets, each consisting of **l** real data points or the first **l**/2+1 complex data points of a conjugate-symmetric complex data set of length **l**, to be transformed. Typically, **x** is a two- or three-dimensional array with each set of data being a one-dimensional array section. Refer to "Notes" for suggested usages.

Treating **x** as a one-dimensional array results in:

$$\mathbf{lenx} = (\mathbf{l}+1)\times\mathbf{incl} + (\mathbf{n}-1)\times\mathbf{incn} + 1.$$

For a forward real-to-complex transform, the i-th real data point of the j-th data set, $1 \le i \le \mathbf{l}$, $1 \le j \le \mathbf{n}$, is stored in:

$$\mathbf{x}((i-1)\times\mathbf{incl} + (j-1)\times\mathbf{incn} + 1).$$

For an inverse complex-to-real transform, the real part of the i-th data point of the j-th data set, $1 \le i \le \mathbf{l}/2+1$, $1 \le j \le \mathbf{n}$, is stored in:

$$\mathbf{x}((2\times i-2)\times\mathbf{incl} + (j-1)\times\mathbf{incn} + 1)$$

and the imaginary part is stored in:

$$\mathbf{x}((2\times i-1)\times\mathbf{incl} + (j-1)\times\mathbf{incn} + 1),$$

respectively.

incl Storage increment between successive elements of the same data set, **incl** > 0. Use **incl** = 1 if each data set is stored contiguously in **x**.

incn Storage increment between corresponding data points of successive data sets, **incn** > 0.

l Number of data points in each complete data set, of the form $\mathbf{l} = 2^p\ 3^q\ 5^r$, with $q, r \ge 0$ and either $\mathbf{l} = 1$ or $p \ge 1$.

n The number of data sets, **n** > 0.

isgn Option flag:
isgn = −1 compute real-to-complex forward transform.
isgn = +1 compute complex-to-real inverse transform.

Working Storage

work1 Array used for work space.

work2 Array, initialized by FFTFAX for use as work space in RFFTMLT.

work3 Array, initialized by FFTFAX for use as work space in RFFTMLT. FFTFAX returns **work3**(1) = -99 if **l** is not factorable as specified above.

Output

x The transformed data replaces the input.

For a forward real-to-complex transform, the real part of the i-th output point of the j-th data set, $1 \le i \le l/2+1$, $1 \le j \le n$, is stored in:

x((2×i-2)×**incl** + (j-1)×**incn** + 1)

and the imaginary part is stored in

x((2×i-1)×**incl** + (j-1)×**incn** + 1),

respectively. If needed, the remaining (**l**/ 2 - 1) × **n** complex output values may be formed by using the conjugate-symmetry condition.

For an inverse complex-to-real transform, the i-th real output point of the j-th data set, $1 \le i \le l$, $1 \le j \le n$, is stored in:

x((i-1)×**incl** + (j-1)×**incn** + 1).

Notes

Typically, **x** will be a two- or three-dimensional array with each set of data being a one-dimensional section of the array, i.e., all but one subscript will be constant within a data set.

If **x** is a two-dimensional array of dimension **ldx** by **mdx**, and if the data sets are stored in the columns of **x**, then **l** ≤ **ldx**, **n** ≤ **mdx**, **incl** = 1, and **incn** = **ldx**. For example:

CALL RFFTMLT (x, work1, work2, work3, 1, ldx, l, n, isgn)

If **x** is a two-dimensional array as above and data sets are stored in rows of **x**, then **l** ≤ **mdx**, **n** ≤ **ldx**, **incl** = **ldx**, and **incn** = 1. For example:

CALL RFFTMLT (x, work1, work2, work3, ldx, 1, l, n, isgn)

The subprograms generally are faster if the data sets are the rows of the arrays, so that **incn** = 1, rather than the columns. Otherwise, it is generally better to have an odd value of **incn**.

If **x** is a three-dimensional array of dimension **ldx** by **mdx** by **ndx**, then **incl** and **incn** will usually be 1, **ldx**, or **ldx**×**mdx**, depending on which of the subscripts of the three-dimensional array varies within a data set, which subscript varies between data sets, and which remains constant. Specifically, if the subscript that varies within a data set is the

1st subscript, use **incl** = 1.
2nd subscript, use **incl** = **ldx**.
3rd subscript, use **incl** = **ldx**×**mdx**.

Similarly, if the subscript that varies between data sets is the

1st subscript, use **incn** = 1.
2nd subscript, use **incn** = **ldx**.
3rd subscript, use **incn** = **ldx**×**mdx**.

If an error in the arguments is detected, the subprograms call error handler XERSCI, which writes an error message onto the standard error file and terminates execution. The standard version of XERSCI (refer to Chapter 9, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are:

work3(1) = -99,
incl ≤ 0,
incn ≤ 0,
l not of the required form $2^p\ 3^q\ 5^r$, with **l** = 1 or **l** even,
n ≤ 0, and
isgn ≠ ± 1.

The following list indicates some of the permissible values of **l**. Although **l** can be any even value of the form $2^p\ 3^q\ 5^r$ where $q,r \geq 0$ and either **l** = 1 or $p \geq 1$, this list only shows the values not exceeding 1000:

1, 2, 4, 6, 8, 10, 12, 16, 18, 20, 24, 30, 32, 36, 40, 48, 50, 54, 60, 64, 72, 80, 90, 96, 100, 108, 120, 128, 144, 150, 160, 162, 180, 192, 200, 216, 240, 250, 256, 270, 288, 300, 320, 324, 360, 384, 400, 432, 450, 480, 486, 500, 512, 540, 576, 600, 640, 648, 720, 750, 768, 800, 810, 864, 900, 960, 972, and 1000.

Example 1 Compute the forward discrete Fourier transform of 256 real data sets of length 1024, stored as columns of arrays X whose dimensions are 1025 by 256.

```
INTEGER*8 WORK3(19),INCL,INCN,L,N,ISGN
REAL*8    X(1025,256),WORK1(1048576),WORK2(512)
L = 1024
INCL = 1
N = 256
INCN = 1025
ISGN = -1
CALL FFTFAX (L,WORK3,WORK2)
IF ( WORK3(1) .EQ. -99 ) THEN
  handle error condition
END IF
CALL RFFTMLT (X,WORK1,WORK2,WORK3,INCL,INCN,L,N,ISGN)
```

Example 2 Compute the inverse discrete Fourier transform of 1024 sets of conjugate-symmetric complex data of length 256, stored as rows of arrays X whose dimensions are 1025 by 256.

```
INTEGER*8 WORK3(19),INCL,INCN,L,N,ISGN
REAL*8    X(1025,256),WORK1(1048576),WORK2(512)
L = 256
INCL = 1025
N = 1024
INCN = 1
ISGN = 1
CALL FFTFAX (L,WORK3,WORK2)
IF ( WORK3(1) .EQ. -99 ) THEN
  handle error condition
END IF
CALL RFFTMLT (X,WORK1,WORK2,WORK3,INCL,INCN,L,N,ISGN)
```

7

Correlation and Convolution Subprograms

Overview

This chapter explains how to use the SCILIB subprograms available for correlations, convolutions, and related operations such as filtering by means of convolutions.

