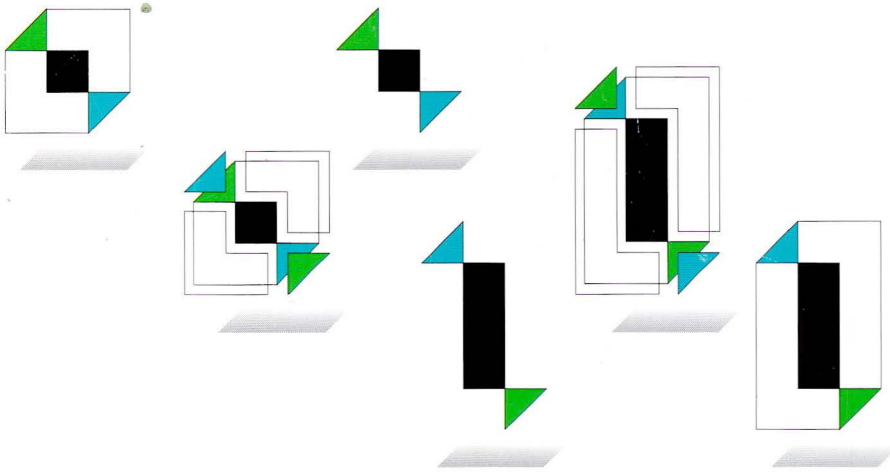




HP MLIB VECLIB User's Guide

Third Edition

HP MLIB VECLIB User's Guide

Third Edition



B5649-90007

HP MLIB VECLIB

August 1997

Printed in: USA

Notice

© Copyright Hewlett-Packard Company 1997. All Rights Reserved.
Reproduction, adaptation, or translation without prior written
permission is prohibited, except as allowed under the copyright laws.

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

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

**Revision Information for
HP MLIB VECLIB User's Guide**

Edition	Document No.	Description
Third	B5649-90007	Released August 1997. Includes general updates.
Second	B5649-90003	Released January 1997.
First	710-011030-006	Initial release October 1994.

Contents

Preface	xxi
Purpose and Audience	xxi
Organization	xxi
Notational Conventions	xxii
Associated Documents	xxiii
Ordering Documentation	xxv
Technical Assistance	xxv
1 Introduction to VECLIB.....	1
Overview	1
Chapter Objectives	2
What You Need to Know to Use VECLIB	3
Standardization	3
Two VECLIB Libraries	3
Accessing VECLIB	4
Interactions Between VECLIB, SCILIB, and LAPACK	5
Performance Value	6
Optimization	6
Parallel Processing	7
Profiling VECLIB Applications	9
Floating-Point Formats	10
Roundoff Effects	10
Data Types and Precision	10
VECLIB Naming Convention	10
Required Data Item Byte Lengths and How to Get Them	11
Error Handling	14
HP MLIB Man Pages	15
Support Services	15
2 Basic Vector Operations.....	17
Overview	17

Table of contents

Chapter Objectives	17
What You Need to Know to Use These Subprograms	18
BLAS Storage Conventions	18
Subprograms Included in This Chapter	21
ISAMAX/IDAMAX/.../IZAMAX Index of Maximum of Magnitudes	22
ISAMIN/IDAMIN/.../IZAMIN Index of Minimum of Magnitudes	25
ISCTxx/IDCTxx/IICTxx/.../IZCTxx Count Selected Vector Elements	28
ISMAX/IDMAX/IIMAX Index of Maximum Element of Vector	31
ISMIN/IDMIN/IIMIN Index of Minimum Element of Vector	33
ISSVxx/IDSVxx/IISVxx/ICSVxx/IZSVxx Search Vector for Element	35
SAMAX/DAMAX/IAMAX/SCAMAX/DZAMAX Maximum of Magnitudes	38
SAMIN/DAMIN/IAMIN/SCAMIN/DZAMIN Minimum of Magnitudes	41
SASUM/DASUM/IASUM/SCASUM/DZASUM Sum of Magnitudes	44
SAXPY/DAXPY/CAXPY/.../ZAXPYC Elementary Vector Operation	47
SAXPYI/DAXPYI/.../ZAXPYI Sparse Elementary Vector Operation	50
SCLIP/DCLIP/ICLIP Two-Sided Vector Clip	53
SCLIPL/DCLIPL/ICLIPL Left-Sided Vector Clip	56
SCLIPR/DCLIPR/ICLIPR Right-Sided Vector Clip	59
SCOPY/DCOPY/ICOPY/CCOPY/CCOPYC/.../ZCOPYC Copy Vector	62
SDOT/DDOT/CDOTC/CDOTU/ZDOTC/ZDOTU Dot Product	65
SDOTI/DDOTI/CDOTCI/CDOTUI/.../ZDOTUI Sparse Dot Product	69
SFRAC/DFRAC Extract Fractional Parts	72
SGTHR/DGTHR/IGTHR/CGTHR/ZGTHR Gather Sparse Vector	74
SGTHRZ/DGTHRZ/.../ZGTHRZ Gather and Zero Sparse Vector	76
SLSTxx/DLSTxx/ILSTxx/.../ZLSTxx List Selected Vector Elements	79
SMAX/DMAX/IMAX Maximum of Vector	83
SMIN/DMIN/IMIN Minimum of Vector	85
SNRM2/DNRM2/SCNRM2/DZNRM2 Euclidean Norm	87
SNRSQ/DNRSQ/SCNRSQ/DZNRSQ Euclidean Norm Squared	89
SRAMP/DRAMP/IRAMP Generate Linear Ramp	91
SROT/DROT/CROT/CSROT/ZROT/ZDROT Apply Givens Rotation	93
SROTG/DROTG/CROTG/ZROTG Construct Givens Rotation	97
SROTI/DROTI Apply Sparse Givens Rotation	99
SROTM/DROTM Apply Modified Givens Rotation	101
SROTMG/DROTMG Construct Modified Givens Rotation	104
SRSC/DRSC/CRSC/CSRSC/ZRSC/ZDRSC Scale Vector	107

SSCAL/DSCAL/CSCAL/CSSCAL/CSCALC/.../ZSCALC	Scale Vector	109
SSCTR/DSCTR/ISCTR/CSCTR/ZSCTR	Scatter Sparse Vector	112
SSUM/DSUM/ISUM/CSUM/ZSUM	Vector Sum	114
SSWAP/DSWAP/ISWAP/CSWAP/ZSWAP	Swap Two Vectors	116
SWDOT/DWDOT/CWDOTC/.../ZWDOTU	Weighted Dot Product	119
SZERO/DZERO/IZERO/CZERO/ZZERO	Clear Vector	123
3 Basic Matrix Operations		125
Overview		125
Chapter Objectives		125
What You Need to Know to Use These Subprograms		126
Subroutine Naming Convention		126
Subprograms Included in This Chapter		128
SGBMV/DGBMV/CGBMV/ZGBMV	Matrix-Vector Multiply	129
SGEMM/DGEMM/CGEMM/ZGEMM	Matrix-Matrix Multiply	134
SGEMMS/DGEMMS/.../ZGEMMS	Strassen Matrix-Matrix Multiply	138
SGEMV/DGEMV/CGEMV/ZGEMV	Matrix-Vector Multiply	143
SGER/DGER/CGERC/CGERU/ZGERC/ZGERU	Rank-1 Update	147
SSBMV/DSBMV/CHBMV/ZHBMV	Matrix-Vector Multiply	150
SSPMV/DSPMV/CHPMV/ZHPMV	Matrix-Vector Multiply	155
SSPR/DSPR/CHPR/ZHPR	Rank-1 Update	160
SSPR2/DSPR2/CHPR2/ZHPR2	Rank-2 Update	164
SSYMM/DSYMM/CEMM/.../ZSYMM	Matrix-Matrix Multiply	169
SSYMV/DSYMV/CEMV/ZHEMV	Matrix-Vector Multiply	173
SSYR/DSYR/CHER/ZHER	Rank-1 Update	177
SSYR2/DSYR2/CHER2/ZHER2	Rank-2 Update	180
SSYR2K/DSYR2K/CHER2K/CSYR2K/.../ZSYR2K	Rank-2k Update	184
SSYRK/DSYRK/CHERK/CSYRK/ZHERK/ZSYRK	Rank-k Update	189
STBMV/DTBMV/CTBMV/ZTBMV	Matrix-Vector Multiply	193
STBSV/DTBSV/CTBSV/ZTBSV	Solve Triangular Band System	198
STPMV/DTPMV/CTPMV/ZTPMV	Matrix-Vector Multiply	204
STPSV/DTPSV/CTPSV/ZTPSV	Solve Triangular System	208
STRMM/DTRMM/CTRMM/ZTRMM	Triangular Matrix-Matrix Multiply	213
STRMV/DTRMV/CTRMV/ZTRMV	Matrix-Vector Multiply	217
STRSM/DTRSM/CTRSM/ZTRSM	Solve Triangular Systems	221

Table of contents

STRSV/DTRSV/CTRSV/ZTRSV Solve Triangular System	225
XERBLA Error Handler	229
4 Linear Equations	231
Overview	231
Chapter Objectives	232
What You Need to Know to Use These Subprograms	232
Subroutine Naming Convention	232
Condition Number	234
Determinant and Inverse	235
LINPACK Subprograms Not in This Guide	235
Subprograms Included in This Chapter	236
SGBCO/DGBCO/CGBCO/ZGBCO Estimate Condition	237
SGBDI/DGBDI/CGBDI/ZGBDI Determinant	242
SGBFA/DGBFA/CGBFA/ZGBFA Band LU Factorization.....	246
SGBSL/DGBSL/CGBSL/ZGBSL Solve Band Linear Equations.....	251
SGECO/DGECO/CGECO/ZGECO Estimate Condition	254
SGEDI/DGEDI/CGEDI/ZGEDI Determinant and Inverse	257
SGEFA/DGEFA/CGEFA/ZGEFA LU Factorization.....	261
SGESL/DGESL/CGESL/ZGESL Solve Linear Equations.....	264
SGTSL/DGTSL/CGTSL/ZGTSL Solve Tridiagonal Linear Equations.....	268
SGTSLV/DGTSLV/CGTSLV/ZGTSLV Solve Tridiagonal Linear Equations	271
SPBCO/DPBCO/CPBCO/ZPBCO Estimate Condition	274
SPBDI/DPBDI/CPBDI/ZPBDI Determinant	278
SPBFA/DPBFA/CPBFA/ZPBFA Cholesky Factorization	281
SPBSL/DPBSL/CPBSL/ZPBSL Solve Linear Equations	285
SPOCO/DPOCO/CPOCO/ZPOCO Estimate Condition	288
SPODI/DPODI/CPODI/ZPODI Determinant and Inverse	291
SPOFA/DPOFA/CPOFA/ZPOFA Cholesky Factorization	295
SPOSL/DPOSL/CPOSL/ZPOSL Solve Linear Equations	298
SPTSL/.../ZPTSL Solve Positive Definite Tridiagonal Linear Equations	301
5 Eigenvalues and Eigenvectors.....	305
Overview	305
Chapter Objectives	306

What You Need to Know to Use These Subprograms	306
EISPACK Subprograms Not in This Guide	306
Subprograms Included in This Chapter	306
RS Eigenvalues and Eigenvectors of a Real Symmetric Matrix	307
TQL2 Eigenvalues and Eigenvectors of a Real Symmetric Matrix	310
TQLRAT Eigenvalues of a Real Symmetric Matrix	313
TRED1 Reduce Real Symmetric Matrix to Tridiagonal Form	316
TRED2 Reduce Real Symmetric Matrix to Tridiagonal Form	318
6 Sparse Linear Equations	321
Overview	321
Chapter Objectives	322
What You Need to Know to Use These Subprograms	322
Sparsity and Storage Formats	322
Direct Versus Iterative Solution	324
Fill and Reordering	324
Stability	326
Global Communications Array	326
Error Convention	327
Output Controls	327
Paths of Control	327
Sample Program	330
Subprograms Included in This Chapter	332
DSLEFS One-Call Usage	333
DSLEIN Initialize Sparse Linear Equations	338
DSLEI1 Matrix Structure Input by Single Entry	339
DSLEIC Matrix Structure Input by Column	340
DSLEIE Matrix Structure Input by Finite Element	342
DSLEIM Matrix Structure Input by Matrix	344
DSLEIF End of Matrix Structure Input	346
DSLEOR Reordering and Symbolic Factorization	347
DSLEV1 Matrix Value Input by Single Entry	349
DSLEVC Matrix Value Input by Column	350
DSLEVE Matrix Value Input by Finite Element	352
DSLEV1 Matrix Value Input by Matrix	354

Table of contents

DSLECO	Numeric Factorization and Condition Number Estimation.....	357
DSLEFA	Numeric Factorization	359
DSLESL	Solve	361
DSLEDA	Deallocate Working Storage.....	363
DSLEFF	Specify Singularity Treatment	364
DSLEOC	Output Control	366
DSLEOP	Select the Reordering Scheme	367
DSLEPS	Print Statistics.....	368
DSLERD	Return the Diagonal Elements of the Matrix in its Current Form ...	369
DSLERS	Restore Problem State from a Savefile.....	371
DSLESR	Retrieve Runtime Statistics	372
DSLESV	Save Problem State to a Savefile	373
7	Sparse Eigenvalues and Eigenvectors	375
	Overview	375
	Chapter Objectives	376
	What You Need to Know to Use These Subprograms	376
	Generalized Symmetric Eigenproblems	376
	Sparsity and Storage Formats	377
	Description of Sparse Eigenvalue Problems	379
	Trust Regions and Matrix Inertias (Sturm Sequence Counts)	380
	Convention for Returning Eigenvalues and Eigenvectors	381
	Global Communications Array	382
	Error Convention	382
	Output Controls	383
	Paths of Control	383
	Sample Program	385
	Subprograms Included in This Chapter	388
	DSEVE1 One Call Usage	389
	DSEVIN Initialize Sparse Eigenvalues/Eigenvectors.....	397
	DSEVI1 Matrix Structure Input by Single Entry	399
	DSEVIC Matrix Structure Input by Column.....	401
	DSEVIE Matrix Structure Input by Finite Element.....	403
	DSEVIM Matrix Structure Input by Matrix	405
	DSEVIF End of Matrix Structure Input	408

DSEVOR	Reordering and Symbolic Factorization	409
DSEVV1	Matrix Value Input by Single Entry	410
DSEVVC	Matrix Value Input by Column	412
DSEVVD	Matrix Value Input to Main Diagonal	414
DSEVVE	Matrix Value Input by Finite Element	416
DSEVVM	Matrix Value Input by Matrix	419
DSEVES	Eigenextraction	422
DSEVEX	Eigenextraction	427
DSEVRC	Return Eigenvalue/Eigenvector Results	434
DSEVRL	Return Eigenvalue Results	436
DSEVCK	Check Accuracy of Results	438
DSEVDA	Deallocate Working Storage	440
DSEVOC	Output Control	441
DSEVPS	Print Statistics	442
DSEVRS	Restore Problem State from a Savefile	443
DSEVSV	Save Problem State to a Savefile	444
8 Skyline Linear Equations		445
Overview		445
Chapter Objectives		446
What You Need to Know to Use These Subprograms		446
Sparsity and Storage Formats		446
Direct Versus Iterative Solution		449
Fill and Reordering		450
Stability		450
Global Communications Array		450
Error Convention		451
Output Controls		451
Paths of Control		451
Sample Program		455
Subprograms Included in This Chapter		456
DSKYFS One-Call Usage		457
DSKYFX One-Call Usage		460
DSKYIN Initialize Skyline Linear Equations		463
DSKYI1 Matrix Structure Input by Single Entry		464

Table of contents

DSKYIC	Matrix Structure Input by Column.	465
DSKYIE	Matrix Structure Input by Finite Element.	467
DSKYIM	Matrix Structure Input by Matrix.	469
DSKYIS	Matrix Structure Input by Skyline Matrix.	471
DSKYIF	End of Matrix Structure Input.	473
DSKYOR	Automatic Reordering.	474
DSKYOU	User-Supplied Reordering.	475
DSKYV1	Matrix Value Input by Single Entry.	476
DSKYVC	Matrix Value Input by Column.	478
DSKYVE	Matrix Value Input by Finite Element.	480
DSKYVM	Matrix Value Input by Matrix.	482
DSKYVS	Matrix Value Input by Skyline Matrix.	484
DSKYFA	Numeric Factorization.	486
DSKYSL	Solve Right-Hand Side Vector.	487
DSKYDF	Numeric Factorization.	489
DSKYDS	Solve Right-Hand Side Vector.	491
DSKYDA	Deallocate Working Storage.	493
DSKYOC	Output Control.	494
DSKYPS	Print Statistics.	495
DSKYRS	Restore Problem State from a Savefile.	496
DSKYSR	Retrieve Runtime Statistics.	497
DSKYSV	Save Problem State to a Savefile.	498

9 Fast Fourier Transforms. 499

Overview.	499
Chapter Objectives.	499
What You Need to Know to Use These Subprograms.	500
Subprograms Included in This Chapter.	500
C1DFFT/Z1DFFT One-Dimensional FFT.	501
S1DFFT/D1DFFT One-Dimensional FFT.	504
C2DFFT/Z2DFFT Two-Dimensional FFT.	507
S2DFFT/D2DFFT Two-Dimensional FFT.	509
C3DFFT/Z3DFFT Three-Dimensional FFT.	511
S3DFFT/D3DFFT Three-Dimensional FFT.	513
CFFTS/ZFFTS Simultaneous One-Dimensional FFT.	515

SFFTS/DFFTS	Simultaneous One-Dimensional FFT	519
CRC1FT/ZRC1FT	Real-to-Complex One-Dimensional FFT.....	523
SRC1FT/DRC1FT	Real-to-Complex One-Dimensional FFT	526
CRC2FT/ZRC2FT	Real-to-Complex Two-Dimensional FFT.....	529
SRC2FT/DRC2FT	Real-to-Complex Two-Dimensional FFT	532
CRC3FT/ZRC3FT	Real-to-Complex Three-Dimensional FFT	535
SRC3FT/DRC3FT	Real-to-Complex Three-Dimensional FFT	538
CRCFTS/ZRCFTS	Simultaneous Real-to-Complex 1-D FFT.....	542
SRCFTS/DRCFTS	Simultaneous Real-to-Complex 1-D FFT.....	547
10	Correlation and Convolution Subprograms	553
Overview		553
Chapter Objectives		553
What You Need to Know to Use These Subprograms		553
Subprograms Included in This Chapter		554
SCONV/DCONV	Correlation and Convolution.....	555
STCONV/DTCNV	Tapered Correlation and Convolution.....	559
11	Linear Recurrences.....	563
Overview		563
Chapter Objectives		563
What You Need to Know to Use These Subprograms		563
Subprograms Included in This Chapter		563
SFLR1C/DFLR1C	First Order Linear Recurrence	564
SFLR1M/SFLR1P/DFLR1M/DFLR1P	First Order Linear Recurrence.....	566
SFLR2C/DFLR2C	First Order Linear Recurrence	569
SFLR2M/SFLR2P/DFLR2M/DFLR2P	First Order Linear Recurrence.....	572
SFLRLM/.../DFLRLP	Last Term of First Order Linear Recurrence	576
SPPROD/DPPROD	Compute Vector of Partial Products.....	580
SPSUM/DPSUM/IPSUM	Compute Vector of Partial Sums.....	583
SSLR2/DSLRL2	Second Order Linear Recurrence	586
SSLR3/DSLRL3	Second Order Linear Recurrence	589
SSLRL/DSLRL	Last Term of Second Order Linear Recurrence	592
12	Miscellaneous Routines	597

Table of contents

Overview	597
Chapter Objective	597
What You Need to Know to Use These Subprograms	598
Subprograms Included in This Chapter	598
CPUTIME Measure CPU Time Used	599
DALLOC Deallocate Nonvolatile Dynamic Memory	600
MLIB_GETNUMTHREADS Determine Permitted Extent of Parallelism	602
MLIB_SETNUMTHREADS Control Extent of Parallelism	603
NALLOC Allocate Nonvolatile Dynamic Memory	604
RALLOC Reallocate Nonvolatile Dynamic Memory	607
RAN VAX-Compatible Random Numbers	610
RANV VAX-Compatible Random Numbers	612
SRAN/DRAN Scalar Long Period Random Number Generator	614
SRANV/DRANV Vector Long Period Random Number Generator	616
SSORT/DSORT/ISORT Sort Array	619
WALLTIME Measure Wallclock Time Used	621
XERVEC VECLIB Error Handler	622
Appendix A: Calling VECLIB from C	625
Introduction	625
General Interlanguage Programming Rules for VECLIB	625
Header Files	627
Examples	629
Calling Subroutines	629
Calling VECLIB Subroutines via an Interface Function	630
Calling Functions	632
REAL Functions	632
COMPLEX Functions	632
Linking VECLIB Subprograms into a C Program	633
Error Handling	633
Changing the Error Handler Signal	634
Supplying Your Own Signal Handler	635
Appendix B: Subprograms not in this Guide	637

Table of contents

LINPACK	637
EISPACK	641

Table of contents

Figures

Figure 6-1	Row and Column Index Sparse Matrix Representation	323
Figure 6-2	Column Pointer, Row Index Sparse Matrix Representation	324
Figure 6-3	Paths of Control	329
Figure 6-4	Paths of Control (continued)	330
Figure 7-1	Row and Column Index Sparse Matrix Representation	378
Figure 7-2	Column Pointer, Row Index Sparse Matrix Representation	379
Figure 7-3	Paths of Control	384
Figure 7-4	Paths of Control (continued)	385
Figure 8-1	Row and Column Index Sparse Matrix Representation	447
Figure 8-2	Skyline Matrix Representation	448
Figure 8-3	Reverse Skyline Matrix Representation	448
Figure 8-4	Column Pointer, Row Index Sparse Matrix Representation	449
Figure 8-5	Paths of Control	453
Figure 8-6	Paths of Control (continued)	454
Figure A-1	Calling VECLIB Subroutines from Fortran and C	630
Figure A-2	Matrix Multiplication via an Interface Function	631
Figure A-3	Calling a Real-Valued Function from C	632
Figure A-4	Calling a Complex-Valued Function from C	632
Figure A-5	Default VECLIB Error Handling in a C Program	634
Figure A-6	Changing the VECLIB Error Handling Signal	635
Figure A-7	Changing the Error Handling Signal Handler	636

List of Figures

Tables

Table 1-1	VECLIB Naming Convention—Data Type	11
Table 1-2	Data Item Byte Length vs. Data Type and Library	12
Table 1-3	Data Item Byte Length vs. Declaration and Compiler Option	13
Table 3-1	Extended BLAS Naming Convention — Data Type	126
Table 3-2	Extended BLAS Naming Convention — Matrix Form	126
Table 3-3	Extended BLAS Naming Convention — Computation	127
Table 3-4	Extended BLAS Naming Convention — Subprogram Names	127
Table 4-1	LINPACK Naming Convention — Data Type	232
Table 4-2	LINPACK Naming Convention — Form or Decomposition	233
Table 4-3	LINPACK Naming Convention — Computation	233
Table 4-4	LINPACK Naming Convention — Subprogram Names	234
Table A-1	Relationship Between Fortran and C Declarations	627
Table B-1	LINPACK Subprograms not in this Guide	637
Table B-2	EISPACK Subprograms not in this Guide	641

List of Tables

Preface

Purpose and Audience

This guide describes the VECLIB software library and shows how to use it. VECLIB, a component of MLIB, is a collection of Fortran-callable subprograms optimized for use on Hewlett-Packard supercomputers and workstations. This library provides mathematical software and computational kernels for application programs.