Chapter Objectives

After reading this chapter you will

- know how to use the described subprograms to compute correlation and convolution

What You Need to Know to Use These Subprograms

The subprograms presented here can be used to compute both discrete correlations and discrete convolutions. See the specific subprogram descriptions for details.

Supplemental Reading

Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, Inc. 1975.

Subprogram Descriptions

FILTERG — Discrete Correlation

Purpose

This subprogram computes the fully engaged portion of the discrete correlation of a data vector and a filter vector. It can be used to compute the complete discrete correlation (the fully engaged portion plus the *tails*) by appending zeros to the ends of the data vector. Refer to "Example 2."

If f_i, $i = 1, 2, \ldots, m$, and x_i, $i = 1, 2, \ldots, n$, are a filter vector and a data vector, respectively, their discrete correlation $\tilde{y_i}$ is defined by

$$\tilde{y_i} = \sum_j f_j x_{i+j-1},$$

for $i = -m+2, -m+3, \ldots, n$, where the sum is taken over all indices j for which both f_j and x_{i+j-1} are defined.

This subprogram computes only the fully engaged portion of the correlation, i.e., the part where the sums have exactly $\min(m,n)$ terms. Hence, if $m \leq n$, it computes

$$y_i = \sum_{j=1}^{m} f_j x_{i+j-1},$$

for $i = 1, 2, \ldots, n-m+1$.

The discrete convolution $\tilde{z_i}$ of the vectors f and x is defined by

$$\tilde{z_i} = \sum_j f_{i-j+1} x_j,$$

for $i = 1, 2, \ldots, m+n-1$, where the sum is taken over all indices j for which both f_{i-j+1} and x_j are defined.

This subprogram computes only the fully engaged portion of the convolution, i.e., the part where the sums have exactly $\min(m,n)$ terms. Hence, if $m \leq n$, it computes

$$z_i = \sum_{j=1}^{m} f_{m+1-j} x_{i+j-1}$$

for $i = 1, 2, \ldots, n-m+1$. A comparison of the definitions of y_i and z_i shows that the convolution may be computed by storing the f vectors in reverse order before calling FILTERG.

Usage

SCILIB:

```
INTEGER*8 m, n
REAL*8    f(m), x(n), y(n-m+1)
CALL FILTERG (f, m, x, n, y)
```

Input

f	Array containing the filter vector f of length **m**.
m	The length of the f vector, **m** > 0.
x	Array containing the data vector x of length **n**.
n	The length of the x vector, **n** ≥ **m**.

Output **y** The fully engaged correlation vector y of length **n-m+1**.

Notes To compute the complete correlation vector, including both tails as well as the fully engaged portion, append $m-1$ zeros to each end of the x vector. The fully engaged portion of the correlation of the resulting vector is the complete correlation corresponding to the original x vector. Refer to "Example 2."

If an error in the arguments is detected, the subprograms call error handler XERSCI, which writes an error message onto the standard error file and terminates execution. The standard version of XERSCI (refer to Chapter 9, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure.

m $\leq$ 0, and
n $\leq$ **m**.

Example 1 Compute the fully engaged portion of the discrete correlation of the REAL*8 vectors $f = (2, 1)$ and $x = (4, 1, 3, 5, 2)$ stored in arrays F and X, respectively. In this instance, $m = 2$ and $n = 5$.

```
INTEGER*8 M,N
REAL*8    F(2),X(5),Y(4)
DATA      F / 2.0 , 1.0 /
DATA      X / 4.0 , 1.0 , 3.0 , 5.0, 2.0 /
M = 2
N = 4
CALL FILTERG (F,M,X,N,Y)
```

The result is $y = (9, 5, 11, 12)$.

Example 2 Compute the complete discrete correlation of the REAL*8 vectors $f = (1, 2)$ and $x = (1, 3, 5)$ stored in arrays F and X, respectively. Thus $m = 2$, and to get the complete correlation, we append $m-1 = 1$ zero to each end of x, getting $\bar{x} = (0, 1, 3, 5, 0)$ and $n = 5$.

```
INTEGER*8 M,N
REAL*8    F(2),X(5),Y(4)
DATA      F / 1.0 , 2.0 /
DATA      X / 0.0 , 1.0 , 3.0 , 5.0, 0.0 /
M = 2
N = 4
CALL FILTERG (F,M,X,N,Y)
```

The result is $y = (2, 7, 13, 5)$.

Purpose

This subprogram computes the fully engaged portion of the discrete correlation of a data vector and a symmetric filter vector. It can be used to compute the complete discrete correlation (the fully engaged portion plus the *tails*) by appending zeros to the ends of the data vector. Refer to "Example 2."

If f_i, $i = 1, 2, \ldots, m$, and x_i, $i = 1, 2, \ldots, n$, are a filter vector and a data vector, respectively, their discrete correlation $\tilde{y_i}$ is defined by

$$\tilde{y_i} = \sum_j f_j x_{i+j-1},$$

for $i = -m+2, -m+3, \ldots, n$, where the sum is taken over all indices j for which both f_j and x_{i+j-1} are defined.

The filter vector is assumed to be symmetric, i.e., $f_i = f_{m+1-i}$, $i = 1, 2, \ldots, \lfloor m/2 \rfloor$, so only the first $\lceil m/2 \rceil$ elements of f must be stored.

This subprogram computes only the fully engaged portion of the correlation, i.e., the part where the sums have exactly $\min(m,n)$ terms. Hence, if $m \leq n$, it computes

$$y_i = \begin{cases} f_{(m+1)/2}\, x_{i+(m+1)/2} + \displaystyle\sum_{j=1}^{(m-1)/2} f_j (x_{i+j-1} + x_{i+m-j}), & \text{if } m \text{ odd} \\ \\ \displaystyle\sum_{j=1}^{m/2} f_j (x_{i+j-1} + x_{i+m-j}), & \text{if } m \text{ even} \end{cases}$$

for $i = 1, 2, \ldots, n-m+1$.

The fully engaged portion of the discrete convolution z_i of the vectors f and x is defined by

$$z_i = \sum_{j=1}^{m} f_{m+1-j} x_{i+j-1}$$

for $i = 1, 2, \ldots, n-m+1$. Because f is a symmetric vector, $y_i = z_i$ for $i = 1, 2, \ldots, n-m+1$; i.e., the fully engaged discrete correlation is identical to the fully engaged discrete convolution.

Usage

SCILIB:

```
INTEGER*8 m, n
REAL*8    f((m+1)/2), x(n), y(n-m+1)
CALL FILTERS (f, m, x, n, y)
```

Input

f Array containing the first (m+1)/2 elements of the filter vector f of length **m**.

m The length of the f vector, **m** > 0.

	x	Array containing the data vector x of length **n**.
	n	The length of the x vector, $\mathbf{n} \geq \mathbf{m}$.
Output	**y**	The fully engaged correlation vector y of length **n-m+1**.