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

- Convert, develop, or optimize programs for use on supported computers
- Optimize existing software to improve performance and increase productivity

Organization

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

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

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 and appendices:

- Chapter 1 introduces general concepts about VECLIB.
- Chapter 2 describes basic vector operations included in VECLIB.
- Chapter 3 explains basic matrix operations.

- Chapter 4 describes linear equation subprograms in VECLIB.
- Chapter 5 explains eigenanalysis capabilities available to VECLIB users.
- Chapter 6 explains sparse symmetric linear equation subprograms.
- Chapter 7 describes sparse symmetric eigenvalue subprograms.
- Chapter 8 describes skyline linear equation subprograms.
- Chapter 9 describes the discrete Fourier transforms in VECLIB.
- Chapter 10 describes subprograms to compute convolutions and correlations of data sets.
- Chapter 11 includes subprograms that solve first and second order linear recurrences.
- Chapter 12 describes miscellaneous subprograms to produce random numbers, sort the elements of a vector in ascending or descending order, and convert numbers stored in Hewlett-Packard floating-point format to or from IBM floating-point format.
- Appendix A describes how to call VECLIB subprograms from within C programs.
- Appendix B lists the LINPACK and EISPACK subprograms that are not included in this guide.
- 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

f77 *prog_name.o*

instructs you to type the command *f77*, 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)

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

f77 prog_name.o

instructs you to type the command **f77**, followed by the name of a program or subprogram object file.

UPPERCASE CONSTANT WIDTH indicates Fortran programs.

Brackets (**[]**) enclose optional entries.

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 to help you:

- *HP MLIB LAPACK User's Guide* (B5649-90005). This guide provides information on the subprograms provided with the LAPACK library.
- *HP MLIB SCILIB User's Guide* (B5649-90006). This guide provides definitions for many additional subprograms available to VECLIB users through inclusion of SCILIB but not documented in the *HP MLIB VECLIB User's Guide*.
- *Exemplar Programming Guide: S-Class and X-Class Servers* (B5600-90001). This guide describes efficient programming techniques for the Exemplar family of computers.
- *Exemplar C and Fortran 77 Programmer's Guide: S-Class and X-Class Servers* (B5600-90002). This guide describe the Exemplar C and Fortran 77 compilers.

- *HP C/HP-UX Reference Manual* (92453-90024). This manual presents reference information on the C programming language as implemented by Hewlett-Packard.
- *HP C Programmer's Guide* (92434-90002). This guide contains detailed discussions of selected C topics.
- *Fortran/9000 Programmer's Reference* (B3906-90002). This book is a language reference for Hewlett-Packard Fortran 77.
- *Fortran/9000 Programmer's Guide* (B3906-90001). This manual is a task reference. It describes features and requirements in terms of the tasks a programmer might perform. These tasks include how to compile, link, run, debug, and optimize programs.
- *Programming on HP-UX* (B2355-90652). This book describes how to develop software on HP-UX using the HP compilers, assemblers, linker, libraries, and object files.
- *Exemplar Architecture: S-Class and X-Class Servers* (A4716-90001). This book describes the architectures of the S2000 and X2000 servers.
- *CXpa Reference: Exemplar S-Class and X-Class Servers* (B5639-90002). This guide explains the operation of the CXpa Performance Analyzer and the steps needed to create and interpret a CXpa profile.
- *CXdb Quick Reference: Exemplar S-Class and X-Class Servers* (B5639-90007). This reference describes prominent features of the CXdb visual debugger.
- *CXtrace User's Guide: Exemplar S-Class and X-Class Servers* (B5639-90003). This guide describes CXtrace, a trace-based performance analysis tool for C and Fortran 77 MPI and PVM applications.
- *VAST/f90 User's Guide* (B5610-90001). This guide describes the VAST/f90 compiling system. VAST/f90 translates Fortran 90 programs into Fortran 77, which is then compiled by an HP Fortran 77 compiler.
- *HP MPI User's Guide* (B6011-90001). This book discusses message-passing programming using the Message-Passing Interface library.
- *HP PVM User's Guide* (B5885-90001). This book discusses message-passing programming using the Parallel Virtual Machine library.
- *HP Fortran 90 Programmer's Reference* (B5876-90001). This book is a complete Fortran 90 language reference. It also covers compiler options, compiler directives, and library information.
- *HP Fortran 90 Programmer's Notes* (B5876-90002). This book provides extensive usage information, including how to compile and link, suggestions and tools for migrating to HP Fortran 90, and how to call C and HP-UX routines from HP Fortran 90.

- *Exemplar C++ Programming Guide: S-Class and X-Class Servers* (B5630-90001). This book describes the Exemplar C++ compiler.
- *HP-UX Assembly Language Reference Manual* (92432-90001). This manual describes the HP-UX assembler for the PA-RISC processor.
- *HP PA-RISC 2.0 Architecture Reference* (B5655-90009). This manual describes the architecture of the Hewlett-Packard PA-RISC 2.0 processor.
- *PA-RISC Procedure Calling Conventions Reference* (09740-90015). This manual describes the conventions for creating PA-RISC assembly language procedure calls.

Ordering Documentation

To order additional copies of this document or other documents listed in 'Associated Documents,' send requests to:

Hewlett-Packard Company
Convex Division
Customer Service
P.O. Box 833851
Richardson, TX 75083-3851 USA

Please include the order number (xxxxx-9xxxx) or the exact title of the document.

Technical Assistance

If you have questions that are not answered in this book, contact the Hewlett-Packard Convex Technical Assistance Center (TAC) at the following locations:

- Within the continental U.S., call 1 (800) 952-0379.
- From Canada, call 1 (800) 345-2384.
- All other locations, contact your local Hewlett-Packard office.

Technical Assistance

You can also use the contact utility if you would like to report any problems you may have with HP MLIB LAPACK or its associated documentation.

1 Introduction to VECLIB

Overview

VECLIB, a component of HP MLIB, is a collection of Fortran-callable subprograms optimized for use on Hewlett-Packard supercomputers and workstations, providing mathematical software and computational kernels for application programs involving arrays. This library contains subprograms for:

- Dense vector operations, including the Basic Linear Algebra Subprograms (BLAS)
- Sparse vector operations, including the Sparse BLAS
- Matrix operations, including the Level 2 and Level 3 BLAS
- Linear equation solution, including LINPACK
- Eigensystem solution, including EISPACK
- Sparse symmetric linear equation solutions
- Sparse symmetric ordinary and generalized eigensystem solutions
- Skyline linear equations
- Discrete Fourier transforms
- Convolution and correlation
- Linear recurrences
- Miscellaneous tasks, such as sorting and generating random numbers

Although VECLIB was designed for use with Fortran programs, C programs can call VECLIB subprograms, as described in Appendix A.

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

- conformance to various public-domain standards
- the VECLIB and VECLIB8 library files
- accessing VECLIB subprograms
- optimizations, including parallel processing and interactions with other Hewlett-Packard analysis and optimization products

- supported floating-point formats
- roundoff effects
- the naming convention
- how to use the two libraries and various compiler options
- error handling
- online documentation
- Hewlett-Packard support services

The following documents provide supplemental material for this chapter:

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.

Chapter Objectives

After reading this chapter you will:

- Know why there are two libraries and how to access them
- Understand how VECLIB works in a parallel computing environment
- Know how VECLIB interacts with the CXpa Performance Analyzer and other profilers and architecture-specific features
- Know some VECLIB naming conventions
- Understand roundoff effects
- Know how to use Fortran type declarations and compiler options
- Understand how VECLIB handles errors
- Know how to access the online *HP MLIB* Man Pages
- Know what to do if you are having trouble using VECLIB subprograms

What You Need to Know to Use VECLIB

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

Standardization

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

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

Two VECLIB Libraries

Often, it is desirable to run a single-precision program in double precision. To support changing the precision without changing the code, the Hewlett-Packard Fortran compilers provide two compilation options, `+autodbl` and `+autodbl4`, that affect the size of Fortran data types. For compatibility with these compiler options, VECLIB provides two libraries, VECLIB and VECLIB8.

- The VECLIB library works with default-sized Fortran INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and LOGICAL data types, that is, the sizes you get when you do not use the `*n` size specifications or either of the `+autodbl` or `+autodbl4` compiler options.
- The VECLIB8 library is a subset of the VECLIB library that is designed for use by programs that are compiled with either the `+autodbl` or `+autodbl4` compiler options. The VECLIB8 library is available only on computer systems with compilers that support the INTEGER*8 data type.

For more information about these options, refer to a Fortran language reference and the "Required Data Item Byte Lengths and How to Get Them" section in this chapter. To determine if a subprogram is included in VECLIB8, consult the VECLIB8 section under each subprogram specification in the other chapters.

Accessing VECLIB

The VECLIB and VECLIB8 libraries consist 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 VECLIB or VECLIB8 be used as an object library at link time.

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

System type	CPU Type	Installation Directory
C-, D-, or K-Class	PA-8000	/opt/mlib/lib/pa2.0
S- or X-Class	PA-8000	/opt/mlib/lib/pa2.0parallel
C-, D-, J-, or K-Class	PA-7200	/opt/mlib/lib/pa1.1
SPP-1200 or SPP-1600	PA-7200	/opt/mlib/lib/pa1.1parallel

The file names of the VECLIB and VECLIB8 libraries are *libveclib.a* and *libveclib8.a*, respectively.

There are several ways to specify the linking of your program with VECLIB or VECLIB8:

- Specify the entire path of the library file on the f77 or f90 command line that links your program. For example, with the f77 compiler on a C-Class PA-8000 system, use:

f77 [options] file /opt/mlib/lib/pa2.0/libveclib.a

- Use the -l option on the f77 or f90 command line that links your program, preceded by the option -Wl, -L<path>, where <path> is the appropriate installation directories. For example, the previous f77 command line could be written as:

f77 [options] file -Wl, -L/opt/mlib/lib/pa2.0 -lveclib

- Set the LDOPTS environment variable to include -L<path>, where <path> is the appropriate installation directory, and use the -l option on the f77 or f90 command line that links your program. For example, with the f90 compiler, use:

f90 [options] file -lveclib

or

f90 [options] file -lveclib8

Do not use subprograms from both -lveclib and -lveclib8 in the same program.

NOTE

If you are running MLIB on an S- or X-Class machine, the VECLIB, SCILIB, and LAPACK libraries contain parallelized subprograms. See the “Parallel Processing” section for additional information about linking programs.

LAPACK is documented in the *HP MLIB LAPACK User's Guide*. If your program uses subprograms from both VECLIB and LAPACK, specify both **-lveclib** and **-llapack**, or both **-lveclib8** and **-llapack8**. For example, using the second method above to specify the location of the libraries, link with

```
f77 [options] file -lveclib -llapack
```

or

```
f90 [options] file -lveclib8 -llapack8
```

See “Interactions Between VECLIB, SCILIB, and LAPACK” for details about how to order the two **-l** options. Do not use subprograms from both **-lveclib** and **-llapack8** or both **-lveclib8** and **-llapack** in the same program.

SCILIB is documented in the *HP MLIB SCILIB User's Guide*. If you use subprograms from both VECLIB and SCILIB, specify both libraries when you link your program. For example, using method 2 above on an S-Class machine, link with

```
f90 [options] file -Wl, -L/opt/mlib/lib/pa2.0parallel -lveclib8 -lscilib
```

Add the linker option **-llapack8** if LAPACK subprograms are 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 **-lveclib** and **-lscilib** in the same program.

Interactions Between VECLIB, SCILIB, and LAPACK

Each library file in VECLIB, SCILIB, and LAPACK is complete in itself. This is accomplished by including various subprograms in more than one library. You need not load one library merely because you have used subprograms from another. For example, subroutine SGEMV is in each product, but with identical functionality. Load only the libraries you need and list them in any order on your load command line, as described in the previous section. However, a few differences between the libraries may force you to put the libraries into a specific order to obtain the results you expect.

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, knowing the architecture of the target computer has become more important in order 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. Using VECLIB helps you achieve good performance at low cost.

Optimization

VECLIB is highly efficient. It takes full advantage of the special features of your computer's architecture. The VECLIB libraries provide mathematical software and computational kernels for programs involving arrays, which include dense and sparse vectors, matrices, and array sections. Most VECLIB subprograms have been coded in assembly language, but 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 your computer's architecture or assembly-language programming. In this respect, VECLIB is an extension of the Fortran language.

Parallel Processing

Parallel processing is available on Hewlett-Packard S- and X-Class computer systems. 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.

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

Linking for Parallel or Non-Parallel Processing

To enable parallel processing at link time, your link step must produce a multithreaded executable. You always get a multithreaded executable if you link with the Fortran 77, C, or C++ compiler using the **+Oparallel** flag:

f77 [*options* including **+O3 +Oparallel**] *file* **-lveclib**

Fortran 90, on the other hand, will not produce a multithreaded executable unless you include the **-Wl,+tm,S2000,+parallel** or **-Wl,+tm,X2000,+parallel** option on the f90 command line that links your program:

f90 [*options* including **-Wl,+tm,S2000,+parallel**] *file* **-lveclib -lpthread -lcps -lpthread -lail**

To disable VECLIB's automatic parallelism at link time, omit the **+Oparallel** or **+parallel** option and include the following required runtime libraries:

f77 [*options*] *file* **-lveclib -lpthread -lcps -lpthread -lail**

and

f90 [*options* including **-Wl,+tm,S2000**] *file* **-lveclib -lpthread -lcps -lpthread -lail**

Another alternative is to link with or without parallelism enabled and use the *mpa*(1) utility to modify the attributes of the executable file to disable or enable parallelism. Refer to the *mpa*(1) man page for details.

Controlling VECLIB Parallelism at Run Time

If you simply compile and link your program, parallelized VECLIB subprograms will execute on a single processor even though the executable is marked for parallel execution. Two methods are provided to enable parallel processing within VECLIB at run time. First, a shell environment variable, **MLIB_NUMBER_OF_THREADS**, allows you to enable parallelism within VECLIB subprograms and to specify the maximum number of threads that may be used in parallel regions. Not setting **MLIB_NUMBER_OF_THREADS** has the same result as setting it to 1—parallel processing is disabled within VECLIB subroutines. Setting it to a value greater than the number of threads

in the subcomplex allows parallelized VECLIB subprograms to use as many CPUs as are available to the process.

The following command lines show the C shell syntax and Korn shell syntax to use when setting the variable to 8 processors:

C shell: **setenv MLIB_NUMBER_OF_THREADS 8**

Korn shell: **export MLIB_NUMBER_OF_THREADS=8**

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

The second method you can use to control parallelism within VECLIB is the subroutine **MLIB_SETNUMTHREADS**. You can call this subroutine at any time to set the maximum number of parallel threads used in subsequent VECLIB, SCILIB, or LAPACK calls. The specified value overrides the absence of the **MLIB_NUMBER_OF_THREADS** environment variable or any value assigned to it. Additionally, you can use **MLIB_SETNUMTHREADS** to restore VECLIB parallel processing to its run-time default. Refer to the *mllib_setnumthreads(3m)* man page for usage information.

Finally, in addition to the above VECLIB controls, at run time you can use the *mpa(1)* utility or the **MP_NUMBER_OF_THREADS** environment variable to control parallelism. These controls set the maximum amount of parallelism that your program can use, and the VECLIB-specific mechanisms offer finer control within that maximum. Refer to the *mpa(1)* or *f77(1)* man page, respectively, for details.

VECLIB Parallel Processing

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

If you are using an S- or X-Class system, you can realize the performance benefits of parallel processing in three ways:

- Call any parallelized VECLIB subprogram. Let it use parallelism internally if it determines that it is appropriate to do so, based on such factors as problem size, system configuration, and user environment.
- Call VECLIB subprograms in a parallelized loop or region. To use this mechanism, you must be familiar with the techniques of parallel processing. Refer to the *Exemplar Programming Guide: S-Class and X-Class Servers* for details.

- Use the MPI explicit parallel model. See the *MPI(1)* man page for details.

VECLIB subprograms are reentrant. This 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 VECLIB subprograms in a parallelized loop or region. The compiler does not automatically parallelize loops containing a function reference or subroutine call. You can force it to parallelize such a loop by inserting compiler directives before the loop. For example, the following Fortran code makes parallel calls to subprogram SAXPY:

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

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

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

Profiling VECLIB Applications

The CXpa Performance Analyzer is an interactive tool for HP computer systems 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 CXpa, you must first compile your program with the *+pa* or *+pal* compiler option. This option instruments the compiled program such that its performance can be measured at the subprogram level, the loop level, or the parallelized loop level.

VECLIB has been instrumented for use with CXpa such that CXpa will not automatically collect or analyze performance information for VECLIB subroutines. However, you can use CXoi, the Exemplar PA-RISC Object and Archive File Instrumenter, to inset CXpa instrumentation into the VECLIB library. When you use these instrumented libraries, the performance of VECLIB subprograms can be included in the analysis. See *cxoi(1)* for more information.

CXpa is an optional product. For more information about CXpa, refer to the *CXpa Reference: Exemplar S-Class and X-Class Servers*, or contact your HP sales representative.

Floating-Point Formats

Hewlett-Packard Exemplar and other computers with PA-RISC-based architectures operate on floating-point data in the IEEE format. For further information on Hewlett-Packard floating-point formats, refer to the *Fortran User's Guide*.

Roundoff Effects

VECLIB subprograms may use a different arithmetic order of evaluation than other implementations. Different roundoff characteristics may result. Accuracy of results is usually about the same, so using VECLIB 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 indicate 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 VECLIB differ materially, both sets of answers are probably inaccurate and you should investigate further. If the program correctly applies stable computational algorithms, the problem itself is probably ill-posed.

Data Types and Precision

In general, VECLIB provides the same range of functionality for both real and complex data. For most computations, there are matching subprograms, for real data and for complex data, but there are a few exceptions. For example, corresponding to the subprograms for real dot products, there are subprograms for complex dot products in both the conjugated and unconjugated forms because both types of complex dot products occur. However, there is no complex analogue of the subprograms for solving a real symmetric sparse linear system. Matching subprograms for real and complex data have been coded to maintain a close correspondence between the two, wherever possible; but in some areas the correspondence is necessarily weaker.

Most subprograms in VECLIB are provided in both 32-bit and 64-bit precision versions. All VECLIB8 subprograms use only 64-bit precision.

VECLIB Naming Convention

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

Table 1-1 VECLIB Naming Convention—Data Type

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

For example, subprograms that compute the sum of vector elements are named according to data type: SSUM, DSUM, ISUM, CSUM, and ZSUM for the VECLIB library, and SSUM, ISUM, and CSUM for the VECLIB8 library. Some function subprograms use two of these letters, the first describing the data type of the function and the second indicating the type of data on which it operates.

Required Data Item Byte Lengths and How to Get Them

There is a relationship between the data type of a subprogram, designated by the first character of its name, which is denoted by T in Table 1-1, and the byte lengths of its arguments. This relationship, which differs between the VECLIB and VECLIB8 libraries, is shown in Table 1-2:

Table 1-2 Data Item Byte Length vs. Data Type and Library

	VECLIB Argument Lengths			VECLIB8 Argument Lengths		
T	INTEGER LOGICAL	REAL	COMPLEX	INTEGER LOGICAL	REAL	COMPLEX
S	4	4	8	8	8	16
D	4	8	16	†	†	†
C	4	4	8	8	8	16
Z	4	8	16	†	†	†

† - the D and Z subprograms are not included in VECLIB8.

As mentioned in "Two VECLIB Libraries," it often is desirable to run a 32-bit precision program with 64 bits of precision. To support changing the precision without changing the code, Hewlett-Packard Fortran compilers provide two compilation options, namely, `+autodbl` and `+autodbl4`, that affect the size of Fortran data types. By taking care in your use of Fortran data type declarations, you can write a program that will compile with one set of compiler options and run correctly in 32-bit precision with the VECLIB library or compile with another set of compiler options and run correctly in 64-bit precision with VECLIB8. One constraint is that the program must not use any D or Z subprograms from VECLIB because the D and Z subprograms are not included in VECLIB8. (Note that a program that calls D or Z VECLIB subprograms would not be a 32-bit program.)

Another scenario is that you might be porting a program from a computer with default 8-byte integer and real data items, and you want to use 4-byte integers while continuing to use 8-byte reals. A program change is required, since the original program would be using the S and C VECLIB subprograms, while the HP version would have to use the D and Z subprograms from the VECLIB library, because they are the only ones that combine 4-byte integers with 8-byte reals. The C preprocessor `#define` statement may be an appropriate conversion tool. Or, for example, the *f90* command line options `+cpp -DSDOT=DDOT` `-Dsdot=ddot` may be used to translate all SDOT references to DDOT. You can deal with constants by replacing them with variables or using the Fortran `PARAMETER` statement.

Table 1-3 shows how the lengths of data items depends on their declarations and the compiler options used.

Table 1-3 Data Item Byte Length vs. Declaration and Compiler Option

Fortran Declaration	Fortran 90 Compiler Option		
	none	+autodbl	+autodbl4
INTEGER	4	8	8
INTEGER*4	4	4	4
INTEGER*8	8	8	8
Integer by default	4	8	8
REAL	4	8	8
REAL*4	4	4	4
REAL*8	8	8	8
Real by default	4	8	8
DOUBLE PRECISION	8	16	8
Double Precision constant	8	16	8
COMPLEX	8	16	16
COMPLEX*8	8	8	8
COMPLEX*16	16	16	16
Complex constant	8	16	16
DOUBLE COMPLEX	16	16	16
Double Complex constant	16	16	16
LOGICAL	4	8	8
LOGICAL*4	4	4	4
LOGICAL*8	8	8	8
Logical constant	4	8	8

By comparing Tables 1-2 and 1-3, you can see that if the Fortran data types are not given length specifiers (for example, **REAL** is used instead of **REAL*4**) and neither of the compiler options, **+autodbl** or **+autodbl4**, is used, then the VECLIB library is type compatible for the I, S, C prefixes and also for D if the type is **DOUBLE PRECISION**. On the other hand, if no length specifiers are used and one of these compiler options is chosen, VECLIB8 is type-compatible for the I, S, and C prefixes. This provides the easiest interchangeability between 32-bit and 64-bit execution.