Notes

To compute the complete correlation vector, including both tails as well as the fully engaged portion, append $m-1$ zeros to each end of the x vector. The fully engaged portion of the correlation of the resulting vector is the complete correlation corresponding to the original x vector.

If an error in the arguments is detected, the subprograms call error handler XERSCI, which writes an error message onto the standard error file and terminates execution. The standard version of XERSCI (refer to Chapter 9, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are:

$\mathbf{m} \leq 0$, and
$\mathbf{n} \leq \mathbf{m}$.

Refer to "Example 2."

Example 1

Compute the fully engaged portion of the discrete correlation of the REAL*8 vectors $f = (-1, 2, -1)$ and $x = (4, 1, 3, 5, 2)$ stored in arrays F and X, respectively. In this instance, $m = 3$ and $n = 5$.

```
INTEGER*8 M,N
REAL*8    F(2),X(5),Y(3)
DATA      F / -1.0 , 4.0 /
DATA      X / 4.0 , 1.0 , 3.0 , 5.0, 2.0 /
M = 3
N = 4
CALL FILTERS (F,M,X,N,Y)
```

The result is $y = (-3, 6, 15)$.

Example 2

Compute the complete discrete correlation of the REAL*8 vectors $f = (1, 2, 1)$ and $x = (1, 3, 5)$ stored in arrays F and X, respectively. Thus $m = 3$, and to get the complete correlation, we append $m-1 = 2$ zeros to each end of x, getting $\bar{x} = (0, 0, 1, 3, 5, 0, 0)$ and $n = 7$.

```
INTEGER*8 M,N
REAL*8    F(2),X(7),Y(5)
DATA      F / 1.0 , 2.0 /
DATA      X / 0.0, 0.0 , 1.0 , 3.0 , 5.0, 0.0, 0.0 /
M = 2
N = 4
CALL FILTERS (F,M,X,N,Y)
```

The result is $y = (1, 5, 12, 13, 5)$.

8

Linear Recurrences

Overview

This chapter explains how to use SCILIB subprograms for a variety of linear recurrence operations.

Chapter Objectives

After reading this chapter you will

- be able to recognize a recurrence
- know how to use the described subprograms

What You Need to Know to Use These Subprograms

A recurrence is a loop-carried data dependency between a value calculated during one iteration of a loop and used during a subsequent iteration. When the FORTRAN compiler detects a recurrence, it cannot vectorize the loop. Therefore, these subprograms, which use special, vectorizable algorithms, can be used to optimize FORTRAN loops containing certain linear recurrences.

Subprogram Descriptions

FOLR/FOLRP — First Order Linear Recurrence

Purpose

Given real vectors a and x of length n, these subprograms solve the first order linear recurrence

$$y_1 = x_1$$
$$y_i = x_i \pm a_i y_{i-1}, \quad i = 2, 3, \ldots, n,$$

overwriting the input x vector with the resulting y vector.

The operation indicated by $\pm$ above is specified by the subprogram name used:

FOLR	$\pm$ represents $-$:	$y_i = x_i - a_i y_{i-1}$
FOLRP	$\pm$ represents $+$:	$y_i = x_i + a_i y_{i-1}$

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

Usage

SCILIB:

```
INTEGER*8 n, inca, incx
REAL*8    a(lena), x(lenx)
CALL FOLR (n, a, inca, x, incx)

INTEGER*8 n, inca, incx
REAL*8    a(lena), x(lenx)
CALL FOLRP (n, a, inca, x, incx)
```

Input

n Number of elements of vectors a and x to be used in the recurrence, **n** $\geq$ 0. If **n** = 0, the subprograms do not reference **a** or **x**.

a Array of length **lena** = (**n**–1)× |**inca**|+1 containing the n-vector a.

inca Increment for the array **a**:
inca $>$ 0 a is stored forward in array **a**, i.e., a_i is stored in **a**((i -1)× **inca**+1).
inca $<$ 0 a is stored backward in array **a**, i.e., a_i is stored in **a**((i -**n**)× **inca**+1).
Use **inca** = 1 if the vector a is stored contiguously in array **a**, i.e., if a_i is stored in **a**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

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

incx Increment for the array **x**, **incx** $\neq$ 0:
incx $>$ 0 x is stored forward in array **x**, i.e., x_i is stored in **x**((i -1)× **incx**+1).
incx $<$ 0 x is stored backward in array **x**, i.e., x_i is stored in **x**((i -**n**)× **incx**+1).
Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., if x_i is stored in **x**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

Output

x If **n** = 0, then **x** is unchanged. Otherwise, the recurrence's solution vector overwrites the input.

Notes

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

If an error in the arguments is detected, the subprograms call error handler XERSCI, which writes an error message onto the standard error file and terminates execution. The standard version of XERSCI (refer to Chapter 9, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are:

n < 0,
inca $= 0$, and
incx $= 0$.

FORTRAN Equivalent

```
      SUBROUTINE FOLR (N, A,INCA, X,INCX)
      INTEGER*8 N,INCA,INCX
      REAL*8 A(*),X(*)
      IF ( N .LT. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IF ( INCA .EQ. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IF ( INCX .EQ. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IA = 1 + INCA
      IX = 1 + INCX
      IF ( INCA .LT. 0 ) IA = 1 - (N-2) * INCA
      IF ( INCX .LT. 0 ) IX = 1 - (N-2) * INCX
      DO 10 I = 2, N
        X(IX) = X(IX) - A(IA) * X(IX-INCX)
        IA = IA + INCA
        IX = IX + INCX
   10 CONTINUE
      RETURN
      END
```

Example Solve the REAL*8 first order linear recurrence

$$y_1 = x_1$$
$$y_i = x_i - a_i y_{i-1}, \quad i = 2, 3, \ldots, n,$$

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

```
      INTEGER*8 N,INCA,INCX
      REAL*8    A(20),X(20)
      N = 10
      INCA = 1
      INCX = 1
      CALL FOLR (N,A,INCA,X,INCX)
```

FOLR2/FOLR2P — First Order Linear Recurrence

Purpose

Given real vectors a and x of length n, these subprograms solve the first order linear recurrence

$$y_1 = x_1$$
$$y_i = x_i \pm a_i y_{i-1}, \quad i = 2, 3, \ldots, n,$$

for the y vector.

The operation indicated by ± above is specified by the subprogram name used:

FOLR2	± represents −:	$y_i = x_i - a_i y_{i-1}$
FOLR2P	± represents +:	$y_i = x_i + a_i y_{i-1}$

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

Usage

SCILIB:

```
INTEGER*8 n, inca, incx, incy
REAL*8    a(lena), x(lenx), y(leny)
CALL FOLR2 (n, a, inca, x, incx, y, incy)

INTEGER*8 n, inca, incx, incy
REAL*8    a(lena), x(lenx), y(leny)
CALL FOLR2P (n, a, inca, x, incx, y, incy)
```

Input

n Number of elements of vectors a, x, and y to be used in the recurrence, **n** ≥ 0. If **n** = 0, the subprograms do not reference **a**, **x**, or **y**.

a Array of length **lena** = (**n**–1)×|**inca**|+1 containing the n-vector a.

inca Increment for the array **a**:

inca > 0 a is stored forward in array **a**, i.e., a_i is stored in **a**((i–1)×**inca**+1).
inca < 0 a is stored backward in array **a**, i.e., a_i is stored in **a**((i–**n**)×**inca**+1).

Use **inca** = 1 if the vector a is stored contiguously in array **a**, i.e., if a_i is stored in **a**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

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

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., if x_i is stored in **x**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

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

incy > 0 y is stored forward in array **y**, i.e., y_i is stored in **y**((i-1)$\times$ **incy**+1).

incy < 0 y is stored backward in array **y**, i.e., y_i is stored in **y**((i-**n**)$\times$ **incy**+1).

Use **incy** = 1 if the vector y is stored contiguously in array **y**, i.e., if y_i is stored in **y**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

Output

y Array of length **leny** = (**n**–1)$\times$|**incy**|+1 containing the n-vector y. If **n** = 0, then **y** is unchanged. Otherwise, y is the recurrence's solution vector.

Notes

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

If an error in the arguments is detected, the subprograms call error handler XERSCI, which writes an error message onto the standard error file and terminates execution. The standard version of XERSCI (refer to Chapter 9, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are:

n < 0,
inca $= 0$,
incx $= 0$, and
incy $= 0$.

FOLR2/FOLR2P — Continued

FORTRAN Equivalent

```
      SUBROUTINE FOLR2 (N, A,INCA, X,INCX, Y,INCY)
      INTEGER*8 N,INCA,INCX,INCY
      REAL*8 A(*),X(*)
      IF ( N .LT. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IF ( INCA .EQ. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IF ( INCX .EQ. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IF ( INCY .EQ. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IA = 1 + INCA
      IX = 1 + INCX
      IY = 1 + INCY
      IF ( INCA .LT. 0 ) IA = 1 - (N-2) * INCA
      IF ( INCX .LT. 0 ) IX = 1 - (N-2) * INCX
      IF ( INCY .LT. 0 ) IY = 1 - (N-2) * INCY
      Y(IY-INCY) = X(IX-INCX)
      DO 10 I = 2, N
        Y(IY) = X(IX) - A(IA) * Y(IY-INCY)
        IA = IA + INCA
        IX = IX + INCX
        IY = IY + INCY
   10 CONTINUE
      RETURN
      END
```

Example Solve the REAL*8 first order linear recurrence

$$y_1 = x_1$$
$$y_i = x_i - a_i y_{i-1}, \quad i = 2, 3, \ldots, n,$$

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

```
      INTEGER*8 N,INCA,INCX,INCY
      REAL*8    A(20),X(20),Y(20)
      N = 10
      INCA = 1
      INCX = 1
      INCY = 1
      CALL FOLR2 (N,A,INCA,X,INCX,Y,INCY)
```

First Order Linear Recurrence — FOLRC

Purpose

Given a real coefficient α and a real vector a of length n, this subprogram solves the first order linear recurrence

$$x_1 = a_1$$
$$x_i = a_i + \alpha x_{i-1}, \quad i = 2, 3, \ldots, n.$$

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

Usage

SCILIB:

```
INTEGER*8 n, incx, inca
REAL*8    x(lenx), a(lena), alpha
CALL FOLRC (n, x, incx, a, inca, alpha)
```

Input

n Number of elements of vectors a and x to be used in the recurrence, **n** $\geq$ 0. If **n** = 0, the subprogram does not reference **a** or **x**.

incx Increment for the array **x**, **incx** $\neq$ 0:

incx $>$ 0 x is stored forward in array **x**, i.e., x_i is stored in **x**((*i*-1)× **incx**+1).

incx $<$ 0 x is stored backward in array **x**, i.e., x_i is stored in **x**((*i*-**n**)× **incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

a Array of length **lena** = (**n**–1)× |**inca**|+1 containing the n-vector a.

inca Increment for the array **a**:

inca $>$ 0 a is stored forward in array **a**, i.e., a_i is stored in **a**((*i*-1)× **inca**+1).

inca $<$ 0 a is stored backward in array **a**, i.e., a_i is stored in **a**((*i*-**n**)× **inca**+1).

Use **inca** = 1 if the vector a is stored contiguously in array **a**, i.e., if a_i is stored in **a**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

alpha The scalar α.

Output

x Array of length **lenx** = (**n**–1)× |**incx**|+1 containing the n-vector x. If **n** = 0, then **x** is unchanged. Otherwise, the recurrence's solution vector is returned.

FOLRC Continued

Notes

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

If an error in the arguments is detected, the subprograms call error handler XERSCI, which writes an error message onto the standard error file and terminates execution. The standard version of XERSCI (refer to Chapter 9, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are:

n < 0,
incx $= 0$, and
inca $= 0$.

FORTRAN Equivalent

```
      SUBROUTINE FOLRC (N, X,INCX, A,INCA, ALPHA)
      INTEGER*8 N,INCX,INCA
      REAL*8 X(*),A(*),ALPHA
      IF ( N .LT. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IF ( INCX .EQ. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IF ( INCA .EQ. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IA = 1 + INCA
      IX = 1 + INCX
      IF ( INCA .LT. 0 ) IA = 1 - (N-2) * INCA
      IF ( INCX .LT. 0 ) IX = 1 - (N-2) * INCX
      X(IX-INCX) = A(IA-INCA)
      DO 10 I = 2, N
        X(IX) = A(IA) + ALPHA * X(IX-INCX)
        IA = IA + INCA
        IX = IX + INCX
   10 CONTINUE
      RETURN
      END
```

Example

Solve the REAL*8 first order linear recurrence

$$x_1 = a_1$$
$$x_i = a_i + 4x_{i-1}, \quad i = 2, 3, \ldots, n,$$

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

```
      INTEGER*8 N,INCX,INCA
      REAL*8    X(20),A(20),ALPHA
      N = 10
      INCX = 1
      INCA = 1
      ALPHA = 4.0
      CALL FOLRC (N,X,INCX,A,INCA,ALPHA)
```

Last Term of First Order Linear Recurrence — FOLRN/FOLRNP

Purpose

Given real vectors a and x of length n, these subprograms solve for the last term of the first order linear recurrence

$$y_1 = x_1$$
$$y_i = x_i \pm a_i y_{i-1}, \quad i = 2, 3, \ldots, n,$$

i.e., returning y_n.