In cases of mixed data types, you must choose the correct VECLIB library and subprogram carefully. In particular, VECLIB8 accepts no **INTEGER*4** arguments. For more information on Fortran data types and Fortran compiler options, refer to the *Fortran Language Reference*.

Error Handling

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

- Low-level subprograms
- High-level subprograms

Low-level Subprograms

Low-level subprograms are only minimally capable of detecting or handling errors. These subprograms attempt to do what is reasonable when a usage error occurs, but they do not attempt to warn you that something is wrong. For example, function SDOT, which computes the dot-product of two dense real vectors of length N , gives the mathematically proper result, zero, when $N \leq 0$. This is considered mathematically correct because, by definition, an empty sum is zero. Because SDOT conforms to the published BLAS argument list and usage, however, SDOT does not notify you that you may have made a mistake. To notify you would add severe usage conflicts with codes that already use the BLAS. Because these low-level subprograms do what is reasonable, you can simplify a program and maybe even speed it up by using VECLIB subprograms without checking for special cases. For example, relying on SDOT having a zero result for $N \leq 0$ may eliminate an IF statement that would take the same branch almost all the time through a loop.

High-level Subprograms

High-level subprograms, on the other hand, are capable of detecting and reporting errors. These subprograms usually do more work than the low-level subprograms, so the relative expense of checking and reporting errors in great detail may be extremely small. For example, some errors are

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

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

The convention used for `ier` is

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

Some VECLIB subprograms do not have a success/error code in their argument lists, but instead call another VECLIB subprogram to process the error condition. Two error handlers are provided: XERBLA and XERVEC; these are documented in Chapter 3 and Chapter 12 of this guide, respectively. The documentation for each VECLIB subprogram indicates if either of these error handlers is used. The standard versions of XERBLA and XERVEC write an error message onto the standard error file. If you are running on a C Series system and your 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 XERVEC that alters this action; see the documentation for these subprograms for more information.

The description of each high-level subprogram defines the specific error-code numbers and the related error conditions when *ier* $\neq$ 0. Always check the output argument *ier* after calling a VECLIB subprogram that has it as an argument and take appropriate action should the output argument suggest a problem.

HP MLIB Man Pages

The *HP MLIB* man pages contain online documentation that includes information from the user's guides. They contain an introduction to VECLIB and to each set of subprograms in VECLIB and reference entries for each subprogram.

Because of the limited number of fonts supported and the difficulty of presenting mathematical equations in the *man*(1) system, the *HP MLIB* man pages are not a substitute for the user's guides; detailed information on VECLIB is in the user's guides.

The *HP MLIB* man pages are installed in the directory `/opt/mlib/share/man`. You must have this path in your MANPATH environment variable to access man pages for VECLIB, SCILIB, or LAPACK.

To access the *HP MLIB* man pages, use the shell command

```
man mlib
```

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

```
man veclib.
```

Support Services

Hewlett-Packard maintains a staff to provide technical help if you have difficulty. Located in the Hewlett-Packard 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 VECLIB has been tested extensively and is very reliable. Therefore, before contacting

the TAC about a VECLIB 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, VECLIB may produce a different answer or give the answers in a different order than expected. If the problem is ill-conditioned, VECLIB may not be able to compute a reliable answer 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.

2 Basic Vector Operations

Overview

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

- The Basic Linear Algebra Subprograms (BLAS)
- The Sparse BLAS
- Hewlett-Packard extensions to the BLAS

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

The following documents provide supplemental material for this chapter:

Dodson, D.S., R.G. Grimes, and J.G. Lewis. "Sparse Extensions to the Fortran Basic Linear Algebra Subprograms." *ACM Transactions on Mathematical Software*. June, 1991. Vol. 17, No. 2.

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

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

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 computational- expensive operations of linear algebra. Even though you can code most of these operations in fewer than 10 lines of Fortran, using VECLIB 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 VECLIB you will normally be replacing the standard Fortran BLAS with the VECLIB BLAS and increasing the efficiency of LINPACK on your Hewlett-Packard supercomputer.

You need not limit your use of the VECLIB 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 $N3 = N1 \times N2$. 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 , $i = 1, 2, \dots, N$, 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

What You Need to Know to Use These Subprograms

components x_1, x_2, x_3 , and x_4 are stored in the array elements $X(1), X(3), X(5)$, and $X(7)$, respectively.

Backward Storage. Some BLAS subprograms permit the backward storage of vectors, which is specified by using a negative INCX. If $INCX < 0$, then x_i is stored in $X(1 + (N-i) \times |INCX|)$ 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 subprograms 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 is $I(N, B(I, 1), LDB)$.

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

This usage 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*4 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*4 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*4 SDOT, A(LDA,3), X(3), Y(LDA)
N = 3
Y(2) = SDOT (N, A(2,1), LDA, X,1)

```

Subprograms Included in This Chapter

Following are the vector subprograms included with VECLIB.

Name ISAMAX/IDAMAX/.../IZAMAX
Index of Maximum of Magnitudes

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

$$|x_i| = \max(|x_j| : j = 1, 2, \dots, n).$$

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

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

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

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

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

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

Usage VECLIB:

```

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

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

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

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

```

```

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

```

VECLIB8:

```

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

```

```

INTEGER*8      i, IIMAX, n, incx, x(lenx)
i = IIMAX(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 $n \leq 0$, the subprograms do not reference x .
	x	Array of length $\text{lenx} = (n-1) \times \text{incx} + 1$ containing the n -vector x .
	incx	Increment for the array x . x is stored forward in array x with increment $ \text{incx} $; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$. Use $\text{incx} = 1$ 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 element of x of maximum magnitude.
Notes	The handling of $\text{incx} < 0$ differs between ISAMAX in VECLIB and SCILIB.	

**Fortran
Equivalent**

```

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

```

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

```

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

```


Name ISAMIN/IDAMIN/.../IZAMIN
Index of Minimum of Magnitudes

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

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

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

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

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

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

This definition is not used because of computational speed.

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

Usage VECLIB:

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

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

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

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

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

VECLIB8:

```

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

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

      INTEGER*8      i, ICAMIN, n, incx
      COMPLEX*16     x(lenx)
      i = ICAMIN(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 $\text{lenx} = (n-1) \times \text{incx} + 1$ containing the n -vector x .
	incx	Increment for the array x . x is stored forward in array x with increment $ \text{incx} $; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$. Use $\text{incx} = 1$ 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 element of x of minimum magnitude.
Notes	The handling of $\text{incx} < 0$ differs between ISAMIN in VECLIB and SCILIB.	

**Fortran
Equivalent**

```

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

```

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

```

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

```

Name ISCTxx/IDCTxx/IICTxx/.../ZCTxx
Count Selected Vector Elements

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

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

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

For complex subprograms, these characters and corresponding function values are

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

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

Usage VECLIB:

```

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

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

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

```

```

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

```

```

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

```

VECLIB8:

```

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

```

```

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

```

```

INTEGER*8      i, ICCTxx, n, incx
COMPLEX*16     a, x(lenx)
i = ICCTxx(n, x, incx, a)

```

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 $\text{lenx} = (n-1) \times \text{incx} + 1$ 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 $x((i-1) \times \text{incx} + 1)$. Use $\text{incx} = 1$ 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 $n \leq 0$, then $i = 0$. Otherwise, i is the number of elements of x that satisfy the relationship with a specified by the subprogram name.

**Fortran
Equivalent**

```

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

```

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

```

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

```

Name ISMAX/IDMAX/IIMAX

Index of Maximum Element of Vector

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

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

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

Usage VECLIB:

INTEGER*4 i , ISMAX, n , $incx$

REAL*4 $x(lenx)$

$i = \text{ISMAX}(n, x, incx)$

INTEGER*4 i , IDMAX, n , $incx$

REAL*8 $x(lenx)$

$i = \text{IDMAX}(n, x, incx)$

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

$i = \text{IIMAX}(n, x, incx)$

VECLIB8:

INTEGER*8 i , ISMAX, n , $incx$

REAL*8 $x(lenx)$

$i = \text{ISMAX}(n, x, incx)$

INTEGER*8 i , IIMAX, n , $incx$, $x(lenx)$

$i = \text{IIMAX}(n, x, incx)$

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 $lenx = (n-1) \times |incx| + 1$ 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 $x((i-1) \times |incx| + 1)$.

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 this chapter.

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

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

**Fortran
Equivalent**

```

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

```

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

```

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

```


Name ISMIN/IDMIN/IIMIN

Index of Minimum Element of Vector

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

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

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

Usage VECLIB:

INTEGER*4 i , ISMIN, n , incx

REAL*4 $x(\text{lenx})$

$i = \text{ISMIN}(n, x, \text{incx})$

INTEGER*4 i , IDMIN, n , incx

REAL*8 $x(\text{lenx})$

$i = \text{IDMIN}(n, x, \text{incx})$

INTEGER*4 i , IIMIN, n , incx, $x(\text{lenx})$

$i = \text{IIMIN}(n, x, \text{incx})$

VECLIB8:

INTEGER*8 i , ISMIN, n , incx

REAL*8 $x(\text{lenx})$

$i = \text{ISMIN}(n, x, \text{incx})$

INTEGER*8 i , IIMIN, n , incx, $x(\text{lenx})$

$i = \text{IIMIN}(n, x, \text{incx})$

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

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

Use $\text{incx} = 1$ 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 VECLIB and SCILIB.

**Fortran
Equivalent**

```

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

```

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

```

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

```

Name ISSVxx/IDSVxx/IISVxx/ICSVxx/IZSVxx
Search Vector for Element

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

The last two characters of the subprogram name specify the relationship of interest between the element of the vector and the scalar. For real and integer subprograms, these characters, represented by "xx" in the prototype Fortran statements, and the corresponding function values 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	$\min\{i : x_i \neq a\}$

For complex subprograms, these characters and corresponding function values are

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

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

Usage

VECLIB:

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

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

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

```

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

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

```

VECLIB8:

```

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

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

INTEGER*8      i, ICSVxx, n, incx
COMPLEX*16     a, x(lenx)
i = ICSVxx(n, x, incx, a)

```

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 $\text{lenx} = (n-1) \times \text{incx} + 1$ containing the n -vector x .
	incx	Increment for the array x . x is stored forward in array x with increment $ \text{incx} $; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$. Use $\text{incx} = 1$ 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 $n \leq 0$ or if no element of x satisfies the relationship with a specified by the subprogram name, then $i = 0$. Otherwise, i is the index i of the first element x_i of x that satisfies the relationship with a specified by the subprogram name. Recall that x_i is stored in $x((i-1) \times \text{incx} + 1)$.

**Fortran
Equivalent**

```

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

```

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

```

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

```

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

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

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

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

$$s = \max(|\operatorname{Re}(x_i)| + |\operatorname{Im}(x_i)| : i = 1, 2, \dots, n).$$

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

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

s is computed instead of t since it is faster because it does not require any 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

VECLIB:

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

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

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

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

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

VECLIB8:

```

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

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

      INTEGER*8      n, incx
      REAL*8         s, SCAMAX
      COMPLEX*16     x(lenx)
      s = SCAMAX(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 $\text{lenx} = (n-1) \times |\text{incx}| + 1$ containing the n -vector x .

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

Use $\text{incx} = 1$ 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 $n \leq 0$, then $s = 0$. Otherwise, s is the maximum of the magnitudes of the elements of x .

**Fortran
Equivalent**

```

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

```

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

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


Name SAMIN/DAMIN/IAMIN/SCAMIN/DZAMIN
Minimum of Magnitudes

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

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

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

$$s = \min(|\operatorname{Re}(x_i)| + |\operatorname{Im}(x_i)| : i = 1, 2, \dots, n)$$

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

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

s is computed instead of t since it is faster because it does not require any 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 VECLIB:

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

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

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

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

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

VECLIB8:

```

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

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

      INTEGER*8      n, incx
      REAL*8         s, SCAMIN
      COMPLEX*16     x(lenx)
      s = SCAMIN(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 $\text{lenx} = (n-1) \times \text{incx} + 1$ containing the n-vector x. incx Increment for array x. x is stored forward in array x with increment incx; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$. Use $\text{incx} = 1$ 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 $n \leq 0$, then $s = \infty$, the largest representable machine number. Otherwise, s is the minimum of the magnitudes of the elements of x.

**Fortran
Equivalent**

```

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

```

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

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

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

Purpose Given a real or integer vector x of length n , SASUM, DASUM, or IASUM computes the l_1 norm of x , 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 or DZASUM computes

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

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

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

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

VECLIB:

```

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

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

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

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

```

```

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

```

VECLIB8:

```

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

INTEGER*8      n, incx, s, IASUM, x(lenx)
s = IASUM(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 $n \leq 0$, the subprograms do not reference x .

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

incx Increment for the array x . x is stored forward in array x with increment $|\text{incx}|$; i.e., x_i is stored in $x((i-1) \times |\text{incx}| + 1)$.
Use $\text{incx} = 1$ 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 $n \leq 0$, then $s = 0$. Otherwise, s is the sum of magnitudes of the elements of x .

**Fortran
Equivalent**

```

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

```

Example

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

```

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

```

Name SAXPY/DAXPY/CAXPY.../ZAXPYC
Elementary Vector Operation

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

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

where $\bar{x}$ is the complex conjugate of x . The vectors 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 VECLIB:

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

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

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

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

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

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

VECLIB8:

```
INTEGER*8      n, incx, incy
REAL*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)
```

```

INTEGER*8      n, incx, incy
COMPLEX*16     a, x(lenx), y(leny)
CALL CAXPYC(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 $n \leq 0$, the subprograms do not reference x or y .
	a	The scalar a .
	x	Array of length $\text{lenx} = (n-1) \times \text{incx} + 1$ containing the n -vector x . x is used in conjugated form by CAXPYC and ZAXPYC and in unconjugated form by the other subprograms. Refer to "Purpose."
	incx	Increment for the array x : $\text{incx} \geq 0$ x is stored forward in array x; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$. $\text{incx} < 0$ x is stored backward in array x; i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$. Use $\text{incx} = 1$ 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 $\text{leny} = (n-1) \times \text{incy} + 1$ containing the n -vector y .
	incy	Increment for the array y , $\text{incy} \neq 0$: $\text{incy} > 0$ y is stored forward in array y; i.e., y_i is stored in $y((i-1) \times \text{incy} + 1)$. $\text{incy} < 0$ y is stored backward in array y; i.e., y_i is stored in $y((i-n) \times \text{incy} + 1)$. Use $\text{incy} = 1$ 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 $n \leq 0$ or $a = 0$, then y is unchanged. Otherwise, $ax+y$ overwrites the input.

Notes If $\text{incx} = 0$, then $x_i = x(1)$ for all i .

The result is unspecified if $\text{incy} = 0$ or if $\mathbf{x}$ and $\mathbf{y}$ overlap such that any element of $\mathbf{x}$ shares a memory location with any element of $\mathbf{y}$.

**Fortran
Equivalent**

```

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

```

Example 1 Compute the REAL*8 elementary vector operation

$$y \leftarrow 2x + y,$$

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

```

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

```

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

```

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

```

Name SAXPYI/DAXPYI/.../ZAXPYI
Sparse Elementary Vector Operation

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

$$y \leftarrow ax + y.$$

More precisely, let x be a sparse n -vector with $m \leq n$ *interesting* (usually nonzero) elements, and let $\{k_1, k_2, \dots, k_m\}$ 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 $\mathbf{x}$ and $\mathbf{indx}$ such that $\mathbf{indx}(i) = k_i$ and $\mathbf{x}(i) = x_{k_i}$ then these subprograms compute

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

Usage VECLIB:

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

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

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

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

VECLIB8:

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

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

Input	m	Number of interesting elements of x , $m \leq n$, where n is the length of y . If $m \leq 0$, the subprograms do not reference x , indx , or y .
	a	The scalar a .
	x	Array of length m containing the interesting elements of x . $x(j) = x_i$ if $\text{indx}(j) = i$.
	indx	Array containing the indices $\{k_i\}$ of the interesting elements of x . The indices must satisfy $1 \leq \text{indx}(i) \leq n \quad i = 1, 2, \dots, m$ and $\text{indx}(i) \neq \text{indx}(j) \quad 1 \leq i \neq j \leq m,$ where n is the length of y .
Output	y	Array containing the elements of y , $y(i) = y_i$.
	y	If $m \leq 0$ or $a = 0$, then y is unchanged. Otherwise, $ax+y$ overwrites the input. Only the elements of y whose indices are included in indx are changed.
Notes	The result is unspecified if any element of indx is out of range, if any two elements of indx have the same value, or if x , indx , and y overlap such that any element of x or any index shares a memory location with any element of y .	

**Fortran
Equivalent**

```

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

```

Example Compute the REAL*8 elementary vector operation

$$y \leftarrow 2x + y,$$

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

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

Name SCLIP/DCLIP/ICLIP
Two-Sided Vector Clip

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

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

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 VECLIB:

```

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

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

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

```

VECLIB8:

```

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

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

```

Input

- n** Number of elements of vectors x and y to be used. If $n \leq 0$, the subprograms do not reference x or y .
- a** The scalar a .
- b** The scalar b .
- x** Array of length $\text{lenx} = (n-1) \times |\text{incx}| + 1$ 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)$\times$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 this chapter.
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 this chapter.
Output	y Array of length leny = (n-1)$\times$ incy +1 containing the n-vector y. If n $\leq$ 0, then y is unchanged. Otherwise, y is set as specified in "Purpose."

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

Fortran Equivalent

```

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

```

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

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

Name SCLIPL/DCLIPL/ICLIPL
Left-Sided Vector Clip

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

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

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 VECLIB:

```

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

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

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

```

VECLIB8:

```

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

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

```

Input

n	Number of elements of vectors x and y to be used. If $n \leq 0$, the subprograms do not reference x or y .
a	The scalar a .
x	Array of length $\text{lenx} = (n-1) \times \text{incx} + 1$ containing the n -vector x .
incx	Increment for the array x : $\text{incx} \geq 0$ x is stored forward in array x; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$.

incx < 0 x is stored backward in array x ; i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$.

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.

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 $y((i-1) \times \text{incy} + 1)$.

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

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** = $(n-1) \times |\text{incy}| + 1$ containing the n -vector y . If $n \leq 0$, then y is unchanged. Otherwise, y is set as specified in "Purpose."

Notes

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

**Fortran
Equivalent**

```

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

```

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

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

Name SCLIPR/DCLIPR/ICLIPR
Right-Sided Vector Clip

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

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

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 VECLIB:

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

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

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

VECLIB8:

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

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

Input

n	Number of elements of vectors x and y to be used. If $n \leq 0$, the subprograms do not reference x or y .
b	The scalar b .
x	Array of length $\text{lenx} = (n-1) \times \text{incx} + 1$ 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)$\times$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 this chapter.
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 this chapter.
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 set as specified in "Purpose."

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

Fortran Equivalent

```

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

```

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

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

Name SCOPY/DCOPY/ICOPY/CCOPY/CCOPYC/.../ZCOPYC
Copy Vector

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

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

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 VECLIB:

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

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

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

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

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

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

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

VECLIB8:

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

```

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

```

```

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

```

```

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

```

Input

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

x Array of length $\text{lenx} = (n-1) \times |\text{incx}| + 1$ containing the n -vector x . x is used in conjugated form by CCOPYC and ZCOPYC and in unconjugated form by the other subprograms. Refer to "Purpose."

incx Increment for the array x :

$\text{incx} \geq 0$ x is stored forward in array x ; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$.

$\text{incx} < 0$ x is stored backward in array x ; i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$.

Use $\text{incx} = 1$ 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 $\text{incx} = 0$. Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

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

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

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

Use $\text{incy} = 1$ 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 $\text{leny} = (n-1) \times |\text{incy}| + 1$ containing the n -vector y . If $n \leq 0$, then y is unchanged. Otherwise, $y \leftarrow x$.

Notes If $\text{incx} = 0$, 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)
  REAL*4 X(*), Y(*)
  IF ( N .LE. 0 ) RETURN
  IX = 1
  IY = 1
  IF ( INCX .LT. 0 ) IX = 1 - (N-1) * INCX
  IF ( INCY .LT. 0 ) IY = 1 - (N-1) * INCY
  DO 10 I = 1, N
    Y(IY) = X(IX)
    IX = IX + INCX
    IY = IY + INCY
  10 CONTINUE
  RETURN
END

```

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

```

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

```

Example 2 Initialize a one-dimensional array to zero.

```

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

```


Name SDOT/DDOT/CDOTC/CDOTU/ZDOTC/ZDOTU
Dot Product

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

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

where $\bar{x}$ is the complex conjugate of x . The vectors 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 VECLIB:

```

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

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

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

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

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

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

```

VECLIB8:

```

      INTEGER*8      n, incx, incy
      REAL*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 $n \leq 0$, the subprograms do not reference x or y .

x Array of length $\text{lenx} = (n-1) \times |\text{incx}| + 1$ containing the n -vector x . x is used in conjugated form by CDOTC and ZDOTC and in unconjugated form by the other subprograms. 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 $x((i-1) \times \text{incx} + 1)$.

incx < 0 x is stored backward in array x ; i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$.

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 $\text{leny} = (n-1) \times |\text{incy}| + 1$ 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 $y((i-1) \times \text{incy} + 1)$.

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

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 $n \leq 0$, then $s = 0$. Otherwise,

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

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

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

Notes

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

**Fortran
Equivalent**

```

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

```

Example 1 Compute the REAL*8 dot product

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

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

```

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

```

Example 2 Compute the REAL*8 dot product

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

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

```

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

```

Name SDOTI/DDOTI/CDOTCI/CDOTUI/.../ZDOTUI
Sparse Dot Product

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

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

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

More precisely, let x be a sparse n -vector with $m \leq n$ *interesting* (usually nonzero) elements, let $\{k_1, k_2, \dots, k_m\}$ 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 $\mathbf{x}$ and $\mathbf{indx}$ such that $\mathbf{indx}(i) = k_i$ and $\mathbf{x}(i) = x_{k_i}$, then these subprograms compute

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

Usage VECLIB:

```

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

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

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

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

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

```

```

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

```

VECLIB8:

```

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

```

```

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

```

```

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

```

Input

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

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

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

y Array containing the elements of y , $y(i) = y_i$.

Output

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

$$s = \sum_{i=1}^m x(i) \times y(indx(i)) \text{ unless the subprogram name is CDOTCI or ZDOTCI, in which case}$$

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

**Fortran
Equivalent**

```

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

```

Example Compute the REAL*8 sparse dot product

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

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

```

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

```

Name	SFRAC/DFRAC Extract Fractional Parts	
Purpose	Given a real vector x of length n, these subprograms extract the fractional portions of the elements of x and return them in a vector y. The vectors 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	VECLIB: <pre> INTEGER*4 n, incx, incy REAL*4 x(lenx), y(leny) CALL SFRAC(n, x, incx, y, incy) INTEGER*4 n, incx, incy REAL*8 x(lenx), y(leny) CALL DFRAC(n, x, incx, y, incy) </pre> VECLIB8: <pre> INTEGER*8 n, incx, incy REAL*8 x(lenx), y(leny) CALL SFRAC(n, x, incx, y, incy) </pre>	
Input	n	Number of elements of vectors x and y to be used. If $n \leq 0$, the subprograms do not reference x or y .
	x	Array of length $\text{lenx} = (n-1) \times \text{incx} + 1$ containing the n -vector x .
	incx	Increment for the array x: $\text{incx} \geq 0$ x is stored forward in array x; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$. $\text{incx} < 0$ x is stored backward in array x; i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$. Use $\text{incx} = 1$ 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.
	incy	Increment for the array y, $\text{incy} \neq 0$: $\text{incy} > 0$ y is stored forward in array y; i.e., y_i is stored in $y((i-1) \times \text{incy} + 1)$.

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

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** = $(n-1) \times |\text{incy}| + 1$ containing the n -vector y . If $n \leq 0$, y is unchanged. Otherwise, y_i is the fractional part of x_i .

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

Fortran Equivalent

```

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

```

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

```

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

```

Name SGTHR/DGTHR/IGTHR/CGTHR/ZGTHR
Gather Sparse Vector

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

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

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

Usage VECLIB:

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

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

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

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

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

VECLIB8:

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

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

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

Input

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

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

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

$$1 \leq indx(i) \leq n, \quad i = 1, 2, \dots, m,$$

where n is the length of y .

Output

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

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

Fortran Equivalent

```

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

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

```

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

Name SGTHRZ/DGTHRZ/.../ZGTHRZ
Gather and Zero Sparse Vector

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

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

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

If all nonzero elements of y are listed in $\mathbf{indx}$, then these subprograms simultaneously perform the vector operations

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

Usage VECLIB:

```

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

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

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

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

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

```

VECLIB8:

```

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

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

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

```

Input	m	Number of interesting elements of y , $m \leq n$, where n is the length of y . If $m \leq 0$, the subprograms do not reference x , $indx$, or y .
	y	Array containing the elements of y , $y(i) = y_i$. Only the elements of y whose indices are included in $indx$ are accessed.
	indx	Array containing the indices $\{k_i\}$ of the interesting elements of y . The indices must satisfy $1 \leq indx(i) \leq n, i = 1, 2, \dots, m$ and $indx(i) \neq indx(j), 1 \leq i \neq j \leq m,$ where n is the length of y .
Output	x	If $m \leq 0$, then x is unchanged. Otherwise, the m interesting elements of y : $x(j) = y_i$ if $indx(j) = i$.
	y	$y(indx(i)) = 0, \quad i = 1, 2, \dots, m.$
Notes	The result is unspecified if any element of $indx$ is out of range, if any two elements of $indx$ have the same value, or if x , $indx$, and y overlap such that any element of y or any index shares a memory location with any element of x .	

**Fortran
Equivalent**

```

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

```

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

```

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

```

Name SLSTxx/DLSTxx/ILSTxx/.../ZLSTxx
List Selected Vector Elements

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

The last two characters of the subprogram name specify the relationship of interest between the elements of the vector and the scalar. For real and integer subprograms, these characters, represented by "xx" in the prototype Fortran statements, and the corresponding list contents 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	$\{i : x_i \neq a\}$

For complex subprograms, these characters and corresponding list contents are

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

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

Usage VECLIB:

```

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

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

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

```

```

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

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

```

VECLIB8:

```

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

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

INTEGER*8      n, incx, nindx, indx(n)
COMPLEX*16     a, x(lenx)
CALL CLSTxx(n, x, incx, a, nindx, indx)

```

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 $lenx = (n-1) \times incx + 1$ 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 $x((i-1) \times incx + 1)$. 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 this chapter.
	a	The scalar a .
Output	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.
	indx	Array filled with the list of indices i of the elements x_i of x that satisfy the relationship with a specified by the subprogram name. Only the first $nindx$ elements of $indx$ are changed. Recall that x_i is stored in $x((i-1) \times incx + 1)$.

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

**Fortran
Equivalent**

```

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

```

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

```

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

```

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

```

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

```

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

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

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

Name SMAX/DMAX/IMAX
Maximum of Vector

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

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

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

Usage VECLIB:

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

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

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

VECLIB8:

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

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

Input

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

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

incx Increment for the array x . x is stored forward in array x with increment $|\text{incx}|$; i.e., x_i is stored in $x((i-1) \times |\text{incx}| + 1)$.
Use $\text{incx} = 1$ 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 $n \leq 0$, then $s = -\infty$, the most negative representable machine number. Otherwise, s is the maximum of the elements of x .

**Fortran
Equivalent**

```

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

```

Example

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

```

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

```

Name SMIN/DMIN/IMIN
Minimum of Vector

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

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

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

Usage VECLIB:

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

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

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

VECLIB8:

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

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

Input

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

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

incx Increment for the array x . x is stored forward in array x with increment $|\text{incx}|$; i.e., x_i is stored in $x((i-1) \times |\text{incx}| + 1)$.
Use $\text{incx} = 1$ 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 $n \leq 0$, then $s = \infty$, the largest representable machine number. Otherwise, **s** is the minimum of the elements of x .

**Fortran
Equivalent**

```

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

```

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

```

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

```

Name SNRM2/DNRM2/SCNRM2/DZNRM2
Euclidean Norm

Purpose Given a real or complex vector x of length n , these subprograms compute the Euclidean (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 VECLIB:

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

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

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

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

VECLIB8:

```
INTEGER*8      n, incx
REAL*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 $n \leq 0$, the subprograms do not reference x .

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

incx Increment for the array x . x is stored forward in array x with increment $|\text{incx}|$; i.e., x_i is stored in $x((i-1) \times |\text{incx}| + 1)$.
Use $\text{incx} = 1$ 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 $n \leq 0$, then $s = 0$. Otherwise, s is the Euclidean norm of x .

Fortran Equivalent

```

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

```

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

```

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

```


Name SNRSQ/DNRSQ/SCNRSQ/DZNRSQ
Euclidean Norm Squared

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

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

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

Usage VECLIB:

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

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

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

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

VECLIB8:

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

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

Input n Number of elements of vector x to be used in the calculation. If $n \leq 0$, the subprograms do not reference x .

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

incx Increment for the array x . x is stored forward in array x with increment $|\text{incx}|$; i.e., x_i is stored in $x((i-1) \times |\text{incx}| + 1)$.
Use $\text{incx} = 1$ 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 $n \leq 0$, then $s = 0$. Otherwise, s is the square of the Euclidean norm of x .

Fortran Equivalent

```

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

```

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

```

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

```

Name SRAMP/DRAMP/IRAMP
Generate Linear Ramp

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

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

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

Usage VECLIB:

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

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

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

VECLIB8:

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

```
INTEGER*8      n, incx, a, h, x(lenx)
CALL IRAMP(n, a, h, x, incx)
```

Input

n	Number of elements of x to be generated.
a	The scalar a .
h	The scalar h .
incx	Increment for the array x , $\text{incx} \neq 0$. x is stored forward in array x with increment $ \text{incx} $; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$. Use $\text{incx} = 1$ 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** Array of length $\text{lenx} = (\mathbf{n}-1) \times |\mathbf{incx}| + 1$ containing the n -vector x . If $\mathbf{n} \leq 0$, then $\mathbf{x}$ is not referenced. Otherwise, the specified linear ramp function replaces the input.

Notes The result is unspecified if $\mathbf{incx} = 0$.

**Fortran
Equivalent**

```

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

```

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

```

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

```

Name SROT/DROT/CROT/CSROT/ZROT/ZDROT
Apply Givens Rotation

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

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

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. The indexing through the arrays may be either forward or backward.

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

Usage VECLIB:

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

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

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

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

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

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

VECLIB8:

```

INTEGER*8      n, incx, incy
REAL*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)

```

```

INTEGER*8      n, incx, incy
REAL*8         c, s
COMPLEX*16     x(lenx), y(leny)
CALL CSROT(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 $n \leq 0$, the subprograms do not reference x or y .

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

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

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

$\text{incx} < 0$ x is stored backward in array x ; i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$.

Use $\text{incx} = 1$ 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 $\text{leny} = (n-1) \times |\text{incy}| + 1$ containing the n -vector y .

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

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

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

Use $\text{incy} = 1$ 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 <i>c</i> .
	s	The scalar <i>s</i> .
Output	x and y	If $n \leq 0$ or if $c = 1$ and $s = 0$, then x and y are unchanged. Otherwise, the result vectors overwrite the input.
Notes	The result is unspecified if incx = 0 or incy = 0 or if x and y overlap such that any element of x shares a memory location with any element of y. There are no companion subprograms that construct real Givens rotations for CSROT and ZDROT. VECLIB also contains subprograms that construct and apply modified Givens rotations. They are documented elsewhere in this chapter. The modified Givens subprograms are a little more difficult to use but are more efficient.	

Fortran Equivalent

```

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

```

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

```

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

```

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

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


Name SROTG/DROTG/CROTG/ZROTG
Construct Givens Rotation

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

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

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

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

Usage VECLIB:

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

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

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

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

VECLIB8:

```
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 and ZROTG. In SROTG and DROTG, the reconstruction quantity z overwrites b . The reconstruction quantity z is useful if a matrix is being transformed by a sequence of Givens rotations that must be saved to be applied again. 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 VECLIB also contains subprograms that construct and apply modified Givens rotations. They are documented elsewhere in this chapter. The modified Givens subprograms are a little more difficult to use but are more efficient.

Example Construct a Givens plane rotation that 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 DROTG (X(1),Y(1),C,S)
```

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

Name SROT/DROT
Apply Sparse Givens Rotation

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

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

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

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

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

Usage VECLIB:

```

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

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

```

VECLIB8:

```

      INTEGER*8      m, indx(m)
      REAL*8         x(m), y(n), c, s
      CALL SROT(m, x, indx, y, c, s)

```

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

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

indx Array containing the indices $\{k_i\}$ of the interesting elements of x . The indices must satisfy
 $1 \leq \text{indx}(i) \leq n$, $i = 1, 2, \dots, m$
 and $\text{indx}(i) \neq \text{indx}(j)$, $1 \leq i \neq j \leq m$, where n is the length of y .

y Array containing the elements of y , $y(i) = y_i$.

c The scalar c .

s The scalar s .

Output **x and y** If $m \leq 0$ or if $c = 1$ and $s = 0$, then x and y are unchanged. Otherwise, the result vectors overwrite the input. Only the elements of y whose indices are included in indx are changed.

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

Fortran Equivalent

```

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

```

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

```

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

```

Name SROTMDROTMD
Apply Modified Givens Rotation

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

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

Refer to the description of the companion subprograms SROTMD and DROTMD 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 VECLIB:

```

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

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

```

VECLIB8:

```

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

```

Input

n	Number of elements of vectors x and y to be used. If $n \leq 0$, the subprograms do not reference x or y .
x	Array of length $\text{lenx} = (n-1) \times \text{incx} + 1$ containing the n -vector x .
incx	Increment for the array x , $\text{incx} \neq 0$: $\text{incx} > 0$ x is stored forward in array x; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$. $\text{incx} < 0$ x is stored backward in array x; i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$.

Use $\text{incx} = 1$ 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 $\text{leny} = (n-1) \times |\text{incy}| + 1$ containing the n -vector y .

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

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

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

Use $\text{incy} = 1$ 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 or DROTMG; refer to the description of these companion subprograms for the specific contents of **param**.

Output **x and y** If $n \leq 0$ or if $\text{param}(1) = -2$, **x** and **y** are unchanged. Otherwise, the result vectors overwrite the input.

Notes The result is unspecified if $\text{incx} = 0$ or $\text{incy} = 0$ or if **x** and **y** overlap such that any element of x shares a memory location with any element of y .
VECLIB also contains subprograms that construct and apply regular Givens rotations. They are documented elsewhere in this chapter. The modified Givens subprograms are a little more difficult to use but are more efficient.

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

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

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

```

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

```

Name SROTMG/DROTMG
Construct Modified Givens Rotation

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

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

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

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

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

$$GW = \begin{bmatrix} \bar{d}_1^{1/2} & 0 \\ 0 & \bar{d}_2^{1/2} \end{bmatrix} \cdot \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \cdot \begin{bmatrix} x_1 & \dots & x_n \\ y_1 & \dots & y_n \end{bmatrix}$$

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 VECLIB subprograms SROTM and DROTM are provided to apply the modified Givens notation to a pair of vectors.

In most applications, d_1 and d_2 are initially set to 1, manipulated by SROTMG or DROTMG as the modified Givens rotations are constructed, then applied to the vectors as the final step in the computation. For example, the reduction of an n -by- n matrix to upper-triangular form via modified Givens rotations requires $O(n)$ square roots compared to the $O(n^2)$ required by ordinary Givens rotations. Refer to "Example 2" in the description of SROTM and DROTM.

Usage VECLIB:
REAL*4 d1, d2, x1, y1, param(5)
CALL SROTMG(d1, d2, x1, y1, param)


```

REAL*8      d1, d2, x1, y1, param(5)
CALL DROTMG(d1, d2, x1, y1, param)

```

VECLIB8:

```

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

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

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

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

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

For each of the four values of **param**(1), only the indicated values of **param**(2) through **param**(5) are defined. The 0, 1, and -1 elements are not stored in **param**.

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

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

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

Name SRSCL/DRSCL/CRSCL/CSRSCL/ZRSCL/ZDRSCL
Scale Vector

Purpose Given a real or complex scalar a and a real or complex vector x of length n , these subprograms perform the reciprocal vector scaling operation

$$x \leftarrow x \div a$$

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

Usage VECLIB:

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

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

```
INTEGER*4      n, incx
COMPLEX*8      a, x(lenx)
CALL CRSCL(n, a, x, incx)
```

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

```
INTEGER*4      n, incx
COMPLEX*16     a, x(lenx)
CALL ZRSCL(n, a, x, incx)
```

```
INTEGER*4      n, incx
REAL*8         a
COMPLEX*16     x(lenx)
CALL ZDRSCL(n, a, x, incx)
```

VECLIB8:

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

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

```

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

```

Input	n	Number of elements of vector x to be used in the scaling operation. If $n \leq 0$, the subprograms do not reference x .
	a	The scalar a , $a \neq 0$.
	x	Array of length $\text{lenx} = (n-1) \times \text{incx} + 1$ containing the n -vector x .
	incx	Increment for the array x , $\text{incx} \neq 0$. x is stored forward in array x with increment $ \text{incx} $; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$. Use $\text{incx} = 1$ 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 $n \leq 0$, then x is unchanged. Otherwise, $x+a$ replaces the input.
Notes		The result is unspecified if $\text{incx} = 0$. A divide-by-zero error will occur if $a = 0$ and $n > 0$.

Fortran Equivalent

```

SUBROUTINE SRACL (N,A, X, INCX)
REAL*4 A,X(*)
IF ( N .LE. 0 ) RETURN
IX = 1
INCXA = ABS ( INCX )
DO 10 I = 1, N
    X(IX) = X(IX) / A
    IX = IX + INCXA
10 CONTINUE
RETURN
END

```

Example Scale the REAL*8 vector x by dividing by 2, where x is a vector 10 elements long stored in a one-dimensional array x of dimension 20.

```

INTEGER*4 N, INCX
REAL*8    A,X(20)
N = 10
INCX = 1
A = 2.0D0
CALL DRACL (N,A,X, INCX)

```

Name SSCAL/DSCAL/CSCAL/CSSCAL/CSCALC/.../ZSCALC
Scale Vector

Purpose Given a real or complex scalar a and a real or complex vector x of length n , these subprograms perform the vector scaling operations

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

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 VECLIB:

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

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

```
INTEGER*4      n, incx
COMPLEX*8      a, x(lenx)
CALL CSCAL(n, a, x, incx)
```

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

```
INTEGER*4      n, incx
COMPLEX*8      a, x(lenx)
CALL CSCALC(n, a, x, incx)
```

```
INTEGER*4      n, incx
COMPLEX*16     a, x(lenx)
CALL ZSCAL(n, a, x, incx)
```

```
INTEGER*4      n, incx
REAL*8         a
COMPLEX*16     x(lenx)
CALL ZDSCAL(n, a, x, incx)
```

```
INTEGER*4      n, incx
COMPLEX*16     a, x(lenx)
CALL ZSCALC(n, a, x, incx)
```

VECLIB8:

```

      INTEGER*8      n, incx
      REAL*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)

```

```

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

```

Input	n	Number of elements of vector x to be used in the scaling operation. If $n \leq 0$, the subprograms do not reference x .
	a	The scalar a .
	x	Array of length $\text{lenx} = (n-1) \times \text{incx} + 1$ containing the n -vector x . x is used in conjugated form by CSCALC and ZSCALC and in unconjugated form by the other subprograms. Refer to "Purpose."
	incx	Increment for the array x , $\text{incx} \neq 0$. x is stored forward in array x with increment $ \text{incx} $; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$. Use $\text{incx} = 1$ 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 $n \leq 0$, then x is unchanged. Otherwise, ax replaces the input.
Notes	The result is unspecified if $\text{incx} = 0$.	

**Fortran
Equivalent**

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

Example Scale the REAL*8 vector x by 2, where x is a vector 10 elements long stored in a one-dimensional array x of dimension 20.

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

Name SSCTR/DSCTR/ISCTR/CSCTR/ZSCTR
Scatter Sparse Vector

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

More precisely, let x be a sparse n -vector with $m \leq n$ *interesting* (usually nonzero) elements, and let $\{k_1, k_2, \dots, k_m\}$ be the indices of these elements. If x is represented by arrays $\mathbf{x}$ and $\mathbf{indx}$ such that $\mathbf{indx}(i) = k_i$ and $\mathbf{x}(i) = x_{k_i}$, then

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

Usage VECLIB:

```

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

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

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

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

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

```

VECLIB8:

```

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

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

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

```


Input	m	Number of interesting elements, $m \leq n$, where n is the length of y . If $m \leq 0$, the subprograms do not reference x , indx , or y .
	x	Array of length m containing the interesting elements of x . $x(j) = x_i$ if $\text{indx}(j) = i$.
	indx	Array containing the indices $\{k_i\}$ of the interesting elements of x . The indices must satisfy $1 \leq \text{indx}(i) \leq n$, $i = 1, 2, \dots, m$ and $\text{indx}(i) \neq \text{indx}(j)$, $1 \leq i \neq j \leq m$, where n is the length of y .
Output	y	Array containing the elements of y , $y(i) = y_i$. If $m \leq 0$, then y is unchanged. Otherwise, only the elements of y whose indices are included in indx are changed.
Notes	The result is unspecified if any element of indx is out of range; if any two elements of indx have the same value, or if x , indx , and y overlap such that any element of x or any index shares a memory location with any element of y .	

**Fortran
Equivalent**

```

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

```

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

```

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

```

Name SSUM/DSUM/ISUM/CSUM/ZSUM
Vector Sum

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

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

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

Usage VECLIB:

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

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

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

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

```
INTEGER*4      n, incx
COMPLEX*16     s, ZSUM, x(lenx)
s = ZSUM(n, x, incx)
```

VECLIB8:

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

```
INTEGER*8      n, incx, s, ISUM, x(lenx)
s = ISUM(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 $n \leq 0$, the subprograms do not reference x .
	x	Array of length $\text{lenx} = (n-1) \times \text{incx} + 1$ containing the n -vector x .
	incx	Increment for the array x . x is stored forward in array x with increment $ \text{incx} $; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$. Use $\text{incx} = 1$ 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 $n \leq 0$, then $s = 0$. Otherwise, s is the sum of the elements of x .

**Fortran
Equivalent**

```

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

```

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

```

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

```

Name SSWAP/DSWAP/ISWAP/CSWAP/ZSWAP
Swap Two Vectors

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

$$x \leftrightarrow y.$$

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

Usage VECLIB:

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

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

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

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

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

VECLIB8:

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

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

```
INTEGER*8      n, incx, incy
COMPLEX*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 $n \leq 0$, the subprograms do not reference x or y .
	x	Array of length $\text{lenx} = (n-1) \times \text{incx} + 1$ containing the n -vector x .
	incx	Increment for the array x , $\text{incx} \neq 0$: $\text{incx} > 0$ x is stored forward in array x ; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$. $\text{incx} < 0$ x is stored backward in array x ; i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$. Use $\text{incx} = 1$ 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 $\text{leny} = (n-1) \times \text{incy} + 1$ containing the n -vector y .
	incy	Increment for the array y , $\text{incy} \neq 0$: $\text{incy} > 0$ y is stored forward in array y ; i.e., y_i is stored in $y((i-1) \times \text{incy} + 1)$. $\text{incy} < 0$ y is stored backward in array y ; i.e., y_i is stored in $y((i-n) \times \text{incy} + 1)$. Use $\text{incy} = 1$ 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 $n \leq 0$, then x and y are unchanged. Otherwise, x and y are interchanged in x and y .
Notes		The result is unspecified if $\text{incx} = 0$ or $\text{incy} = 0$ or if x and y overlap such that any element of x shares a memory location with any element of y .

**Fortran
Equivalent**

```

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

```

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

```

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

```

Example 2 Interchange rows 3 and 6 of a 10-by-10 matrix a stored in two-dimensional array A of dimension 20 by 21.

```

INTEGER*4 N, INCA
REAL*8    A(20,21)
N = 10
INCA = 20
CALL DSWAP (N, A(3,1), INCA, A(6,1), INCA)

```

Name SWDOT/DWDOT/CWDOTC/.../ZWDOTU
Weighted Dot Product

Purpose Given a real weight vector w and real or complex data vectors x and y , all of length n , these subprograms compute the weighted dot products

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

where $\bar{x}$ is the complex conjugate of x . The vectors 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 VECLIB:

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

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

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

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

```
INTEGER*4      n, incw, incx, incy
REAL*8         w(lenw)
COMPLEX*16     s, ZWDOTC, x(lenx), y(leny)
s = ZWDOTC(n, w, incw, x, incx, y, incy)
```

```
INTEGER*4      n, incw, incx, incy
REAL*8         w(lenw)
COMPLEX*16     s, ZWDOTU, x(lenx), y(leny)
s = ZWDOTU(n, w, incw, x, incx, y, incy)
```

VECLIB8:

```

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

      INTEGER*8      n, incw, incx, incy
      REAL*8         w(lenw)
      COMPLEX*16     s, CWDOTC, x(lenx), y(leny)
      s = CWDOTC(n, w, incw, x, incx, y, incy)

      INTEGER*8      n, incw, incx, incy
      REAL*8         w(lenw)
      COMPLEX*16     s, CWDOTU, x(lenx), y(leny)
      s = CWDOTU(n, w, incw, x, incx, y, incy)

```

Input

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

w Array of length $\text{lenw} = (n-1) \times |\text{incw}| + 1$ containing the n -vector w .

incw Increment for the array w :

$\text{incw} \geq 0$ w is stored forward in array w ; i.e., w_i is stored in $w((i-1) \times \text{incw} + 1)$.