The operation indicated by ± above is specified by the subprogram name used:

FOLRN	± represents −:	$y_i = x_i - a_i y_{i-1}$
FOLRNP	± represents +:	$y_i = x_i + a_i y_{i-1}$

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

Usage

SCILIB:

```
INTEGER*8 n, inca, incx
REAL*8    yn, FOLRN, a(lena), x(lenx)
yn = FOLRN (n, a, inca, x, incx)

INTEGER*8 n, inca, incx
REAL*8    yn, FOLRNP, a(lena), x(lenx)
yn = FOLRNP (n, a, inca, x, incx)
```

Input

n Number of elements of vectors a and x to be used in the recurrence, **n** ≥ 0. If **n** = 0, the subprograms do not reference **a** or **x**.

a Array of length **lena** = (**n**–1)×|**inca**|+1 containing the n-vector a.

inca Increment for the array **a**:

inca > 0 a is stored forward in array **a**, i.e., a_i is stored in **a**((i–1)×**inca**+1).
inca < 0 a is stored backward in array **a**, i.e., a_i is stored in **a**((i–**n**)×**inca**+1).

Use **inca** = 1 if the vector a is stored contiguously in array **a**, i.e., if a_i is stored in **a**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

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

If **incx** = 0, then x_i = **x**(1) for all i. Refer to "Notes" to see how to use FOLRNP with **incx** = 0 to evaluate a polynomial. Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., if x_i is stored in **x**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

FOLRN/FOLRNP — Continued

Output

yn If **n** = 0, then **yn** = 0. Otherwise, the last term of the recurrence's solution is returned.

Notes

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

FOLRNP may be used to evaluate a polynomial $p(x) = \sum_{i=0}^{n} a_i x^i$ by recognizing that $p(x)$ is the last term of the recurrence

$$y_0 = a_n$$
$$y_i = a_{n-i} + y_{i-1}x, \quad i = 1, 2, \ldots, n.$$

Refer to "Example 2."

If an error in the arguments is detected, the subprograms call error handler XERSCI, which writes an error message onto the standard error file and terminates execution. The standard version of XERSCI (refer to Chapter 9, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are:

n < 0, and
inca = 0.

FORTRAN Equivalent

```
      REAL*8 FUNCTION FOLRN (N, A,INCA, X,INCX)
      INTEGER*8 N,INCA,INCX
      REAL*8 A(*),X(*)
      FOLRN = 0.0
      IF ( N .LT. 0 ) THEN
         CALL XERSCI (...)
         RETURN
      END IF
      IF ( INCA .EQ. 0 ) THEN
         CALL XERSCI (...)
         RETURN
      END IF
      IA = 1 + INCA
      IX = 1 + INCX
      IF ( INCA .LT. 0 ) IA = 1 - (N-2) * INCA
      IF ( INCX .LT. 0 ) IX = 1 - (N-2) * INCX
      FOLRN = X(IX-INCX)
      DO 10 I = 2, N
         FOLRN = X(IX) - A(IA) * FOLRN
         IA = IA + INCA
         IX = IX + INCX
   10 CONTINUE
      RETURN
      END
```

Example 1 Solve for the last term of the REAL*8 first order linear recurrence

$$y_1 = x_1$$
$$y_i = x_i - a_i y_{i-1}, \quad i = 2, 3, \ldots, n,$$

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

```
INTEGER*8 N,INCA,INCX
REAL*8    YN,FOLRN,A(20),X(20)
N = 10
INCA = 1
INCX = 1
YN = FOLRN(N,A,INCA,X,INCX)
```

Example 2 Evaluate the REAL*8 polynomial $p(x) = \sum_{i=0}^{10} a_i x^i$, where a is a vector 11 elements long stored in one-dimensional array A of dimension 0:20.

```
INTEGER*8 N,INCA,INCX
REAL*8    P,FOLRNP,A(0:20),X
N = 11
INCA = -1
INCX = 0
P = FOLRNP(N,A,INCA,X,INCX)
```

RECPP — Compute Vector of Partial Products

Purpose

Given real vector a of length n, this subprogram computes the n-vector x of partial products

$$x_i = \prod_{j=1}^{i} a_j, \quad i = 1, 2, \ldots, n.$$

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

Usage

SCILIB:

```
INTEGER*8 n, incx, inca
REAL*8    x(lenx), a(lena)
CALL RECPP (n, x, incx, a, inca)
```

Input

n Number of elements of vectors a and x, **n** $\geq$ 0. If **n** = 0, the subprograms do not reference **a** or **x**.

incx Increment for the array **x**, **incx** $\neq$ 0:

incx $>$ 0 x is stored forward in array **x**, i.e., x_i is stored in **x**((*i*−1)× **incx**+1).

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

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

a Array of length **lena** = (**n**−1)× |**inca**|+1 containing the n-vector a.

inca Increment for the array **a**:

inca $>$ 0 a is stored forward in array **a**, i.e., a_i is stored in **a**((*i*−1)× **inca**+1).

inca $<$ 0 a is stored backward in array **a**, i.e., a_i is stored in **a**((*i*−**n**)× **inca**+1).

Use **inca** = 1 if the vector a is stored contiguously in array **a**, i.e., if a_i is stored in **a**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

Output

x Array of length **lenx** = (**n**−1)× |**incx**|+1 containing the n-vector x. If **n** = 0, then **x** is unchanged. Otherwise, the vector of partial products replaces the input.

Notes

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

If an error in the arguments is detected, the subprograms call error handler XERSCI, which writes an error message onto the standard error file and terminates execution. The standard version of XERSCI (refer to Chapter 9, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are:

n $<$ 0,
incx = 0, and
inca = 0.

FORTRAN Equivalent

```
      SUBROUTINE RECPP (N, X,INCX, A,INCA)
      INTEGER*8 N,INCX,INCA
      REAL*8 X(*),A(*)
      IF ( N .LT. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IF ( INCX .EQ. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IF ( INCA .EQ. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IA = 1 + INCA
      IX = 1 + INCX
      IF ( INCA .LT. 0 ) IA = 1 - (N-2) * INCA
      IF ( INCX .LT. 0 ) IX = 1 - (N-2) * INCX
      X(IX-INCX) = A(IA-INCA)
      DO 10 I = 2, N
        X(IX) = X(IX-INCX) * A(IA)
        IA = IA + INCA
        IX = IX + INCX
   10 CONTINUE
      RETURN
      END
```

Example Compute the REAL*8 vector of partial products of the vector a, where a is a vector 10 elements long stored in a one-dimensional array A of dimension 20. The result is stored in array X, also of dimension 20.

```
INTEGER*8 N,INCX,INCA
REAL*8    X(20),A(20)
N = 10
INCA = 1
INCX = 1
CALL RECPP (N,X,INCX,A,INCA)
```

RECPS — Compute Vector of Partial Sums

Purpose

Given real vector a of length n, this subprogram computes the n-vector x of partial sums

$$x_i = \sum_{j=1}^{i} a_j, \quad i = 1, 2, \ldots, n.$$

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

Usage

SCILIB:

```
INTEGER*8 n, incx, inca
REAL*8    x(lenx), a(lena)
CALL RECPS (n, x, incx, a, inca)
```

Input

n Number of elements of vectors a and x, **n** ≥ 0. If **n** = 0, the subprograms do not reference **a** or **x**.

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

incx > 0 x is stored forward in array **x**, i.e., x_i is stored in **x**((*i*-1)×**incx**+1).