$\text{incw} < 0$ w is stored backward in array w ; i.e., w_i is stored in $w((i-n) \times \text{incw} + 1)$.

Use $\text{incw} = 1$ if the vector w is stored contiguously in array w ; i.e., if w_i is stored in $w(i)$. Refer to "BLAS Indexing Conventions" in the introduction to this chapter.

x Array of length $\text{lenx} = (n-1) \times |\text{incx}| + 1$ containing the n -vector x . x is used in conjugated form by CWDOTC and ZWDOTC and in unconjugated form by the other subprograms. Refer to "Purpose."

incx Increment for the array x :

$\text{incx} \geq 0$ x is stored forward in array x ; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$.

$\text{incx} < 0$ x is stored backward in array x ; i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$.

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

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

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

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

Output

s The resulting value of the weighted dot product. If $n \leq 0$, then $s = 0$. Otherwise, $s = \sum_{i=1}^n w_i x_i y_i$ unless the subprogram name is CWDOTC or ZWDOTC, in which case $s = \sum_{i=1}^n w_i \bar{x}_i y_i$.

Notes

If **incw** = 0, then $w_i = w(1)$ for all i . If **incx** = 0, then $x_i = x(1)$ for all i . If **incy** = 0, then $y_i = y(1)$ for all i . In any of these cases, another VECLIB dot product subprogram would be more efficient.

**Fortran
Equivalent**

```

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

```

Example 1 Compute the REAL*8 weighted dot product

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

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

```

INTEGER*4 N, INCW, INCX, INCY
REAL*8 S, DWDOT, W(20), X(20), Y(20)
N = 10
INCW = 1
INCX = 1
INCY = 1
S = DWDOT (N, W, INCW, X, INCX, Y, INCY)

```

Example 2 Compute the REAL*8 weighted dot product

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

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

```

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

```

Name	SZERO/DZERO/IZERO/CZERO/ZZERO Clear Vector	
Purpose	These subprograms set all of the elements of a real, integer, or complex n -vector x to zero. The vector may be stored in a one-dimensional array or in either a row or a column of a two-dimensional array.	
Usage	VECLIB: <pre> INTEGER*4 n, incx REAL*4 x(lenx) CALL SZERO(n, x, incx) INTEGER*4 n, incx REAL*8 x(lenx) CALL DZERO(n, x, incx) INTEGER*4 n, incx, x(lenx) CALL IZERO(n, x, incx) INTEGER*4 n, incx COMPLEX*8 x(lenx) CALL CZERO(n, x, incx) INTEGER*4 n, incx COMPLEX*16 x(lenx) CALL ZZERO(n, x, incx) </pre> VECLIB8: <pre> INTEGER*8 n, incx REAL*8 x(lenx) CALL SZERO(n, x, incx) INTEGER*8 n, incx, x(lenx) CALL IZERO(n, x, incx) INTEGER*8 n, incx COMPLEX*16 x(lenx) CALL CZERO(n, x, incx) </pre>	
Input	n	Number of elements of vector x to be set to zero. If $n \leq 0$, the subprograms do not reference x .
	incx	Increment for the array x , $\text{incx} \neq 0$. x is stored forward in array x with increment $ \text{incx} $; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$.

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

Output**x**

Array of length $\text{lenx} = (\mathbf{n}-1) \times |\text{incx}| + 1$ containing the n -vector x which has been set to zero. If $\mathbf{n} \leq 0$, then $\mathbf{x}$ is unchanged. Otherwise, $x \leftarrow 0$.

**Fortran
Equivalent**

```

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

```

Example

Zero the REAL*8 vector x , where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20 (compare with "Example 2" in the description of SCOPY).

```

INTEGER*4 N, INCX
REAL*8 X(20)
N = 10
INCX = 1
CALL DZERO (N, X, INCX)

```

3 Basic Matrix Operations

Overview

This chapter describes the subprograms in the Level 2 (two-loop) Basic Linear Algebra Subprograms (BLAS) and the Level 3 (three-loop) BLAS. Collectively, these two sets of subprograms are called the Extended BLAS.

The most important of these subprograms have been coded in highly-tuned assembly language.

This chapter explains how to use the VECLIB 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.

The following documents provide supplemental material for this chapter:

Dongarra, J.J., J. DuCroz, S. Hammarling, and R. Hanson. "An Extended Set of Fortran Basic Linear Algebra Subprograms." *ACM Transactions on Mathematical Software*. March, 1988. Vol. 14, No. 1.

Dongarra, J.J., J. DuCroz, S. Hammarling, and I. Duff. "A Set of Level 3 Basic Linear Algebra Subprograms." *ACM Transactions on Mathematical Software*. March, 1990. Vol. 16, No. 1.

Higham, Nicholas J. "Is Fast Matrix Multiplication of Practical Use?" *SIAM News*. November, 1990. Vol. 23, No. 6.

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 subroutine 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
D	Double Precision REAL
C	Single Precision COMPLEX
Z	Double Precision COMPLEX

The next two letters in the naming convention indicate the form of the matrix, as presented in Table 3-2:

Table 3-2 Extended BLAS Naming Convention — Matrix Form

XX	Form of Matrix
GE	General
GB	General band
HE	Hermitian
HB	Hermitian band
HP	Hermitian packed
SY	Symmetric
SB	Symmetric band
SP	Symmetric packed
TR	Triangular
TB	Triangular band
TP	Triangular packed

Table 3-3 lists the final one, two, or three characters in the naming convention, indicating the computation of a particular subroutine:

Table 3-3 Extended BLAS Naming Convention — Computation

YY	Subroutine Computation
MM	Matrix-Matrix multiply
MV	Matrix-Vector multiply
R	Rank-1 update
R2	Rank-2 update
RK	Rank-k update
R2K	Rank-2k update
SM	Solve multiple systems of linear equations
SV	Solve a system of linear equations

For example, SGBMV multiplies a vector (MV) by a general band matrix (GB) using the single precision REAL data type (S). ZTRSM solves a system of linear equations with one triangular coefficient matrix and a matrix of right-hand sides, using the double precision COMPLEX data type. Table 3-4 shows the valid combinations of T, XX, and Y, YY, or YYY. Each line indicates the allowable T prefixes and Y, YY, or YYY suffixes for a particular root name XX.

Table 3-4 Extended BLAS Naming Convention — Subprogram Names

Valid T				XX	Valid Y, YY, or YYY									
S	D			GE	MM	MV	R							
		C	Z	GE	MM	MV						RC	RU	
S	D	C	Z	GB		MV								
		C	Z	HE	MM	MV	R	R2	RK	R2K				
		C	Z	HB		MV								
		C	Z	HP		MV	R	R2						
S	D			SY	MM	MV	R	R2	RK	R2K				
		C	Z	SY	MM				RK	R2K				
S	D			SB		MV								
S	D			SP		MV	R	R2						
S	D	C	Z	TR	MM	MV					SM	SV		
S	D	C	Z	TB		MV						SV		
S	D	C	Z	TP		MV						SV		

The subprograms SGEMMS, DGEMMS, CGEMMS, and ZGEMMS, although not part of the standard Extended BLAS, are consistent with this nomenclature.

Subprograms Included in This Chapter

Following are the Extended BLAS subprograms included with VECLIB.

Name SGBMV/DGBMV/CGBMV/ZGBMV
Matrix-Vector Multiply

Purpose These subprograms compute the matrix-vector products Ax , $A^T x$, 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

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

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**VECLIB:**

```

CHARACTER*1  trans
INTEGER*4    m, n, kl, ku, ldab, incx, incy
REAL*4       alpha, beta, ab(ldab, n), x(lenx), y(leny)
CALL SGBMV(trans, m, n, kl, ku, alpha, ab, ldab, x, incx, beta, y, incy)

```

```

CHARACTER*1  trans
INTEGER*4    m, n, kl, ku, ldab, incx, incy
REAL*8       alpha, beta, ab(ldab, n), x(lenx), y(leny)
CALL DGBMV(trans, m, n, kl, ku, alpha, ab, ldab, x, incx, beta, y, incy)

```

```

CHARACTER*1  trans
INTEGER*4    m, n, kl, ku, ldab, incx, incy
COMPLEX*8    alpha, beta, ab(ldab, n), x(lenx), y(leny)
CALL CGBMV(trans, m, n, kl, ku, alpha, ab, ldab, x, incx, beta, y, incy)

```

```

CHARACTER*1  trans
INTEGER*4    m, n, kl, ku, ldab, incx, incy
COMPLEX*16   alpha, beta, ab(ldab, n), x(lenx), y(leny)
CALL ZGBMV(trans, m, n, kl, ku, alpha, ab, ldab, x, incx, beta, y, incy)

```

VECLIB8:

```

CHARACTER*1  trans
INTEGER*8    m, n, kl, ku, ldab, incx, incy
REAL*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, $m \geq 0$. If $m = 0$, the
 subprograms do not reference **ab**, **x**, or **y**.
n Number of columns in matrix A, $n \geq 0$. If $n = 0$, 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, $0 \leq$
 $kl < n$.
ku The upper bandwidth of A, i.e., the number of nonzero
 diagonals above the principal diagonal in the band, $0 \leq$
 $ku < n$.
alpha The scalar α . If **alpha** = 0, the subprograms compute
 $y \leftarrow \beta y$ without referencing **ab** or **x**.
 Array containing the m -by- n band matrix A in the
 compressed form described above. If a_{ij} is in the band,
 it is stored in **ab**($ku+1+i-j, j$). The columns of A are
 stored in the columns of **ab**, and the diagonals of A are
 stored in rows 1 through $kl+ku+1$.
ldab The leading dimension of array **ab** as declared in the
 calling program unit, with $ldab \geq kl+ku+1$.
x Array containing the vector x . The number of elements
 of x and the value of **lenx**, the dimension of the array **x**,
 depend on **trans**:
 'N' or 'n' x has n elements **lenx** = $(n-1) \times |incx| + 1$
 otherwise x has m elements **lenx** = $(m-1) \times |incx| + 1$
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 $x((i-1) \times \text{incx} + 1)$.

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

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** = $(m-1) \times |\text{incy}| + 1$
 otherwise y has n elements **leny** = $(n-1) \times |\text{incy}| + 1$

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 $y((i-1) \times \text{incy} + 1)$.

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

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*4** matrix-vector product $y = Ax$, where A is a 9 by 6 real band matrix whose lower bandwidth is 2 and whose upper bandwidth is 3. A is stored in an array **AB** whose dimensions are 10 by 10, x is a real vector 6 elements long stored in an array **X** of dimension 10, and y is a real vector 9 elements long stored in an array **Y**, also of dimension 10.

```

CHARACTER*1 TRANS
INTEGER*4 M,N,KL,KU,LDAB,INCX,INCY
REAL*4 ALPHA,BETA,AB(10,10),X(10),Y(10)
TRANS = 'N'
M = 9
N = 6
KL = 2
KU = 3
ALPHA = 1.0
BETA = 0.0
LDAB = 10
INCX = 1
INCY = 1
CALL SGBMV (TRANS,M,N,KL,KU,ALPHA,AB,LDAB,X,INCX,BETA,Y,INCY)

```

Example 2 Form the **REAL*8** matrix-vector product $y = 1/2y - pA^T x$, where p is a real scalar, A is a 6-by-9 real band matrix whose lower bandwidth is 1 and whose upper bandwidth is 2. A is stored in an array **AB** whose dimensions are 10 by 10, x is a real vector 6 elements long stored in an array **X** of dimension 10, and y is a real vector 9 elements long stored in an array **Y**, also of dimension 10.

```

INTEGER*4 M,N,KL,KU,LDAB
REAL*8 RHO,AB(10,10),X(10),Y(10)
M = 9
N = 6
KL = 1
KU = 2
LDAB = 10
CALL DGBMV ('TRANSPOSE',M,N,KL,KU,-RHO,AB,LDAB,X,1,0.5D0,Y,1)

```

Name SGEMM/DGEMM/CGEMM/ZGEMM
Matrix-Matrix Multiply

Purpose These subprograms compute the matrix-matrix product AB , where A is an m -by- k matrix, and B is a k -by- n matrix. Optionally, A 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

VECLIB:

```

CHARACTER*1  transa, transb
INTEGER*4    m, n, k, lda, ldb, ldc
REAL*4       alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL SGEMM(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)

CHARACTER*1  transa, transb
INTEGER*4    m, n, k, lda, ldb, ldc
REAL*8       alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL DGEMM(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)

CHARACTER*1  transa, transb
INTEGER*4    m, n, k, lda, ldb, ldc
COMPLEX*8    alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL CGEMM(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)

CHARACTER*1  transa, transb
INTEGER*4    m, n, k, lda, ldb, ldc
COMPLEX*16   alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL ZGEMM(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)

```

VECLIB8:

```

CHARACTER*1  transa, transb
INTEGER*8    m, n, k, lda, ldb, ldc
REAL*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, $m \geq 0$. If $m = 0$, the
 subprograms do not reference a, b, or c.

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

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

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

a Array containing the matrix A, whose size is indicated
 by **transa**:
 'N' or 'n' A is an m -by- k matrix
 otherwise A is a k -by- m matrix

lda	The leading dimension of array a as declared in the calling program unit, with $\text{lda} \geq \max(\text{the number of rows of } A, 1)$.
b	Array containing the matrix <i>B</i> , whose size is indicated by transb : ' N ' or ' n ' <i>B</i> is a <i>k</i> -by- <i>n</i> matrix otherwise <i>B</i> is an <i>n</i> -by- <i>k</i> matrix
ldb	The leading dimension of array b as declared in the calling program unit, with $\text{ldb} \geq \max(\text{the number of rows of } B, 1)$.
beta	The scalar β .
c	Array containing the <i>m</i> -by- <i>n</i> matrix <i>C</i> . Not used as input if beta = 0.
ldc	The leading dimension of array c as declared in the calling program unit, with $\text{ldc} \geq \max(m, 1)$.

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

transa $\neq$ '**N**' or '**n**' or '**T**' or '**t**' or '**C**' or '**c**',
transb $\neq$ '**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*4 matrix product $C = AB$, where A is a 9-by-6 real matrix stored in an array A whose dimensions are 10 by 10, B is a 6-by-8 real matrix stored in an array B of dimension 10 by 10, and C is a 9-by-8 real matrix stored in an array C , also of dimension 10 by 10.

```

CHARACTER*1 TRANSA,TRANSB
INTEGER*4    M,N,K,LDA,LDB,LDC
REAL*4       ALPHA,BETA,A(10,10),B(10,10),C(10,10)
TRANSA = 'N'
TRANSB = 'N'
M = 9
N = 8
K = 6
ALPHA = 1.0
BETA = 0.0
LDA = 10
LDB = 10
LDC = 10
CALL SGEMM (TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)

```

Example 2 Form the REAL*8 matrix product $C = 1/2C + pA^TB$, where p is a real scalar, A is a 6-by-9 real matrix stored in an array A whose dimensions are 10 by 10, B is a 6-by-8 real matrix stored in an array B of dimension 10 by 10, and C is a 9-by-8 real matrix stored in an array C , also of dimension 10 by 10.

```

INTEGER*4 M,N,K,LDA,LDB,LDC
REAL*8    RHO,A(10,10),B(10,10),C(10,10)
M = 9
N = 8
K = 6
LDA = 10
LDB = 10
LDC = 10
CALL DGEMM ('TRAN','NONTRAN',M,N,K,RHO,A,LDA,B,LDB,0.5D0,C,LDC)

```

Name SGEMMS/DGEMMS/.../ZGEMMS
Strassen Matrix-Matrix Multiply

Purpose These subprograms use Strassen's method to compute the matrix-matrix product AB , where A is an m -by- k matrix and B is a k -by- n matrix. Strassen's method is an algorithm for matrix multiplication which, under certain circumstances, uses fewer than mnk multiplications and additions. These subprograms have argument lists identical to the standard Level 3 BLAS subprograms DGEMM and ZGEMM in VECLIB and CGEMM and SGEMM in VECLIB8. So to convert a program to call a Strassen subprogram instead of a standard matrix multiply, it is only necessary to change the subprogram name. With consistent upper or lower case coding, a simple preprocessor directive can select standard or Strassen matrix multiply calls. Work area management is done by the subprograms.

By using Strassen's method, these subprograms may be considerably faster than their VECLIB and VECLIB8 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 VECLIB:

```

CHARACTER*1  transa, transb
INTEGER*4    m, n, k, lda, ldb, ldc
REAL*8       alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL DGEMMS(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)

CHARACTER*1  transa, transb
INTEGER*4    m, n, k, lda, ldb, ldc
COMPLEX*16   alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL ZGEMMS(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)

```

VECLIB8:

```

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

```

Input

transa Transposition option for A :

'N' or 'n' Use m -by- k matrix A

'T' or 't' Use A^T where A is a k -by- m matrix

'C' or 'c' Use A^* where A is a k -by- m matrix

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

transb Transposition option for B :

'N' or 'n' Use k -by- n matrix B

'T' or 't' Use B^T where B is an n -by- k matrix

'C' or 'c' Use B^* where B is an n -by- k matrix

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

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

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

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

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

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

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

otherwise A is a k -by- m matrix

lda	The leading dimension of array a as declared in the calling program unit, with $\text{lda} \geq \max$ (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 $\text{ldb} \geq \max$ (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 $\text{beta} = 0$.
ldc	The leading dimension of array c as declared in the calling program unit, with $\text{ldc} \geq \max(m, 1)$.

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

Notes Except for the extra character in the subprogram name, these subprograms conform to specifications of the Level 3 BLAS subprograms SGEMM, DGEMM, CGEMM, and ZGEMM.

Because of their use of Strassen's method, SGEMMS, DGEMMS, CGEMMS, and ZGEMMS are asymptotically faster than standard matrix multiply methods such as those employed in the standard routines SGEMM, DGEMM, CGEMM, and ZGEMM. In practice these particular implementations are faster than their standard counterparts if $\min(m, n, k) > 200$ for CGEMMS or ZGEMMS, or $\min(m, n, k) > 512$ for SGEMMS or DGEMMS. The speedup in the complex case is much more pronounced. That is due in large part to the complex bilinear reduction technique (implemented underneath Strassen's method) that allows two complex matrices to be multiplied using only $3/4$ of the multiplications required by the traditional method. Also, the relative cost of data motion is lower in the complex case. The gains in the real case are marginal until n becomes very large.

In the operator norm, Strassen's method is slightly less stable than traditional matrix multiplication, and the computation of individual elements is unstable. The emerging consensus seems to be that Strassen's method is sufficiently stable for most applications. Partly for stability reasons, however, only 64-bit Strassen subprograms are available at this time.

For a good overview and bibliography of this subject, see Higham.

If an error in the arguments is detected, the subprograms call error handler XERBLA, which writes an error message onto the standard error file and terminates execution. The standard version of XERBLA (refer to the end of this chapter) 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', 'TRANSPPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Form the **REAL*8** matrix product $C = AB$, where A is a 900-by-600 real matrix stored in an array A whose dimensions are 1000 by 1000, B is a 600-by-800 real matrix stored in an array B of dimension 1000 by 1000, and C is a 900-by-800 real matrix stored in an array C , also of dimension 1000 by 1000.

```

      CHARACTER*1  TRANSA, TRANSB
      INTEGER*4    M,N,K,LDA,LDB,LDC
      REAL*8       ALPHA,BETA,A(1000,1000),B(1000,1000),
&                C(1000,1000)
      TRANSA = 'N'
      TRANSB = 'N'
      M = 900
      N = 800
      K = 600
      ALPHA = 1.0
      BETA = 0.0
      LDA = 1000
      LDB = 1000
      LDC = 1000
      CALL DGEMMS (TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,
&                BETA,C,LDC)

```

Example 2 Form the COMPLEX*16 matrix product $C = 1/2C + \rho A * B$, where ρ is a complex scalar, A is a 600-by-900 complex matrix stored in an array A whose dimensions are 1000 by 1000, B is a 600-by-800 complex matrix stored in an array B of dimension 1000 by 1000, and C is a 900-by-800 complex matrix stored in an array C , also of dimension 1000 by 1000.

```
INTEGER*4  M,N,K,LDA,LDB,LDC
COMPLEX*16 RHO,A(1000,1000),B(1000,1000),C(1000,1000)
M = 900
N = 800
K = 600
LDA = 1000
LDB = 1000
LDC = 1000
CALL ZGEMMS ('CONJ','NORMAL',M,N,K,RHO,A,LDA,B,LDB,
&            (0.5D0,0.0D0),C,LDC)
```

Name SGEMV/DGEMV/CGEMV/ZGEMV
Matrix-Vector Multiply

Purpose These subprograms compute the matrix-vector products Ax , $A^T x$, 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 added to or subtracted from it. This is handled in a convenient, but general, way by two scalar arguments, α and β , which are used as multipliers of the matrix-vector product and the result vector. Specifically, these subprograms compute matrix-vector products of the forms

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

Usage

VECLIB:

```
CHARACTER*1  trans
INTEGER*4    m, n, lda, incx, incy
REAL*4       alpha, beta, a(lda, n), x(lenx), y(leny)
CALL SGEMV(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
```

```
CHARACTER*1  trans
INTEGER*4    m, n, lda, incx, incy
REAL*8       alpha, beta, a(lda, n), x(lenx), y(leny)
CALL DGEMV(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
```

```
CHARACTER*1  trans
INTEGER*4    m, n, lda, incx, incy
COMPLEX*8    alpha, beta, a(lda, n), x(lenx), y(leny)
CALL CGEMV(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
```

```
CHARACTER*1  trans
INTEGER*4    m, n, lda, incx, incy
COMPLEX*16   alpha, beta, a(lda, n), x(lenx), y(leny)
CALL ZGEMV(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
```

VECLIB8:

```
CHARACTER*1  trans
INTEGER*8    m, n, lda, incx, incy
REAL*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 $y \leftarrow \alpha A^T x + \beta y$ 'C' or 'c' Compute $y \leftarrow \alpha A^* x + \beta y$ where A^T is the transpose of A and A^* is the conjugate transpose. In the real subprograms, 'C' and 'c' have the same meaning as 'T' and 't'.
	m	Number of rows in matrix A, $m \geq 0$. If $m = 0$, the subprograms do not reference a , x , or y .
	n	Number of columns in matrix A, $n \geq 0$. If $n = 0$, the subprograms do not reference a , x , or y .
	alpha	The scalar α . If alpha = 0, the subprograms compute $y \leftarrow \beta y$ without referencing A or x .
	a	Array containing the m -by- n matrix A.
	lda	The leading dimension of array a as declared in the calling program unit, with $lda \geq \max(m, 1)$.
	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 n elements lenx = $(n-1) \times incx + 1$ otherwise x has m elements lenx = $(m-1) \times incx + 1$
	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 $x((i-n) \times incx + 1)$; otherwise, 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.
	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 $\text{leny} = (m-1) \times |\text{incy}| + 1$
 otherwise y has n elements $\text{leny} = (n-1) \times |\text{incy}| + 1$

Not used as input if $\text{beta} = 0$.

incy

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

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

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

Use $\text{incy} = 1$ 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

$\text{trans} \neq \text{'N'}$ or 'n' or 'T' or 't' or 'C' or 'c' ,
 $m < 0$,
 $n < 0$,
 $\text{lda} < \max(m, 1)$,
 $\text{incx} = 0$, and
 $\text{incy} = 0$.

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', 'TRANPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Form the REAL*4 matrix-vector product $y = Ax$, where A is a 9-by-6 real matrix stored in an array A whose dimensions are 10 by 10, x is a real vector 6 elements long stored in an array X of dimension 10, and y is a real vector 9 elements long stored in an array Y , also of dimension 10.