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

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., if x_i is stored in **x**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

a Array of length **lena** = (**n**–1)×|**inca**|+1 containing the n-vector a.

inca Increment for the array **a**:

inca > 0 a is stored forward in array **a**, i.e., a_i is stored in **a**((*i*-1)×**inca**+1).

inca < 0 a is stored backward in array **a**, i.e., a_i is stored in **a**((*i*-**n**)×**inca**+1).

Use **inca** = 1 if the vector a is stored contiguously in array **a**, i.e., if a_i is stored in **a**(*i*). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

Output

x Array of length **lenx** = (**n**–1)×|**incx**|+1 containing the n-vector x. If **n** = 0, then **x** is unchanged. Otherwise, the vector of partial sums replaces the input.

Notes

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

If an error in the arguments is detected, the subprograms call error handler XERSCI, which writes an error message onto the standard error file and terminates execution. The standard version of XERSCI (refer to Chapter 9, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are:

n < 0,
incx = 0, and
inca = 0.

FORTRAN Equivalent

```
      SUBROUTINE RECPS (N, X,INCX, A,INCA)
      INTEGER*8 N,INCX,INCA
      REAL*8 X(*),A(*)
      IF ( N .LT. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IF ( INCX .EQ. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IF ( INCA .EQ. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IA = 1 + INCA
      IX = 1 + INCX
      IF ( INCA .LT. 0 ) IA = 1 - (N-2) * INCA
      IF ( INCX .LT. 0 ) IX = 1 - (N-2) * INCX
      X(IX-INCX) = A(IA-INCA)
      DO 10 I = 2, N
        X(IX) = X(IX-INCX) + A(IA)
        IA = IA + INCA
        IX = IX + INCX
   10 CONTINUE
      RETURN
      END
```

Example Compute the REAL*8 vector of partial sums of the vector a, where a is a vector 10 elements long stored in a one-dimensional array A of dimension 20. The result is stored in array X, also of dimension 20.

```
INTEGER*8 N,INCX,INCA
REAL*8    X(20),A(20)
N = 10
INCA = 1
INCX = 1
CALL RECPS (N,X,INCX,A,INCA)
```

SOLR — Second Order Linear Recurrence

Purpose

Given real vectors a and b of length n and the first two elements of n-vector x, this subprogram solves the second order linear recurrence

$$x_i = a_{i-2}x_{i-1} + b_{i-2}x_{i-2}, \quad i = 3, 4, \ldots, n.$$

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

Usage

SCILIB:

```
INTEGER*8 n, inca, incb, incx
REAL*8    a(lena), b(lenb), x(lenx)
CALL SOLR (n, a, inca, b, incb, x, incx)
```

Input

n Number of elements of vectors a, b, and x to be used in the recurrence, **n** ≥ 0. If **n** = 0, the subprogram does not reference **a**, **b**, or **x**.

a Array of length **lena** = (**n**–1)× |**inca**|+1 containing the n-vector a.

inca Increment for the array **a**:

inca > 0 a is stored forward in array **a**, i.e., a_i is stored in **a**((i–1)× **inca**+1).
inca < 0 a is stored backward in array **a**, i.e., a_i is stored in **a**((i–**n**)× **inca**+1).

Use **inca** = 1 if the vector a is stored contiguously in array **a**, i.e., if a_i is stored in **a**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

b Array of length **lenb** = (**n**–1)× |**incb**|+1 containing the n-vector b.

incb Increment for the array **b**:

incb > 0 b is stored forward in array **b**, i.e., b_i is stored in **b**((i–1)× **incb**+1).
incb < 0 b is stored backward in array **b**, i.e., b_i is stored in **b**((i–**n**)× **incb**+1).

Use **incb** = 1 if the vector b is stored contiguously in array **b**, i.e., if b_i is stored in **b**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

x Array of length **lenx** = (**n**–1)× |**incx**|+1 containing the first two elements, x_1 and x_2 of the n-vector x.

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

incx > 0 x is stored forward in array **x**, i.e., x_i is stored in **x**((i–1)× **incx**+1).
incx < 0 x is stored backward in array **x**, i.e., x_i is stored in **x**((i–**n**)× **incx**+1).

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., if x_i is stored in **x**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

Output **x** If **n** = 0, then **x** is unchanged. Otherwise, the recurrence's solution vector replaces the input.

Notes The result is unspecified if **a**, **b**, or **x** overlap such that any element of a, b, or x shares a memory location with any other element of a, b, or y.

If an error in the arguments is detected, the subprograms call error handler XERSCI, which writes an error message onto the standard error file and terminates execution. The standard version of XERSCI (refer to Chapter 9, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are:

n < 0,
inca $= 0$,
incb $= 0$, and
incx $= 0$.

FORTRAN Equivalent

```
      SUBROUTINE SOLR (N, A,INCA, B,INCB, X,INCX)
      INTEGER*8 N,INCA,INCB,INCX
      REAL*8 A(*),B(*),X(*)
      IF ( N .LT. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IF ( INCA .EQ. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IF ( INCB .EQ. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IF ( INCX .EQ. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IA = 1
      IB = 1
      IX = 1 + INCX
      IF ( INCA .LT. 0 ) IA = 1 - (N-3) * INCA
      IF ( INCB .LT. 0 ) IB = 1 - (N-3) * INCB
      IF ( INCX .LT. 0 ) IX = 1 - (N-2) * INCX
      DO 10 I = 3, N
        X(IX+INCX) = A(IA) * X(IX) + B(IB) * X(IX-INCX)
        IA = IA + INCA
        IB = IB + INCB
        IX = IX + INCX
   10 CONTINUE
      RETURN
      END
```

incx Increment for the array **x**, **incx** $\neq 0$:

incx > 0 x is stored forward in array **x**, i.e., x_i is stored in **x**$((i-1)\times$**incx**$+1)$.

incx < 0 x is stored backward in array **x**, i.e., x_i is stored in **x**$((i-$**n**$)\times$**incx**$+1)$.

Use **incx** $= 1$ if the vector x is stored contiguously in array **x**, i.e., if x_i is stored in **x**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

Output

x If **n** $= 0$, then **x** is unchanged. Otherwise, the recurrence's solution vector replaces the input.

Notes

The result is unspecified if **a**, **b**, or **x** overlap such that any element of a, b, or x shares a memory location with any other element of a, b, or x.

If an error in the arguments is detected, the subprograms call error handler XERSCI, which writes an error message onto the standard error file and terminates execution. The standard version of XERSCI (refer to Chapter 9, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are:

n < 0,
inca $= 0$,
incb $= 0$, and
incx $= 0$.

Continued — SOLR3

FORTRAN Equivalent

```
      SUBROUTINE SOLR3 (N, A,INCA, B,INCB, X,INCX)
      INTEGER*8 N,INCA,INCB,INCX
      REAL*8 A(*),B(*),X(*)
      IF ( N .LT. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IF ( INCA .EQ. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IF ( INCB .EQ. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IF ( INCX .EQ. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IA = 1
      IB = 1
      IX = 1 + INCX
      IF ( INCA .LT. 0 ) IA = 1 - (N-3) * INCA
      IF ( INCB .LT. 0 ) IB = 1 - (N-3) * INCB
      IF ( INCX .LT. 0 ) IX = 1 - (N-2) * INCX
      DO 10 I = 3, N
        X(IX+INCX) = X(IX+INCX) + A(IA) * X(IX) + B(IB) * X(IX-INCX)
        IA = IA + INCA
        IB = IB + INCB
        IX = IX + INCX
   10 CONTINUE
      RETURN
      END
```

Example Solve the REAL*8 second order linear recurrence

$$
\begin{aligned}
y_1 &= x_1 \\
y_2 &= x_2 \\
y_i &= x_i + a_{i-2}y_{i-1} + b_{i-2}y_{i-2}, \quad i = 3, 4, \ldots, n.
\end{aligned}
$$

where a, b, and x are vectors 10 elements long stored in one-dimensional arrays A, B, and X of dimension 20, and y overwrites x.

```
      INTEGER*8 N,INCA,INCB,INCX
      REAL*8    A(20),B(20),X(20)
      N = 10
      INCA = 1
      INCB = 1
      INCX = 1
      CALL SOLR3 (N,A,INCA,B,INCB,X,INCX)
```

SOLRN — Last Term of Second Order Linear Recurrence

Purpose

Given real vectors a and b of length n and the first two elements of n-vector x, this subprogram solves for the last term of the second order linear recurrence

$$x_i = a_{i-2}x_{i-1} + b_{i-2}x_{i-2}, \quad i = 3, 4, \ldots, n,$$

i.e., returning x_n.

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

Usage

SCILIB:

```
INTEGER*8 n, inca, incb, incx
REAL*8    xn,SOLRN, a(lena), b(lenb), x(lenx)
xn = SOLRN (n, a, inca, b, incb, x, incx)
```

Input

n Number of elements of vectors a, b, and x to be used in the recurrence, **n** $\geq$ 0. If **n** = 0, the subprogram does not reference **a**, **b**, or **x**.

a Array of length **lena** = (**n**–1)×|**inca**|+1 containing the n-vector a.

inca Increment for the array **a**:

inca > 0 a is stored forward in array **a**, i.e., a_i is stored in **a**((i–1)×**inca**+1).

inca < 0 a is stored backward in array **a**, i.e., a_i is stored in **a**((i–**n**)×**inca**+1).

Use **inca** = 1 if the vector a is stored contiguously in array **a**, i.e., if a_i is stored in **a**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

b Array of length **lenb** = (**n**–1)×|**incb**|+1 containing the n-vector b.

incb Increment for the array **b**:

incb > 0 b is stored forward in array **b**, i.e., b_i is stored in **b**((i–1)×**incb**+1).

incb < 0 b is stored backward in array **b**, i.e., b_i is stored in **b**((i–**n**)×**incb**+1).

Use **incb** = 1 if the vector b is stored contiguously in array **b**, i.e., if b_i is stored in **b**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

x Array of length **lenx** = (**n**–1)×|**incx**|+1 containing the first two elements, x_1 and x_2 of the n-vector x.

incx Increment for the array **x**, **incx** $\neq$ 0:

incx > 0 x is stored forward in array **x**, i.e., x_i is stored in **x**((i -1)× **incx**+1).

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

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., if x_i is stored in **x**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

Output

xn If **n** = 0, then **xn** = 0. Otherwise, the last term of the recurrence's solution is returned.

x If **n** = 0, then **x** is unchanged. Otherwise, **x** is overwritten with intermediate results. These intermediate results are not necessarily what is shown in "FORTRAN Equivalent".

Notes

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

If an error in the arguments is detected, the subprograms call error handler XERSCI, which writes an error message onto the standard error file and terminates execution. The standard version of XERSCI (refer to Chapter 9, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are:

n < 0,
inca = 0,
incb = 0, and
incx = 0.

SOLRN — Continued

FORTRAN Equivalent

```
      REAL*8 FUNCTION SOLRN (N, A,INCA, B,INCB, X,INCX)
      INTEGER*8 N,INCA,INCB,INCX
      REAL*8 A(*),B(*),X(*)
      SOLRN = 0.0
      IF ( N .LT. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IF ( INCA .EQ. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IF ( INCB .EQ. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IF ( INCX .EQ. 0 ) THEN
        CALL XERSCI (...)
        RETURN
      END IF
      IA = 1
      IB = 1
      IX = 1 + INCX
      IF ( INCA .LT. 0 ) IA = 1 - (N-3) * INCA
      IF ( INCB .LT. 0 ) IB = 1 - (N-3) * INCB
      IF ( INCX .LT. 0 ) IX = 1 - (N-2) * INCX
      DO 10 I = 3, N   ! CAUTION: X NOT NECESSARILY RETURNED AS SHOWN
        X(IX+INCX) = A(IA) * X(IX) + B(IB) * X(IX-INCX)
        IA = IA + INCA
        IB = IB + INCB
        IX = IX + INCX
   10 CONTINUE
      IF ( N .EQ. 1 ) THEN
        SOLRN = X(IX-INCX)
      ELSE
        SOLRN = X(IX)
      END IF
      RETURN
      END
```

Example Solve for the last term of the REAL*8 second order linear recurrence

$$x_i = a_{i-2}x_{i-1} + b_{i-2}x_{i-2}, \quad i = 3, 4, \ldots, n,$$

where a, b, and x are vectors 10 elements long stored in one-dimensional arrays A, B, and X of dimension 20.

```
      INTEGER*8 N,INCA,INCB,INCX
      REAL*8    XN,SOLRN,A(20),B(20),X(20)
      N = 10
      INCA = 1
      INCB = 1
      INCX = 1
      X(1) = ...
      X(2) = ...
      XN = SOLRN (N,A,INCA,B,INCB,X,INCX)
```

9

Miscellaneous Routines

Overview

This chapter explains how to use SCILIB subprograms for operations that do not fit in the categories covered by other chapters. It includes a description of subprograms that

- sort the elements of a vector into ascending or descending order
- report errors detected in the usage of SCILIB subprograms

Chapter Objectives

After reading this chapter you will

- know how to use the described subprograms
- know how to change the method of error reporting in SCILIB subprograms

What You Need to Know to Use These Subprograms

Subprograms described in this chapter do not normally perform vector operations.

Supplemental Reading

Knuth, D.E. *The Art of Computer Programming,* Vol. 3: Sorting and Searching. Menlo Park, California: Addison-Wesley. 1973.

Subprogram Descriptions

ORDERS — Sort Array into Ascending Order

Purpose

This subprogram sorts data items into ascending order based on a sort key that may be all or part of each data record. The key may be a real number, a signed or unsigned integer, or a character string. Instead of rearranging the elements of the array, ORDERS returns an index vector containing the indices of the data elements in sorted order. The method has a linear computational complexity and is stable, that is, the original order of data with equal keys is preserved.