```

CHARACTER*1 TRANS
INTEGER*4    M,N,LDA, INCX, INCY
REAL*4       ALPHA, BETA, A(10,10), X(10), Y(10)
TRANS = 'N'
M = 9
N = 6
ALPHA = 1.0
BETA = 0.0
LDA = 10
INCX = 1
INCY = 1
CALL SGEMV (TRANS, M, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)

```

Example 2 Form the REAL*8 matrix-vector product $y = 1/2y - \rho A^T x$, where ρ is a real scalar, A is a 6-by-9 real matrix stored in an array A whose dimensions are 10 by 10, x is a real vector 6 elements long stored in an array X of dimension 10, and y is a real vector 9 elements long stored in an array Y , also of dimension 10.

```

INTEGER*4 M,N,LDA
REAL*8    RHO, A(10,10), X(10), Y(10)
M = 9
N = 6
LDA = 10
CALL DGEMV ('TRANSPOSE', M, N, -RHO, A, LDA, X, 1, 0.5D0, Y, 1)

```

Name SGER/DGER/CGERC/CGERU/ZGERC/ZGERU
Rank-1 Update

Purpose These subprograms compute the rank-1 updates

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

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 VECLIB:

```
INTEGER*4      m, n, lda, incx, incy
REAL*4         alpha, a(lda, n), x(lenx), y(leny)
CALL SGER(m, n, alpha, x, incx, y, incy, a, lda)
```

```
INTEGER*4      m, n, lda, incx, incy
REAL*8         alpha, a(lda, n), x(lenx), y(leny)
CALL DGER(m, n, alpha, x, incx, y, incy, a, lda)
```

```
INTEGER*4      m, n, lda, incx, incy
COMPLEX*8      alpha, a(lda, n), x(lenx), y(leny)
CALL CGERC(m, n, alpha, x, incx, y, incy, a, lda)
```

```
INTEGER*4      m, n, lda, incx, incy
COMPLEX*8      alpha, a(lda, n), x(lenx), y(leny)
CALL CGERU(m, n, alpha, x, incx, y, incy, a, lda)
```

```
INTEGER*4      m, n, lda, incx, incy
COMPLEX*16     alpha, a(lda, n), x(lenx), y(leny)
CALL ZGERC(m, n, alpha, x, incx, y, incy, a, lda)
```

```
INTEGER*4      m, n, lda, incx, incy
COMPLEX*16     alpha, a(lda, n), x(lenx), y(leny)
CALL ZGERU(m, n, alpha, x, incx, y, incy, a, lda)
```

VECLIB8:

```
INTEGER*8      m, n, lda, incx, incy
REAL*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*, $m \geq 0$. If $m = 0$, the subprograms do not reference *a*, *x*, or *y*.

n Number of columns in matrix *A* and elements of vector *y*, $n \geq 0$. 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 $\text{lenx} = (m-1) \times |\text{incx}| + 1$ containing the *m*-vector *x*.

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

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

$\text{incx} < 0$ *x* is stored backward in array *x*; i.e., x_i is stored in $\mathbf{x}((i-m) \times \text{incx} + 1)$.

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

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

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

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

Use $\text{incy} = 1$ 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.

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

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

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*4 rank-1 update xy^T to **A**, where **A** is a 6-by-9 real matrix stored in an array **A** whose dimensions are 10 by 10, **x** is a real vector 6 elements long stored in an array **X** of dimension 10, and **y** is a real vector 9 elements long stored in an array **Y**, also of dimension 10.

```
INTEGER*4 M,N,LDA,INCX,INCY
REAL*4     ALPHA,A(10,10),X(10),Y(10)
M = 6
N = 9
ALPHA = 1.0
LDA = 10
INCX = 1
INCY = 1
CALL SGER (M,N,ALPHA,X,INCX,Y,INCY,A,LDA)
```

Example 2 Apply a COMPLEX*8 conjugated rank-1 update $-2xy^*$ to **A**, where **A** is a 6-by-9 complex matrix stored in an array **A** whose dimensions are 10 by 10, **x** is a complex vector 6 elements long stored in an array **X** of dimension 10, and **y** is a complex vector 9 elements long stored in an array **Y**, also of dimension 10.

```
INTEGER*4 M,N,LDA
COMPLEX*8 A(10,10),X(10),Y(10)
M = 6
N = 9
LDA = 10
CALL CGERC (M,N,(-2.0E0,0.0E0),X,1,Y,1,A,LDA)
```

Name SSBMV/DSBMV/CHBMV/ZHBMV
Matrix-Vector Multiply

Purpose These subprograms compute the matrix-vector product Ax , where A is an n -by- n real symmetric or complex Hermitian band matrix and x is a real or complex n -vector. The product may be stored in the result array, or it 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-vector product and the result vector. Specifically, these subprograms compute the matrix-vector product of the form

$$y \leftarrow \alpha Ax + \beta y.$$

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

SSBMV	or	DSBMV	A is a real symmetric band matrix
CHBMV	or	ZHBMV	A is a complex Hermitian band matrix

A symmetric or Hermitian band matrix is a symmetric or Hermitian matrix whose nonzero elements all are on or fairly near the principal diagonal. Specifically, $a_{ij} \neq 0$ only if $|i-j| \leq kd$ for some integer kd , called the half bandwidth.

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

VECLIB:

```

CHARACTER*1  uplo
INTEGER*4    n, kd, ldab, incx, incy
REAL*4       alpha, beta, ab(ldab, n), x(lenx), y(leny)
CALL SSBMV(uplo, n, kd, alpha, ab, ldab, x, incx, beta, y, incy)

CHARACTER*1  uplo
INTEGER*4    n, kd, ldab, incx, incy
REAL*8       alpha, beta, ab(ldab, n), x(lenx), y(leny)
CALL DSBMV(uplo, n, kd, alpha, ab, ldab, x, incx, beta, y, incy)

```

```

CHARACTER*1   uplo
INTEGER*4     n, kd, ldab, incx, incy
COMPLEX*8     alpha, beta, ab(ldab, n), x(lenx), y(leny)
CALL CHBMV(uplo, n, kd, alpha, ab, ldab, x, incx, beta, y, incy)

CHARACTER*1   uplo
INTEGER*4     n, kd, ldab, incx, incy
COMPLEX*16    alpha, beta, ab(ldab, n), x(lenx), y(leny)
CALL ZHBMV(uplo, n, kd, alpha, ab, ldab, x, incx, beta, y, incy)

```

VECLIB8:

```

CHARACTER*1   uplo
INTEGER*8     n, kd, ldab, incx, incy
REAL*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, $n \geq 0$. If $n = 0$, the subprograms do not reference ab or x .
kd	The number of nonzero diagonals above or below the principal diagonal.
alpha	The scalar α . If alpha = 0, the subprograms compute $y \leftarrow \beta y$ without referencing ab or x .
ab	Array containing the n -by- n symmetric band matrix A in the compressed form described above. The columns of the band of A are stored in the columns of ab , and the diagonals of the band of A are stored in the rows of ab .
ldab	The leading dimension of array ab as declared in the calling program unit, with $ldab \geq kd+1$.
x	Array of length $lenx = (n-1) \times incx + 1$ 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 $x((i-1) \times \text{incx} + 1)$.

incx < 0 x is stored backward in array x ; i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$.

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 $\text{leny} = (n-1) \times |\text{incy}| + 1$ 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 $y((i-1) \times \text{incy} + 1)$.

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

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*4 matrix-vector product $y = Ax$, where A is a 75-by-75 real symmetric band matrix with half bandwidth 15 whose lower triangular part is stored in an array AB whose dimensions are 25 by 100, and x and y are real vectors 75 elements long stored in arrays X and Y of dimension 100, respectively.

```

      CHARACTER*1 UPLO
      INTEGER*4   N,KD,LDAB, INCX, INCY
      REAL*4      AB(25,100),X(100),Y(100)
      UPLO = 'L'
      N = 75
      KD = 15
      LDAB = 25
      INCX = 1
      INCY = 1
      CALL SSBMV (UPLO, N, KD, 1.0, AB, LDAB, X, INCX, 0.0, Y, INCY)

```

Example 2 Form the REAL*8 matrix-vector product $y = Ax$, where A is a 75-by-75 real symmetric band matrix with half bandwidth 15 whose upper triangle is stored in an array AB whose dimensions are 25 by 100, and x and y are real vectors 75 elements long stored in arrays X and Y of dimension 100, respectively.

```

      INTEGER*4 N,KD,LDAB
      REAL*4    AB(25,100),X(100),Y(100)
      N = 75
      KD = 15
      LDAB = 25
      CALL DSBMV ('UPPER', N, KD, 1.0, AB, LDAB, X, 1, 1.0, Y, 1)

```

Name SSPMV/DSPMV/CHPMV/ZHPMV
Matrix-Vector Multiply

Purpose These subprograms compute the matrix-vector product Ax , where A is an n -by- n real symmetric or complex Hermitian matrix stored in packed form as described in "Matrix Storage," and x is a real or complex n -vector. The product 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	or	DSPMV	A is a real symmetric matrix
CHPMV	or	ZHPMV	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

VECLIB:

```

CHARACTER*1  uplo
INTEGER*4    n, incx, incy
REAL*4       alpha, beta, ap(lenap), x(lenx), y(leny)
CALL SSPMV(uplo, n, alpha, ap, x, incx, beta, y, incy)

```

```

CHARACTER*1  uplo
INTEGER*4    n, incx, incy
REAL*8       alpha, beta, ap(lenap), x(lenx), y(leny)
CALL DSPMV(uplo, n, alpha, ap, x, incx, beta, y, incy)

```

```

CHARACTER*1  uplo
INTEGER*4    n, incx, incy
COMPLEX*8    alpha, beta, ap(lenap), x(lenx), y(leny)
CALL CHPMV(uplo, n, alpha, ap, x, incx, beta, y, incy)

```

```

CHARACTER*1  uplo
INTEGER*4    n, incx, incy
COMPLEX*16   alpha, beta, ap(lenap), x(lenx), y(leny)
CALL ZHPMV(uplo, n, alpha, ap, x, incx, beta, y, incy)

```

VECLIB8:

```

CHARACTER*1  uplo
INTEGER*8    n, incx, incy
REAL*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, $n \geq 0$. If $n = 0$, the subprograms do not reference ap , x , or y .
	alpha	The scalar α . If alpha = 0, the subprograms compute $y \leftarrow \beta y$ without referencing ap or x .
	ap	Array of length lenap = $n \times (n+1)/2$ containing the upper or lower triangle, as specified by uplo , of an n -by- n real symmetric or complex Hermitian matrix A, stored by columns in the packed form described above.
	x	Array of length lenx = $(n-1) \times \text{incx} + 1$ 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 $\mathbf{x}((i-1) \times \text{incx} + 1)$. incx < 0 x is stored backward in array x ; i.e., x_i is stored in $\mathbf{x}((i-n) \times \text{incx} + 1)$. Use incx = 1 if the vector x is stored contiguously in array x , i.e., if x_i is stored in $\mathbf{x}(i)$. Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.
	beta	The scalar β .
	y	Array of length leny = $(n-1) \times \text{incy} + 1$ 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 $\mathbf{y}((i-1) \times \text{incy} + 1)$. incy < 0 y is stored backward in array y ; i.e., y_i is stored in $\mathbf{y}((i-n) \times \text{incy} + 1)$.

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,
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*4 matrix-vector product $y = Ax$, where **A** is a 9-by-9 real symmetric matrix whose upper triangle is stored in packed form in an array **AP** of dimension 55, **x** is a real vector 9 elements long stored in an array **X** of dimension 10, and **y** is a real vector 9 elements long stored in an array **Y**, also of dimension 10.

```
CHARACTER*1 UPLO
INTEGER*4   N, INCX, INCY
REAL*4      ALPHA, BETA, AP(55), X(10), Y(10)
UPLO = 'U'
N = 9
ALPHA = 1.0
BETA = 0.0
INCX = 1
INCY = 1
CALL SSPMV (UPLO, N, ALPHA, AP, X, INCX, BETA, Y, INCY)
```

Example 2 Form the COMPLEX*8 matrix-vector product $y = 1/2y - \rho Ax$, where ρ is a complex scalar, A is a 9-by-9 complex Hermitian matrix whose lower triangle is stored in packed form in an array AP of dimension 55, x is a complex vector 9 elements long stored in an array X of dimension 10, and y is a complex vector 9 elements long stored in an array Y , also of dimension 10.

```
INTEGER*4 N
COMPLEX*8 RHO,AP(55),X(10),Y(10)
N = 9
CALL CHPMV ('LOWER',N,-RHO,AP,X,1,(0.5D0,0.0D0),Y,1)
```

Name SSPR/DSPR/CHPR/ZHPR
Rank-1 Update

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

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

where A is an n -by- n real symmetric or complex Hermitian matrix stored in packed form as described in "Matrix Storage," α is a real scalar, x is a real or complex n -vector, and x^* is the conjugate transpose of x . (The conjugate transpose of a real vector is simply the transpose.)

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

SSPR	or	DSPR	A is a real symmetric matrix
CHPR	or	ZHPR	A is a complex Hermitian matrix

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 $\mathbf{ap}$ as follows:

k	1	2	3	4	5	6	7	8	9	10
$\mathbf{ap}(k)$	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

VECLIB:

```

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

```

```

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

```

```

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

```

```

CHARACTER*1  uplo
INTEGER*4    n, incx
REAL*8       alpha
COMPLEX*16   ap(lenap), x(lenx)
CALL ZHPR(uplo, n, alpha, x, incx, ap)

```

VECLIB8:

```

CHARACTER*1   uplo
INTEGER*8     n, incx
REAL*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	<table> <tr> <td data-bbox="392 593 449 624">uplo</td><td data-bbox="614 593 1306 772"> 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. </td></tr> <tr> <td data-bbox="392 781 414 807">n</td><td data-bbox="614 781 1306 876"> Number of rows and columns in matrix A and elements of vector x, $n \geq 0$. If $n = 0$, the subprograms do not reference ap or x. </td></tr> <tr> <td data-bbox="392 885 464 911">alpha</td><td data-bbox="614 885 1306 946"> The scalar α. If alpha = 0, the subprograms do not reference ap or x. </td></tr> <tr> <td data-bbox="392 954 406 980">x</td><td data-bbox="614 954 1306 1015"> Array of length $\text{lenx} = (n-1) \times \text{incx} + 1$ containing the n-vector x. </td></tr> <tr> <td data-bbox="392 1024 449 1050">incx</td><td data-bbox="614 1024 1306 1362"> Increment for the array x, incx $\neq 0$: incx > 0 x is stored forward in array x; i.e., x_i is stored in $\mathbf{x}((i-1) \times \text{incx} + 1)$. incx < 0 x is stored backward in array x; i.e., x_i is stored in $\mathbf{x}((i-n) \times \text{incx} + 1)$. Use incx = 1 if the vector x is stored contiguously in array x, i.e., if x_i is stored in $\mathbf{x}(i)$. Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2. </td></tr> <tr> <td data-bbox="392 1371 421 1397">ap</td><td data-bbox="614 1371 1306 1499"> Array of length $\text{lenap} = n \times (n+1)/2$ 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. </td></tr> </table>	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, $n \geq 0$. If $n = 0$, 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 $\text{lenx} = (n-1) \times \text{incx} + 1$ 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 $\mathbf{x}((i-1) \times \text{incx} + 1)$. incx < 0 x is stored backward in array x; i.e., x_i is stored in $\mathbf{x}((i-n) \times \text{incx} + 1)$. Use incx = 1 if the vector x is stored contiguously in array x, i.e., if x_i is stored in $\mathbf{x}(i)$. Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.	ap	Array of length $\text{lenap} = n \times (n+1)/2$ 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.
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, $n \geq 0$. If $n = 0$, 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 $\text{lenx} = (n-1) \times \text{incx} + 1$ 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 $\mathbf{x}((i-1) \times \text{incx} + 1)$. incx < 0 x is stored backward in array x; i.e., x_i is stored in $\mathbf{x}((i-n) \times \text{incx} + 1)$. Use incx = 1 if the vector x is stored contiguously in array x, i.e., if x_i is stored in $\mathbf{x}(i)$. Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.												
ap	Array of length $\text{lenap} = n \times (n+1)/2$ 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 $\neq$ 'L' or 'l' 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 **uplo** argument as 'LOWER' for 'L' or 'UPPER' for 'U'. Refer to "Example 2."

Example 1 Apply a REAL*4 symmetric rank-1 update xx^T to A , where A is a 9-by-9 real symmetric matrix whose upper triangle is stored in packed form in an array **AP** of dimension 55, and x is a real vector 9 elements long stored in an array **X** of dimension 10.

```
CHARACTER*1 UPLO
INTEGER*4   N, INCX
REAL*4      ALPHA, AP(55), X(10)
UPLO = 'U'
N = 9
ALPHA = 1.0
INCX = 1
CALL SSPR (UPLO, N, ALPHA, X, INCX, AP)
```

Example 2 Apply a COMPLEX*8 Hermitian rank-1 update $-2xx^*$ to A , where A is a 9-by-9 complex Hermitian matrix whose lower triangle is stored in packed form in an array **AP** of dimension 55, and x is a complex vector 9 elements long stored in an array **X** of dimension 10.

```
INTEGER*4 N
COMPLEX*8 AP(55), X(10)
N = 9
CALL CHPR ('LOWER', N, -2.0, X, 1, AP)
```

Name SSPR2/DSPR2/CHPR2/ZHPR2
Rank-2 Update

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

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

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

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

SSPR2	or	DSPR2	A is a real symmetric matrix
CHPR2	or	ZHPR2	A is a complex Hermitian matrix

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

VECLIB:

```

CHARACTER*1  uplo
INTEGER*4    n, incx, incy
REAL*4       alpha, ap(lenap), x(lenx), y(leny)
CALL SSPR2(uplo, n, alpha, x, incx, y, incy, ap)

```

```

CHARACTER*1  uplo
INTEGER*4    n, incx, incy
REAL*8       alpha, ap(lenap), x(lenx), y(leny)
CALL DSPR2(uplo, n, alpha, x, incx, y, incy, ap)

```

```

CHARACTER*1  uplo
INTEGER*4    n, incx, incy
COMPLEX*8    alpha, ap(lenap), x(lenx), y(leny)
CALL CHPR2(uplo, n, alpha, x, incx, y, incy, ap)

```

```

CHARACTER*1  uplo
INTEGER*4    n, incx, incy
COMPLEX*16   alpha, ap(lenap), x(lenx), y(leny)
CALL ZHPR2(uplo, n, alpha, x, incx, y, incy, ap)

```

VECLIB8:

```

CHARACTER*1  uplo
INTEGER*8    n, incx, incy
REAL*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 , $n \geq 0$. If $n = 0$, the subprograms do
 not reference ap , x , or y .

alpha The scalar α . If $\alpha = 0$, the subprograms do not
 reference ap , x , or y .

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

incx Increment for the array x , $\text{incx} \neq 0$:
 $\text{incx} > 0$ x is stored forward in array x ; i.e., x_i is
 stored in $x((i-1) \times \text{incx} + 1)$.
 $\text{incx} < 0$ x is stored backward in array x ; i.e., x_i
 is stored in $x((i-n) \times \text{incx} + 1)$.
 Use $\text{incx} = 1$ 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 $\text{leny} = (n-1) \times |\text{incy}| + 1$ containing the
 n -vector y .

incy Increment for the array y , $\text{incy} \neq 0$:
 $\text{incy} > 0$ y is stored forward in array y ; i.e., y_i is
 stored in $y((i-1) \times \text{incy} + 1)$.
 $\text{incy} < 0$ y is stored backward in array y ; i.e., y_i
 is stored in $y((i-n) \times \text{incy} + 1)$.
 Use $\text{incy} = 1$ 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.

ap Array of length $\text{lenap} = n \times (n+1)/2$ 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 $\neq$ '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 Apply a REAL*4 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*4   N, INCX, INCY
REAL*4      ALPHA, AP(55), X(10), Y(10)
UPLO = 'U'
N = 9
ALPHA = 1.0
INCX = 1
INCY = 1
CALL SSPR2 (UPLO, N, ALPHA, X, INCX, Y, INCY, AP)
```

Example 2 Apply a COMPLEX*8 Hermitian rank-2 update $\alpha xy^* + \bar{\alpha} yx^*$ to A , where A is a 9-by-9 complex Hermitian matrix whose lower triangle is stored in packed form in an array AP of dimension 55, α is a complex scalar, x is a complex vector 9 elements long stored in an array X of dimension 10, and y is a complex vector 9 elements long stored in an array Y of dimension 10.