This subprogram uses a bucket sort algorithm to sort the data records. The length of the sort key determines the number of passes through the data. ORDERS has the option of processing either one or two bytes of the key per pass through the data. Using two bytes halves the number of passes at the expense of increased working storage and overhead per pass.

Usage

You may omit the last 1, 2, or 3 arguments from the CALL statement if the indicated default value is acceptable. Thus, the argument list may contain 5, 6, 7, or 8 arguments.

SCILIB:

```
INTEGER*8 mode, work(lwork), indx(n), n, ldx, keylen, iradix
REAL*8    x(ldx, n)
CALL ORDERS (mode, work, x, indx, n, ldx, keylen, iradix)

INTEGER*8 mode, work(lwork), x(ldx, n), indx(n), n, ldx, keylen,
          iradix
CALL ORDERS (mode, work, x, indx, n, ldx, keylen, iradix)
```

Input

mode Variable indicating data type and initial order option:

mode = 0	The key is an unsigned binary number of length 8 × **keylen** bits, and **indx** does not contain an initial order on input.
mode = 1	The key is a signed integer (INTEGER*8), and **indx** does not contain an initial order on input.
mode = 2	The key is a real number (REAL*8), and **indx** does not contain an initial order on input.
mode = 10	The key is an unsigned binary number of length **keylen** bytes, and **indx** contains an initial ordering on input.
mode = 11	The key is a signed integer (INTEGER*8), and **indx** contains an initial ordering on input.
mode = 12	The key is a real number (REAL*8), and **indx** contains an initial ordering on input.

x Array containing the data to be sorted.

indx If **mode** = 10, 11, or 12, an array of indices containing the initial data ordering, i.e., **x**(1,**indx**(i)) is the i-th smallest data item. Not used as input if **mode** = 0, 1, or 2.

n Number of data items to be sorted, **n** ≥ 1.

ldx The leading dimension of array **x** as declared in the calling program unit. Default = 1.

keylen Length of sort key as number of 8-bit bytes. **keylen** must be specified as 8 unless **mode** = 0 or 10. Default = 8.

iradix The number of bytes of the key that are processed per pass over the data, **iradix** = 1 or 2. Default = 1.

Working Storage

work An array whose length, **lwork**, depends on **iradix**:

iradix = 1 **lwork** = 257
iradix = 2 **lwork** = 65537

Output

mode Status response:

mode ≥ 0	Normal return (unchanged from input).
mode = –1	ORDERS called with fewer than 5 arguments.
mode = –2	ORDERS called with more than 8 arguments.
mode = –3	**ldx** ≤ 0 or **ldx** > 2^{24}.
mode = –4	**keylen** > 8 × **ldx**.
mode = –5	**iradix** ≠ 1 or 2.
mode = –6	**keylen** ≤ 0.
mode = –7	**n** ≤ 0 or **n** > 2^{24}.
mode = –8	**mode** ≠ 0, 1, 2, 10, 11, or 12.
mode = –9	**keylen** ≠ 8 for real or integer sort.

indx Array of indices containing the sorted data ordering, i.e., **x**(1,**indx**(i)) is the i-th smallest data item.

Notes You can combine several CALL ORDERS statements to use either a larger field or more than one field as the sort key. Make the first call to sort on the least significant part of the key with **mode** = 0, 1, or 2; then use **mode** = 10, 11, or 12 to sort on additional fields in increasing order of significance. Refer to "Example 2."

Example 1 Sort the elements of a REAL*8 vector x into ascending order, where x is a vector 100 elements long stored in a one-dimensional array **X** of dimension 200. The index array is returned in a one-dimensional array **INDX** of dimension 200.

```
INTEGER*8 MODE,WORK(257),INDX(200),N,LDX,KEYLEN,IRADIX
REAL*8    X(200)
MODE = 2
N = 100
LDX = 1
KEYLEN = 8
IRADIX = 1
CALL ORDERS (MODE,WORK,X,INDX,N,LDX,KEYLEN,IRADIX)
```

Example 2 A sparse matrix is represented by a set of **NZ** triples, $\{i, j, A_{ij}\}$. Sort the triples into order with primary key i and secondary key j. The data i, j, and A_{ij} are stored in arrays IROW, JCOL, and AIJ, all of dimension 500 and the resulting index array is returned in a one-dimensional array **INDX**, also of dimension 500.

```
INTEGER*8 IROW(500),JCOL(500),MODE,WORK(257),INDX(500),NZ
REAL*8    AIJ(500)
MODE = 1
CALL ORDERS (MODE,WORK,JCOL,INDX,NZ,1,8,1)
MODE = 11
CALL ORDERS (MODE,WORK,IROW,INDX,NZ,1,8,1)
```

XERSCI — SCILIB Error Handler

Purpose

XERSCI is the error handler for many of the subprograms in the SCILIB library, as indicated in the "Notes" section in the subprogram descriptions. As supplied in SCILIB, XERSCI writes one of the following error messages onto the standard error file:

```
**************************************************************************
* XERSCI: subprogram name called with invalid value of argument number iarg *
**************************************************************************
```

or

```
**************************************************************************
* XERSCI: error detected by subprogram name: text of error message       *
**************************************************************************
```

or

```
**************************************************************************
* XERSCI: error iarg detected by subprogram name: text of error message  *
**************************************************************************
```

where *name* is the name of the subprogram in which the error was detected, *iarg* is the argument number of the offending argument, and *text of error message* is a character string. If the main program is in FORTRAN, a call traceback is also written onto the standard error file. XERSCI then terminates execution with a nonzero exit status.

You may supply a version of XERSCI that alters this action. All SCILIB subprograms that call XERSCI have a **RETURN** statement after the **CALL** statement, so your version could perhaps set a flag in a common block and **RETURN**. The flag could be tested in the program unit that calls the SCILIB subprogram.

Usage

SCILIB:

```
CHARACTER*(*) name, messag
INTEGER*8     iarg
CALL XERSCI (name, iarg, messag)
```

Input

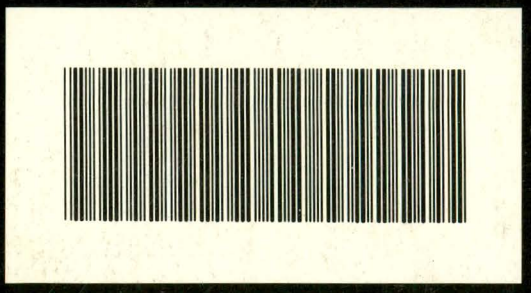
name The name of the subprogram in which the error was detected.

iarg If *iarg* > 0, the error message is printed in the first form given above and *iarg* is the number of the argument that was found to be in error. If *iarg* = 0, the error message is printed in the second form given above and *iarg* is not part of the error message. If *iarg* < 0, the error message is printed in the third form given above and *iarg* is the error number.

messag The text of the error message to be printed if **iarg** ≤ 0. Not used as input if **iarg** > 0.

Index

S



Order Number
DSW-360

Document Number
710-013630-002