```
INTEGER*4 N
COMPLEX*8 ALPHA,AP(55),X(10),Y(10)
N = 9
CALL CHPR2 ('LOWER',N,ALPHA,X,1,Y,1,AP)
```


Name SSYMM/DSYMM/CHEMM/.../ZSYMM
Matrix-Matrix Multiply

Purpose These subprograms compute the matrix-matrix products AB and BA , where A is a real symmetric, complex symmetric, or complex Hermitian matrix, and B is an m -by- n matrix. The size of A , either m by m or n by n , depends on which matrix product is requested. The product 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	or	DSYMM	A is a real symmetric matrix
CHEMM	or	ZHEMM	A is a complex Hermitian matrix
CSYMM	or	ZSYMM	A is a complex symmetric matrix

Matrix Storage Because either triangle of A 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 VECLIB:

```

CHARACTER*1  side, uplo
INTEGER*4    m, n, lda, ldb, ldc
REAL*4       alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL SSYMM(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)

CHARACTER*1  side, uplo
INTEGER*4    m, n, lda, ldb, ldc
REAL*8       alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL DSYMM(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)

CHARACTER*1  side, uplo
INTEGER*4    m, n, lda, ldb, ldc
COMPLEX*8    alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL CHEMM(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)

```

```

CHARACTER*1   side, uplo
INTEGER*4    m, n, lda, ldb, ldc
COMPLEX*8    alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL CSYMM(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)

CHARACTER*1   side, uplo
INTEGER*4    m, n, lda, ldb, ldc
COMPLEX*16   alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL ZHEMM(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)

CHARACTER*1   side, uplo
INTEGER*4    m, n, lda, ldb, ldc
COMPLEX*16   alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL ZSYMM(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)

```

VECLIB8:

```

CHARACTER*1   side, uplo
INTEGER*8    m, n, lda, ldb, ldc
REAL*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 <i>A</i> is the left or right matrix operand: 'L' or 'l' <i>A</i> is the left matrix operand, i.e., compute $C \leftarrow \alpha AB + \beta C$ 'R' or 'r' <i>A</i> is the right matrix operand, i.e., compute $C \leftarrow \alpha BA + \beta C$
	uplo	Upper/lower triangular storage option for <i>A</i> : 'L' or 'l' Reference only the lower triangle of <i>A</i> 'U' or 'u' Reference only the upper triangle of <i>A</i>
	m	Number of rows in matrix <i>C</i> , $m \geq 0$. If $m = 0$, the subprograms do not reference <i>a</i> , <i>b</i> , or <i>c</i> .

n	Number of columns in matrix B , $n \geq 0$. If $n = 0$, the subprograms do not reference a , b , or c .
alpha	The scalar α . If alpha = 0, the subprograms compute $C \leftarrow \beta C$ without referencing a or b .
a	Array whose upper or lower triangle, as specified by uplo , contains the upper or lower triangle of the matrix A . The other triangle of a is not referenced. The size of A is indicated by side : 'L' or 'l' A is m by m 'R' or 'r' A is n by n
lda	The leading dimension of array a as declared in the calling program unit, with $lda \geq \max$ (the number of rows of A , 1).
b	Array containing the m -by- n matrix B .
ldb	The leading dimension of array b as declared in the calling program unit, with $ldb \geq \max(m, 1)$.
beta	The scalar β .
c	Array containing the m -by- n matrix C . Not used as input if beta = 0.
ldc	The leading dimension of array c as declared in the calling program unit, with $ldc \geq \max(m, 1)$.

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*4 matrix product $C = AB$, where A is a 6-by-6 real symmetric real matrix whose upper triangle is stored in the upper triangle of an array A of dimension 10 by 10, B is a 6-by-8 real matrix stored in an array B of dimension 10 by 10, and C is a 6-by-8 real matrix stored in an array C , also of dimension 10 by 10.

```

CHARACTER*1 SIDE,UPLO
INTEGER*4    M,N,LDA,LDB,LDC
REAL*4       ALPHA,BETA,A(10,10),B(10,10),C(10,10)
SIDE = 'L'
UPLO = 'U'
M = 6
N = 8
ALPHA = 1.0
BETA = 0.0
LDA = 10
LDB = 10
LDC = 10
CALL SSYMM (SIDE,UPLO,M,N,ALPHA,A,LDA,B,LDB,BETA,C,LDC)

```

Example 2 Form the COMPLEX*8 matrix-matrix product $C = 1/2BA - \rho C$, where ρ is a scalar, A is an 8-by-8 complex Hermitian matrix whose lower triangle is stored in the lower triangle of an array A of dimension 10 by 10, B is a 6-by-8 complex matrix stored in an array whose dimensions are 10 by 10, and C is a 6-by-8 complex matrix stored in an array C , also of dimension 10 by 10.

```

INTEGER*4 M,N,LDA,LDB,LDC
COMPLEX*8 HALF,RHO,A(10,10),B(10,10),C(10,10)
M = 6
N = 8
LDA = 10
LDB = 10
LDC = 10
HALF = (0.5,0.0)
CALL CHEMM ('RIGHT','LOWER',M,N,-RHO,A,LDA,B,LDB,HALF,C,LDC)

```

Name SSYMV/DSYMV/CHEMV/ZHEMV
Matrix-Vector Multiply

Purpose These subprograms compute the matrix-vector product Ax , where A is an n -by- n real symmetric or complex Hermitian matrix, and x is a real or complex n -vector. The product 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	or	DSYMV	A is a real symmetric matrix
CHEMV	or	ZHEMV	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 VECLIB:

```

CHARACTER*1  uplo
INTEGER*4    n, lda, incx, incy
REAL*4       alpha, beta, a(lda, n), x(lenx), y(leny)
CALL SSYMV(uplo, n, alpha, a, lda, x, incx, beta, y, incy)

CHARACTER*1  uplo
INTEGER*4    n, lda, incx, incy
REAL*8       alpha, beta, a(lda, n), x(lenx), y(leny)
CALL DSYMV(uplo, n, alpha, a, lda, x, incx, beta, y, incy)

CHARACTER*1  uplo
INTEGER*4    n, lda, incx, incy
COMPLEX*8    alpha, beta, a(lda, n), x(lenx), y(leny)
CALL CHEMV(uplo, n, alpha, a, lda, x, incx, beta, y, incy)

CHARACTER*1  uplo
INTEGER*4    n, lda, incx, incy
COMPLEX*16   alpha, beta, a(lda, n), x(lenx), y(leny)
CALL ZHEMV(uplo, n, alpha, a, lda, x, incx, beta, y, incy)

```

VECLIB8:

```

      CHARACTER*1  uplo
      INTEGER*8    n, lda, incx, incy
      REAL*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, $n \geq 0$. 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 $lenx = (n-1) \times incx + 1$ 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 $\text{leny} = (n-1) \times |\text{incy}| + 1$ containing the n -vector y . Not used as input if $\text{beta} = 0$.

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

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

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

Use $\text{incy} = 1$ 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."

Example 1 Form the REAL*4 matrix-vector product $y = Ax$, where A is a 9-by-9 real symmetric matrix whose upper triangle is stored in the upper triangle of an array A whose dimensions are 10 by 10, x is a real vector 9 elements long stored in an array X of dimension 10, and y is a real vector 9 elements long stored in an array Y , also of dimension 10.

```

CHARACTER*1 UPLO
INTEGER*4   N,LDA,INCX,INCY
REAL*4      ALPHA,BETA,A(10,10),X(10),Y(10)
UPLO = 'U'
N = 9
ALPHA = 1.0
BETA = 0.0
LDA = 10
INCX = 1
INCY = 1
CALL SSYMV (UPLO,N,ALPHA,A,LDA,X,INCX,BETA,Y,INCY)

```

Example 2 Form the COMPLEX*8 matrix-vector product $y = 1/2y - \rho Ax$, where ρ is a complex scalar, A is a 9-by-9 complex Hermitian matrix whose lower triangle is stored in the lower triangle of an array A whose dimensions are 10 by 10, x is a complex vector 9 elements long stored in an array X of dimension 10, and y is a complex vector 9 elements long stored in an array Y , also of dimension 10.

```

INTEGER*4 N,LDA
COMPLEX*8 RHO,A(10,10),X(10),Y(10)
N = 9
LDA = 10
CALL CHEMV ('LOWER',N,-RHO,A,LDA,X,1,(0.5,0.0),Y,1)

```


Name SSYR/DSYR/CHER/ZHER
Rank-1 Update

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

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

where A is an n -by- n real symmetric or complex Hermitian matrix, α is a real scalar, x is a real or complex n -vector, and x^* is the conjugate transpose of x . (The conjugate transpose of a real vector is simply the transpose.)

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

SSYR	or	DSYR	A is a real symmetric matrix
CHER	or	ZHER	A is a complex Hermitian matrix

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 VECLIB:

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

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

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

```
CHARACTER*1  uplo
INTEGER*4    n, lda, incx
REAL*8       alpha
COMPLEX*16   a(lda, n), x(lenx)
CALL ZHER(uplo, n, alpha, x, incx, a, lda)
```

VECLIB8:

```

CHARACTER*1  uplo
INTEGER*8    n, lda, incx
REAL*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 , $n \geq 0$. If $n = 0$, the subprograms do not reference a or x .

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

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

incx Increment for the array x , $\text{incx} \neq 0$:
 $\text{incx} > 0$ x is stored forward in array x ; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$.
 $\text{incx} < 0$ x is stored backward in array x ; i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$.
 Use $\text{incx} = 1$ 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 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 $\text{lda} \geq \max(n, 1)$.

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*4 symmetric rank-1 update xx^T to *A*, where *A* is a 9-by-9 real symmetric matrix whose upper triangle is stored in the upper triangle of an array *A* whose dimensions are 10 by 10, and *x* is a real vector 9 elements long stored in an array *X* of dimension 10.

```
CHARACTER*1 UPLO
INTEGER*4   N,LDA, INCX
REAL*4      ALPHA,A(10,10),X(10)
UPLO = 'U'
N = 9
ALPHA = 1.0
LDA = 10
INCX = 1
CALL SSYR (UPLO,N,ALPHA,X,INCX,A,LDA)
```

Example 2 Apply a COMPLEX*8 Hermitian rank-1 update $-2xx^*$ to *A*, where *A* is a 9-by-9 complex Hermitian matrix whose lower triangle is stored in the lower triangle of an array *A* whose dimensions are 10 by 10, and *x* is a complex vector 9 elements long stored in an array *X* of dimension 10.

```
INTEGER*4 N,LDA
COMPLEX*8 A(10,10),X(10)
N = 9
LDA = 10
CALL CHER ('LOWER',N,-2.0,X,1,A,LDA)
```

Name SSYR2/DSYR2/CHER2/ZHER2
Rank-2 Update

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

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

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

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

SSYR2	or	DSYR2	A is a real symmetric matrix
CHER2	or	ZHER2	A is a complex Hermitian matrix

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 VECLIB:

```

CHARACTER*1  uplo
INTEGER*4    n, lda, incx, incy
REAL*4       alpha, a(lda, n), x(lenx), y(leny)
CALL SSYR2(uplo, n, alpha, x, incx, y, incy, a, lda)

CHARACTER*1  uplo
INTEGER*4    n, lda, incx, incy
REAL*8       alpha, a(lda, n), x(lenx), y(leny)
CALL DSYR2(uplo, n, alpha, x, incx, y, incy, a, lda)

CHARACTER*1  uplo
INTEGER*4    n, lda, incx, incy
COMPLEX*8    alpha, a(lda, n), x(lenx), y(leny)
CALL CHER2(uplo, n, alpha, x, incx, y, incy, a, lda)

CHARACTER*1  uplo
INTEGER*4    n, lda, incx, incy
COMPLEX*16   alpha, a(lda, n), x(lenx), y(leny)
CALL ZHER2(uplo, n, alpha, x, incx, y, incy, a, lda)

```

VECLIB8:

```

CHARACTER*1  uplo
INTEGER*8    n, lda, incx, incy
REAL*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 , $n \geq 0$. 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 $\text{lenx} = (n-1) \times \text{incx} + 1$ containing the n -vector x .
	incx	Increment for the array x , $\text{incx} \neq 0$: incx > 0 x is stored forward in array x; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$. incx < 0 x is stored backward in array x; i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$. 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 $\text{leny} = (n-1) \times \text{incy} + 1$ containing the n -vector y .
	incy	Increment for the array y , $\text{incy} \neq 0$: incy > 0 y is stored forward in array y; i.e., y_i is stored in $y((i-1) \times \text{incy} + 1)$.

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

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 $\text{lda} \geq \max(n, 1)$.

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 $\neq$ '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'.

Example 1 Apply a REAL*4 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*4   N,LDA, INCX, INCY
REAL*4      ALPHA,A(10,10),X(10),Y(10)
UPLO = 'U'
N = 9
ALPHA = 1.0
LDA = 10
INCX = 1
INCY = 1
CALL SSYR2 (UPLO,N,ALPHA,X,INCX,Y,INCY,A,LDA)

```

Example 2 Apply a COMPLEX*8 Hermitian rank-2 update $\alpha xy^* + \bar{\alpha} yx^*$ to A , where A is a 9-by-9 complex Hermitian matrix whose lower triangle is stored in the lower triangle of an array A whose dimensions are 10 by 10, α is a complex scalar, x is a complex vector 9 elements long stored in an array X of dimension 10, and y is a complex vector 9 elements long stored in an array Y of dimension 10.

```

INTEGER*4 N,LDA
COMPLEX*8 ALPHA,A(10,10),X(10),Y(10)
N = 9
LDA = 10
CALL CHER2 ('LOWER',N,ALPHA,X,1,Y,1,A,LDA)

```

Name SSYR2K/DSYR2K/CHER2K/CSYR2K/.../ZSYR2K
Rank-2k Update

Purpose These subprograms apply a symmetric or Hermitian rank- $2k$ update to a real symmetric, complex symmetric, or complex Hermitian matrix; specifically they compute the following operations:

for symmetric C : $C \leftarrow \alpha AB^T + \bar{\alpha} AB^T + \beta C$ and $C \leftarrow \alpha A^T B + \bar{\alpha} B^T A + \beta C$

for Hermitian C : $C \leftarrow \alpha AB^* + \bar{\alpha} BA^* + \beta C$ and $C \leftarrow \alpha A^* B + \bar{\alpha} B^* A + \beta C$

where α and β are scalars, $\bar{\alpha}$ is the complex conjugate of α , C is an n -by- n real symmetric, complex symmetric, or complex Hermitian matrix, and A and B are matrices whose size, either n by k or k by n , depends on which form of the update is requested. Here, A^T and B^T are the transposes and A^* and B^* are the conjugate transposes of A and B , respectively. (The conjugate of a real scalar is just the scalar, and the conjugate transpose of a real matrix is simply the transpose.)

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

SSYR2K	or	DSYR2K	C is a real symmetric matrix
CHER2K	or	ZHER2K	C is a complex Hermitian matrix
CSYR2K	or	ZSYR2K	C is a complex symmetric matrix

Matrix Storage Because either triangle of C 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 VECLIB:

```

    CHARACTER*1  uplo, trans
    INTEGER*4    n, k, lda, ldb, ldc
    REAL*4       alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
    CALL SSYR2K(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)

    CHARACTER*1  uplo, trans
    INTEGER*4    n, k, lda, ldb, ldc
    REAL*8       alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
    CALL DSYR2K(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)

```



```

CHARACTER*1   uplo, trans
INTEGER*4     n, k, lda, ldb, ldc
REAL*4        beta
COMPLEX*8     alpha, a(lda, *), b(ldb, *), c(ldc, n)
CALL CHER2K(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)

```

```

CHARACTER*1   uplo, trans
INTEGER*4     n, k, lda, ldb, ldc
COMPLEX*8     alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL CSYR2K(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)

```

```

CHARACTER*1   uplo, trans
INTEGER*4     n, k, lda, ldb, ldc
REAL*8        beta
COMPLEX*16    alpha, a(lda, *), b(ldb, *), c(ldc, n)
CALL ZHER2K(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)

```

```

CHARACTER*1   uplo, trans
INTEGER*4     n, k, lda, ldb, ldc
COMPLEX*16    alpha, beta, a(lda, *), b(ldb, *), c(ldc, n)
CALL ZSYR2K(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)

```

VECLIB8:

```

CHARACTER*1   uplo, trans
INTEGER*8     n, k, lda, ldb, ldc
REAL*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        beta
COMPLEX*16    alpha, a(lda, *), b(ldb, *), c(ldc, n)
CALL CHER2K(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)

```

```

CHARACTER*1   uplo, trans
INTEGER*8     n, k, lda, ldb, ldc
COMPLEX*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 $C \leftarrow \alpha AB^T + \bar{\alpha} BA^T + \beta C$ ' T ' or ' t ' Compute $C \leftarrow \alpha A^T B + \bar{\alpha} B^T A + \beta C$ ' C ' or ' c ' Compute $C \leftarrow \alpha A * B + \bar{\alpha} B * A + \beta C$ ' T ' and ' t ' are invalid in subprograms CHER2K and ZHER2K , and ' C ' and ' c ' are invalid in subprograms CSYR2K and ZSYR2K . In subprograms SSYR2K and DSYR2K , ' C ' and ' c ' have the same meaning as ' T ' and ' t '.
n	Number of rows and columns in matrix <i>C</i> , $n \geq 0$. If $n = 0$, the subprograms do not reference a , b , or c .
k	Number of rows or columns in matrices <i>A</i> and <i>B</i> , 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 <i>A</i> , whose size is indicated by trans : ' N ' or ' n ' <i>A</i> is n by k otherwise <i>A</i> 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 <i>A</i> ,1).
b	Array containing matrix <i>B</i> , which is the same size as matrix <i>A</i> . Refer to the description of a 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 <i>B</i> ,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 <i>C</i> . 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, CSYR2K, ZHER2K, and ZSYR2K.

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*4 rank-6 update $AB^T + BA^T$ to an 8-by-8 real symmetric matrix *C* whose upper triangle is stored in the upper triangle of an array **C** of dimension 10 by 10, where *A* is an 8-by-3 real matrix stored in an array **A**, also of dimension 10 by 10.

```

CHARACTER*1 UPLO,TRANS
INTEGER*4   N,K,LDA,LDB,LDC
REAL*4      ALPHA,BETA,A(10,10),B(10,10),C(10,10)
UPLO = 'U'
TRANS = 'N'
N = 8
K = 3
ALPHA = 1.0
BETA = 1.0
LDA = 10
LDB = 10
LDC = 10
CALL SSYR2K (UPLO,TRANS,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)

```

Example 2 Apply a COMPLEX*8 Hermitian rank-4 update $-2AB^*-2BA^*$ to a 9-by-9 complex Hermitian matrix C whose lower triangle is stored in the lower triangle of an array C of dimension 10 by 10, where A is a 9-by-2 complex matrix stored in an array A of dimension 10 by 10.

```
INTEGER*4 N,K,LDA,LDB,LDC
COMPLEX*8 A(10,10),B(10,10),C(10,10)
N = 9
K = 2
LDA = 10
LDB = 10
LDC = 10
CALL CHER2K ('LOWER', 'NONTRANS', N, K, -2.0, A, LDA, B, LDB,
& 1.0, C, LDC)
```

Name SSYRK/DSYRK/CHERK/CSYRK/ZHERK/ZSYRK
Rank-k Update

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

$$\begin{aligned} C &\leftarrow \alpha A A^T + \beta C, & C &\leftarrow \alpha A^T A + \beta C, \\ C &\leftarrow \alpha A A^* + \beta C, & C &\leftarrow \alpha A^* A + \beta C, \end{aligned}$$

where α and β are scalars, C is an n -by- n real symmetric, complex symmetric, or complex Hermitian matrix, and A is a matrix whose size, either n by k or k by n , depends on which form of the update is requested. Here, A^T and A^* are the transpose and conjugate transpose of A , respectively.

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

SSYRK	or	DSYRK	C is a real symmetric matrix
CHERK	or	ZHERK	C is a complex Hermitian matrix
CSYRK	or	ZSYRK	C is a complex symmetric matrix

Matrix Storage Because either triangle of C 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 VECLIB:

```

CHARACTER*1  uplo, trans
INTEGER*4    n, k, lda, ldc
REAL*4       alpha, beta, a(lda, *), c(ldc, n)
CALL SSYRK(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)

```

```

CHARACTER*1  uplo, trans
INTEGER*4    n, k, lda, ldc
REAL*8       alpha, beta, a(lda, *), c(ldc, n)
CALL DSYRK(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)

```

```

CHARACTER*1  uplo, trans
INTEGER*4    n, k, lda, ldc
REAL*4       alpha, beta
COMPLEX*8    a(lda, *), c(ldc, n)
CALL CHERK(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)

```

```

CHARACTER*1  uplo, trans
INTEGER*4    n, k, lda, ldc
COMPLEX*8    alpha, beta, a(lda, *), c(ldc, n)
CALL CSYRK(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)

```

```

CHARACTER*1  uplo, trans
INTEGER*4    n, k, lda, ldc
REAL*8       alpha, beta
COMPLEX*16   a(lda, *), c(ldc, n)
CALL ZHERK(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)

```

```

CHARACTER*1  uplo, trans
INTEGER*4    n, k, lda, ldc
COMPLEX*16   alpha, beta, a(lda, *), c(ldc, n)
CALL ZSYRK(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)

```

VECLIB8:

```

CHARACTER*1  uplo, trans
INTEGER*8    n, k, lda, ldc
REAL*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 subprograms CHERK and ZHERK, and 'C' and 'c' are invalid in subprograms CSYRK and ZSYRK. In subprograms SSYRK and DSYRK, 'C' and 'c' have the same meaning as 'T' and 't'.
	n	Number of rows and columns in matrix <i>C</i> , $n \geq 0$. If $n = 0$, the subprograms do not reference <i>a</i> or <i>c</i> .
	k	Number of rows or columns in matrix <i>A</i> , $k \geq 0$, depending on <i>trans</i> ; refer to description of <i>A</i> for details. If $k = 0$, the subprograms do not reference <i>a</i> .
	alpha	The scalar α . If alpha = 0, the subprograms compute $C \leftarrow \beta C$ without referencing <i>a</i> .
	a	Array containing the matrix <i>A</i> , whose size is indicated by <i>trans</i> : 'N' or 'n' <i>A</i> is <i>n</i> by <i>k</i> otherwise <i>A</i> is <i>k</i> by <i>n</i>
	lda	The leading dimension of array <i>a</i> as declared in the calling program unit, with $lda \geq \max(\text{the number of rows of } A, 1)$.
	beta	The scalar β .
	c	Array whose upper or lower triangle, as specified by uplo , contains the upper or lower triangle of the <i>n</i> -by- <i>n</i> symmetric or Hermitian matrix <i>C</i> . Not used as input if beta = 0.
	ldc	The leading dimension of array <i>c</i> as declared in the calling program unit, with $ldc \geq \max(n, 1)$.
Output	c	The upper or lower triangle of the updated <i>C</i> matrix, as specified by uplo , replaces the upper or lower triangle of the input, respectively. The other triangle of <i>c</i> 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, CSYRK, ZHERK, and ZSYRK.

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*4 rank-6 update AA^T to an 8-by-8 real symmetric matrix *C* whose upper triangle is stored in the upper triangle of an array *C* of dimension 10 by 10, where *A* is an 8-by-6 real matrix stored in an array *A*, also of dimension 10 by 10.

```

CHARACTER*1 UPLO, TRANS
INTEGER*4   N, K, LDA, LDC
REAL*4      ALPHA, BETA, A(10,10), C(10,10)
UPLO = 'U'
TRANS = 'N'
N = 8
K = 6
ALPHA = 1.0
BETA = 1.0
LDA = 10
LDC = 10
CALL SSYRK (UPLO, TRANS, N, K, ALPHA, A, LDA, BETA, C, LDC)

```

Example 2 Apply a COMPLEX*8 Hermitian rank-2 update $-2AA^*$ to a 9-by-9 complex Hermitian matrix *C* whose lower triangle is stored in the lower triangle of an array *C* of dimension 10 by 10, where *A* is a 9-by-2 complex matrix stored in an array *A* of dimension 10 by 10.

```

INTEGER*4 N, K, LDA, LDC
COMPLEX*8 A(10,10), C(10,10)
N = 9
K = 2
LDA = 10
LDC = 10
CALL CHERK ('LOWER', 'NONTRANS', N, K, -2.0, A, LDA, 1.0, C, LDC)

```


Name STBMV/DTBMV/CTBMV/ZTBMV
Matrix-Vector Multiply

Purpose Given an n -by- n upper- or lower-triangular band matrix A and an n -vector x , these subprograms compute the matrix-vector products Ax , $A^T x$, 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

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

A lower-triangular band matrix is a matrix whose strict upper triangle is zero and whose nonzero lower-triangular elements all are on or fairly near the principal diagonal. Specifically, $a_{ij} \neq 0$ only if $0 \leq i-j \leq kd$ for some integer kd .

In contrast, an upper-triangular band matrix is a matrix whose strict lower triangle is zero and whose nonzero upper-triangular elements all are on or fairly near the principal diagonal, 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

VECLIB:

```

CHARACTER*1  uplo, trans, diag
INTEGER*4    n, kd, ldab, incx
REAL*4       ab(ldab, n), x(lenx)
CALL STBMV(uplo, trans, diag, n, kd, ab, ldab, x, incx)

```

```

CHARACTER*1   uplo, trans, diag
INTEGER*4     n, kd, ldab, incx
REAL*8        ab(ldab, n), x(lenx)
CALL DTBMV(uplo, trans, diag, n, kd, ab, ldab, x, incx)

```

```

CHARACTER*1   uplo, trans, diag
INTEGER*4     n, kd, ldab, incx
COMPLEX*8     ab(ldab, n), x(lenx)
CALL CTBMV(uplo, trans, diag, n, kd, ab, ldab, x, incx)

```

```

CHARACTER*1   uplo, trans, diag
INTEGER*4     n, kd, ldab, incx
COMPLEX*16    ab(ldab, n), x(lenx)
CALL ZTBMV(uplo, trans, diag, n, kd, ab, ldab, x, incx)

```

VECLIB8:

```

CHARACTER*1   uplo, trans, diag
INTEGER*8     n, kd, ldab, incx
REAL*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 $lenx = (n-1) \times |incx| + 1$ 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 $x((i-1) \times incx + 1)$.

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** 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', 'TRANPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Form the **REAL*4** matrix-vector product Ax , where A is a 75-by-75 unit-diagonal, lower-triangular real band matrix with bandwidth 15 that is stored in an array **AB** whose dimensions are 25 by 100, and x is a real vector 75 elements long stored in an array **X** of dimension 100.

```

CHARACTER*1 UPLO,TRANS,DIAG
INTEGER*4   N,KD,LDAB,INCX
REAL*4      AB(25,100),X(100)
UPLO = 'L'
TRANS = 'N'
DIAG = 'U'
N = 75
KD = 15
LDAB = 25
INCX = 1
CALL STBMV (UPLO,TRANS,DIAG,N,KD,AB,LDAB,X,INCX)

```

Example 2 Form the **REAL*8** matrix-vector product $A^T x$, 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*4 N,KD,LDAB
REAL*4    AB(25,100),X(100)
N = 75
KD = 15
LDAB = 25
CALL DTBMV ('UPPER','TRANPOSE','NONUNIT',N,KD,AB,LDAB,X,1)

```

Name STBSV/DTBSV/CTBSV/ZTBSV
Solve Triangular Band System

Purpose Given an n -by- n upper- or lower-triangular band matrix A and an n -vector x , these subprograms overwrite x with the solution y to the system of linear equations $Ay = x$. This operation is the forward elimination or back substitution step of Gaussian elimination for band matrices. Optionally, A 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

$$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, \text{ and } x \leftarrow A^{-*}x$$

where A^{-T} is the inverse of the transpose of A , and A^{-*} is the inverse of the conjugate transpose of A .

These subprograms are more primitive than the LINPACK band equation solvers. As such, they are intended to supplement the equation solvers but not replace them, serving instead as building blocks in constructing optimized linear algebra software. In fact, many of the 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

VECLIB:

```

CHARACTER*1  uplo, trans, diag
INTEGER*4    n, kd, ldab, incx
REAL*4       ab(ldab, n), x(lenx)
CALL STBSV(uplo, trans, diag, n, kd, ab, ldab, x, incx)

```

```

CHARACTER*1  uplo, trans, diag
INTEGER*4    n, kd, ldab, incx
REAL*8       ab(ldab, n), x(lenx)
CALL DTBSV(uplo, trans, diag, n, kd, ab, ldab, x, incx)

```

```

CHARACTER*1  uplo, trans, diag
INTEGER*4    n, kd, ldab, incx
COMPLEX*8    ab(ldab, n), x(lenx)
CALL CTBSV(uplo, trans, diag, n, kd, ab, ldab, x, incx)

```

```

CHARACTER*1  uplo, trans, diag
INTEGER*4    n, kd, ldab, incx
COMPLEX*16   ab(ldab, n), x(lenx)
CALL ZTBSV(uplo, trans, diag, n, kd, ab, ldab, x, incx)

```

VECLIB8:

```

CHARACTER*1  uplo, trans, diag
INTEGER*8    n, kd, ldab, incx
REAL*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, $n \geq 0$. 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 $\text{ldab} \geq \text{kd}+1$.

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

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

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

incx < 0 x is stored backward in array x ; i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$.

Use $\text{incx} = 1$ 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 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 $\neq$ '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 Perform REAL*4 forward elimination using the 75-by-75 unit-diagonal lower-triangular real band matrix with bandwidth 15 that is stored in an array AB whose dimensions are 25 by 100, and x is a real vector 75 elements long stored in an array X of dimension 100.

```

CHARACTER*1 UPLO, TRANS, DIAG
INTEGER*4   N, KD, LDAB, INCX
REAL*4      AB(25,100), X(100)
UPLO = 'L'
TRANS = 'N'
DIAG = 'U'
N = 75
KD = 15
LDAB = 25
INCX = 1
CALL STBSV (UPLO, TRANS, DIAG, N, KD, AB, LDAB, X, INCX)

```

Example 2 Perform REAL*4 back substitution using the 75-by-75 nonunit-diagonal, upper-triangular real band matrix with bandwidth 15 that is stored in an array AB whose dimensions are 25 by 100, and x is a real vector 75 elements long stored in an array X of dimension 100.

```

INTEGER*4 N, KD, LDAB
REAL*4    AB(25,100), X(100)
N = 75
KD = 15
LDAB = 25
CALL STBSV ('UPPER', 'NONTRANS', 'NONUNIT', N, KD, AB, LDAB, X, 1)

```

Name STPMV/DTPMV/CTPMV/ZTPMV
Matrix-Vector Multiply

Purpose Given an n -by- n upper- or lower-triangular matrix A stored in packed form as described in "Matrix Storage" and an n -vector x , these subprograms compute the matrix-vector products Ax , $A^T x$, and $A^* x$, where A^T is the transpose of A , and A^* is the conjugate transpose of A . Specifically, these subprograms compute matrix-vector products of the forms

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

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**VECLIB:**

```

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

```

```

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

```

```

CHARACTER*1  uplo, trans, diag
INTEGER*4    n, incx
COMPLEX*8    ap(lenap), x(lenx)
CALL CTPMV(uplo, trans, diag, n, ap, x, incx)

```

```

CHARACTER*1  uplo, trans, diag
INTEGER*4    n, incx
COMPLEX*16   ap(lenap), x(lenx)
CALL ZTPMV(uplo, trans, diag, n, ap, x, incx)

```

VECLIB8:

```

CHARACTER*1  uplo, trans, diag
INTEGER*8    n, incx
REAL*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 , $n \geq 0$. If $n = 0$, the subprograms do not reference **ap** or **x**.

ap Array of length $\text{lenap} = n \times (n+1)/2$ 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 $\text{lenx} = (n-1) \times |\text{incx}| + 1$ containing the input vector x .

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

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

incx < 0 x is stored backward in array **x**; i.e., x_i is stored in $\mathbf{x}((i-n) \times \text{incx} + 1)$.

Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., if x_i is stored in $\mathbf{x}(i)$. 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, 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', 'TRANPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Form the REAL*4 matrix-vector product Ax , where A is a 9-by-9 unit-diagonal, lower-triangular real matrix stored in packed form in an array AP of dimension 55 and x is a real vector 9 elements long stored in an array X of dimension 10.

```
CHARACTER*1 UPLO,TRANS,DIAG
INTEGER*4   N, INCX
REAL*4      AP(55),X(10)
UPLO = 'L'
TRANS = 'N'
DIAG = 'U'
N = 9
INCX = 1
CALL STPMV (UPLO,TRANS,DIAG,N,AP,X,INCX)
```

Example 2 Form the REAL*8 matrix-vector product $A^T x$, 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*4 N
REAL*8    AP(55),X(10)
N = 6
CALL DTPMV ('UPPER','TRANPOSE','NONUNIT',N,AP,X,1)
```

Name STPSV/DTPSV/CTPSV/ZTPSV
Solve Triangular System

Purpose Given an n -by- n upper- or lower-triangular matrix A stored in packed form as described in "Matrix Storage" and an n -vector x , these subprograms overwrite x with the solution y to the system of linear equations $Ay = x$. This is the forward elimination or back substitution step of Gaussian elimination. Optionally, A may be replaced by A^T , the transpose of A , or by A^* , the conjugate transpose of A . Specifically, these subprograms compute

$$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, \text{ and } x \leftarrow A^{-*}x,$$

where A^{-T} is the inverse of the transpose of A , and A^{-*} is the inverse of the conjugate transpose of A .

These subprograms are more primitive than the LINPACK linear equation solvers. As such, they are intended to supplement but not replace them, serving instead as building blocks in constructing optimized linear algebra software. In fact, many of the 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

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

VECLIB:

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

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

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

```
CHARACTER*1  uplo, trans, diag
INTEGER*4    n, incx
COMPLEX*16   ap(lenap), x(lenx)
CALL ZTPSV(uplo, trans, diag, n, ap, x, incx)
```

VECLIB8:

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

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

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

incx < 0 x is stored backward in array $\mathbf{x}$; i.e., x_i is stored in $\mathbf{x}((i-\mathbf{n}) \times \mathbf{incx} + 1)$.

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

Output $\mathbf{x}$ 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, 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*4 forward elimination using a 75-by-75 unit-diagonal, lower-triangular real matrix stored in packed form in an array AP of dimension 5500, and x is a real vector 75 elements long stored in an array X of dimension 100.

```
CHARACTER*1 UPLO,TRANS,DIAG
INTEGER*4   N, INCX
REAL*4      AP(5500),X(100)
UPLO = 'L'
TRANS = 'N'
DIAG = 'U'
N = 75
INCX = 1
CALL STPSV (UPLO,TRANS,DIAG,N,AP,X,INCX)
```

Example 2 Perform REAL*4 back substitution using a 75-by-75 nonunit-diagonal, upper-triangular real matrix stored in packed form in an array AP of dimension 5500, and x is a real vector 75 elements long stored in an array X of dimension 100.

```
INTEGER*4 N
REAL*4    AP(5500),X(100)
N = 75
CALL STPSV ('UPPER','NONTRANS','NONUNIT',N,AP,X,1)
```

Name STRMM/DTRMM/CTRMM/ZTRMM
Triangular Matrix-Matrix Multiply

Purpose Given a scalar α , an m -by- n matrix B , and an upper- or lower-triangular matrix A , these subprograms compute either of the matrix-matrix products αAB or αBA . The size of A , either m by m or n by n , depends on which matrix product is requested. Optionally, A 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

$$\begin{aligned} B &\leftarrow \alpha AB, & B &\leftarrow \alpha A^T B, & B &\leftarrow \alpha A^* B, \\ B &\leftarrow \alpha BA, & B &\leftarrow \alpha B A^T, & B &\leftarrow \alpha B A^*. \end{aligned}$$

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 VECLIB:

```
CHARACTER*1  side, uplo, transa, diag
INTEGER*4    m, n, lda, ldb
REAL*4       alpha, a(lda, *), b(ldb, *)
CALL STRMM(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
```

```
CHARACTER*1  side, uplo, transa, diag
INTEGER*4    m, n, lda, ldb
REAL*8       alpha, a(lda, *), b(ldb, *)
CALL DTRMM(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
```

```
CHARACTER*1  side, uplo, transa, diag
INTEGER*4    m, n, lda, ldb
COMPLEX*8    alpha, a(lda, *), b(ldb, *)
CALL CTRMM(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
```

```
CHARACTER*1  side, uplo, transa, diag
INTEGER*4    m, n, lda, ldb
COMPLEX*16   alpha, a(lda, *), b(ldb, *)
CALL ZTRMM(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
```

VECLIB8:

```

CHARACTER*1  side, uplo, transa, diag
INTEGER*8    m, n, lda, ldb
REAL*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*, $m \geq 0$. 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 <i>A</i> , whose size is indicated by side : 'L' or 'l' <i>A</i> is <i>m</i> by <i>m</i> 'R' or 'r' <i>A</i> is <i>n</i> by <i>n</i> 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 <i>A</i> , 1).
b	Array containing the <i>m</i> -by- <i>n</i> matrix <i>B</i> . 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(<i>m</i> , 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', 'TRANPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Form the REAL*4 matrix product AB , where A is a 6-by-6 nonunit-diagonal, upper-triangular real matrix stored in an array A whose dimensions are 10 by 10 and B is a 6-by-8 real matrix stored in an array B of dimension 10 by 10. The matrix product will overwrite the input B matrix.

```

      CHARACTER*1 SIDE,UPLO,TRANSA,DIAG
      INTEGER*4   M,N,LDA,LDB
      REAL*4      ALPHA,A(10,10),B(10,10)
      SIDE = 'L'
      UPLO = 'U'
      TRANSA = 'N'
      DIAG = 'N'
      M = 6
      N = 8
      ALPHA = 1.0
      LDA = 10
      LDB = 10
      CALL STRMM (SIDE,UPLO,TRANSA,DIAG,M,N,ALPHA,A,LDA,B,LDB)

```

Example 2 Form the REAL*8 matrix product qBA^T , where q is a real scalar, B is a 6-by-8 real matrix stored in an array B of dimension 10 by 10, and A is a 8-by-8 unit-diagonal lower-triangular real matrix stored in an array A whose dimensions are 10 by 10. The matrix product will overwrite the input B matrix.

```

      INTEGER*4 M,N,LDA,LDB
      REAL*8    Q,A(10,10),B(10,10)
      M = 6
      N = 8
      LDA = 10
      LDB = 10
      CALL DTRMM ('RIGHT','LOWER','TRANS','UNIT',M,N,Q,A,LDA,B,LDB)

```


Name STRMV/DTRMV/CTRMV/ZTRMV
Matrix-Vector Multiply

Purpose Given an n -by- n upper- or lower-triangular matrix A and an n -vector x , these subprograms compute the matrix-vector products Ax , $A^T x$, and $A^* x$, where A^T is the transpose of A , and A^* is the conjugate transpose of A . Specifically, these subprograms compute matrix-vector products of the forms

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

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 VECLIB:

```
CHARACTER*1  uplo, trans, diag
INTEGER*4    n, lda, incx
REAL*4       a(lda, n), x(lenx)
CALL STRMV(uplo, trans, diag, n, a, lda, x, incx)
```

```
CHARACTER*1  uplo, trans, diag
INTEGER*4    n, lda, incx
REAL*8       a(lda, n), x(lenx)
CALL DTRMV(uplo, trans, diag, n, a, lda, x, incx)
```

```
CHARACTER*1  uplo, trans, diag
INTEGER*4    n, lda, incx
COMPLEX*8    a(lda, n), x(lenx)
CALL CTRMV(uplo, trans, diag, n, a, lda, x, incx)
```

```
CHARACTER*1  uplo, trans, diag
INTEGER*4    n, lda, incx
COMPLEX*16   a(lda, n), x(lenx)
CALL ZTRMV(uplo, trans, diag, n, a, lda, x, incx)
```

VECLIB8:

```
CHARACTER*1  uplo, trans, diag
INTEGER*8    n, lda, incx
REAL*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 $x \leftarrow A^T x$
 '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, $n \geq 0$. 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 $lenx = (n-1) \times |incx| + 1$ 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 $x((i-1) \times incx + 1)$.
incx < 0 x is stored backward in array **x**; i.e., x_i is stored in $x((i-n) \times incx + 1)$.

Use $\text{incx} = 1$ 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

$\text{uplo} \neq \text{'L' or 'l' or 'U' or 'u'}$,
 $\text{trans} \neq \text{'N' or 'n' or 'T' or 't' or 'C' or 'c'}$,
 $\text{diag} \neq \text{'N' or 'n' or 'U' or 'u'}$,
 $n < 0$,
 $\text{lدا} < \max(n, 1)$, and
 $\text{incx} = 0$.

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*4 matrix-vector product Ax , where A is a 9-by-9 unit-diagonal lower-triangular real matrix stored in an array A whose dimensions are 10 by 10, and x is a real vector 9 elements long stored in an array X of dimension 10.

```
CHARACTER*1 UPLO, TRANS, DIAG
INTEGER*4   N, LDA, INCX
REAL*4      A(10,10), X(10)
UPLO = 'L'
TRANS = 'N'
DIAG = 'U'
N = 9
LDA = 10
INCX = 1
CALL STRMV (UPLO, TRANS, DIAG, N, A, LDA, X, INCX)
```

Example 2 Form the REAL*8 matrix-vector product $A^T x$, 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*4 N,LDA
REAL*8    A(10,10),X(10)
N = 6
LDA = 10
CALL DTRMV ('UPPER','TRANSPOSE','NONUNIT',N,A,LDA,X,1)
```

Name STRSM/DTRSM/CTRSM/ZTRSM
Solve Triangular Systems

Purpose Given a scalar α , an upper- or lower-triangular matrix A and an m -by- n matrix B , these subprograms compute either of the matrix solutions $\alpha A^{-1}B$ or αBA^{-1} . The size of A , either m by m or n by n , depends on which matrix solution is requested. Optionally, A^{-1} 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

$$\begin{array}{lll} B \leftarrow \alpha A^{-1} B, & B \leftarrow \alpha A^{-T} B, & B \leftarrow \alpha A^{-*} B, \\ B \leftarrow \alpha B A^{-1}, & B \leftarrow \alpha B A^{-T}, & B \leftarrow \alpha B A^{-*}. \end{array}$$

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 VECLIB:

```

CHARACTER*1  side, uplo, transa, diag
INTEGER*4    m, n, lda, ldb
REAL*4       alpha, a(lda, *), b(ldb, *)
CALL STRSM(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)

CHARACTER*1  side, uplo, transa, diag
INTEGER*4    m, n, lda, ldb
REAL*8       alpha, a(lda, *), b(ldb, *)
CALL DTRSM(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)

CHARACTER*1  side, uplo, transa, diag
INTEGER*4    m, n, lda, ldb
COMPLEX*8    alpha, a(lda, *), b(ldb, *)
CALL CTRSM(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)

CHARACTER*1  side, uplo, transa, diag
INTEGER*4    m, n, lda, ldb
COMPLEX*16   alpha, a(lda, *), b(ldb, *)
CALL ZTRSM(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)

```

VECLIB8:

```

CHARACTER*1   side, uplo, transa, diag
INTEGER*8     m, n, lda, ldb
REAL*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, $B \leftarrow \alpha B A^{-1}$

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."

Example 1 Form the REAL^*4 matrix solution $A^{-1}B$, where A is a 6-by-6 nonunit-diagonal, upper-triangular real matrix stored in an array A whose dimensions are 10 by 10 and B is a 6-by-8 real matrix stored in an array B of dimension 10 by 10. The matrix solution will overwrite the input B matrix.

```

      CHARACTER*1  SIDE,UPLO,TRANSA,DIAG
      INTEGER*4    M,N,LDA,LDB
      REAL*4       ALPHA,A(10,10),B(10,10)
      SIDE = 'L'
      UPLO = 'U'
      TRANSA = 'N'
      DIAG = 'N'
      M = 6
      N = 8
      ALPHA = 1.0
      LDA = 10
      LDB = 10
      CALL STRSM (SIDE,UPLO,TRANSA,DIAG,M,N,ALPHA,A,LDA,B,LDB)

```

Example 2 Form the REAL^*8 matrix solution qBA^{-T} , where q is a real scalar, B is a 6-by-8 real matrix stored in an array B of dimension 10 by 10, and A is a 8-by-8 unit-diagonal lower-triangular real matrix stored in an array A whose dimensions are 10 by 10. The matrix solution will overwrite the input B matrix.

```

      INTEGER*4    M,N,LDA,LDB
      REAL*8       Q,A(10,10),B(10,10)
      M = 6
      N = 8
      LDA = 10
      LDB = 10
      CALL DTRSM ('RIGHT','LOWER','TRANS','UNIT',M,N,Q,A,LDA,B,LDB)

```


Name STRSV/DTRSV/CTRSV/ZTRSV
Solve Triangular System

Purpose Given an n -by- n upper- or lower-triangular matrix A and an n -vector x , these subprograms overwrite x with the solution y to the system of linear equations $Ay = x$. This is the forward elimination or back substitution step of Gaussian elimination. Optionally, A may be replaced by A^T , the transpose of A , or by A^* , the conjugate transpose of A . Specifically, these subprograms compute

$$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, \text{ and } x \leftarrow A^{-*}x,$$

where A^{-T} is the inverse of the transpose of A , and A^{-*} is the inverse of the conjugate transpose of A .

These subprograms are more primitive than the LINPACK linear equation solvers. As such, they are intended to supplement but not replace them, serving instead as building blocks in constructing optimized linear algebra software. In fact, many of the 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 VECLIB:

```
CHARACTER*1  uplo, trans, diag
INTEGER*4    n, lda, incx
REAL*4       a(lda, n), x(lenx)
CALL STRSV(uplo, trans, diag, n, a, lda, x, incx)
```

```
CHARACTER*1  uplo, trans, diag
INTEGER*4    n, lda, incx
REAL*8       a(lda, n), x(lenx)
CALL DTRSV(uplo, trans, diag, n, a, lda, x, incx)
```

```
CHARACTER*1  uplo, trans, diag
INTEGER*4    n, lda, incx
COMPLEX*8    a(lda, n), x(lenx)
CALL CTRSV(uplo, trans, diag, n, a, lda, x, incx)
```

```
CHARACTER*1  uplo, trans, diag
INTEGER*4    n, lda, incx
COMPLEX*16   a(lda, n), x(lenx)
CALL ZTRSV(uplo, trans, diag, n, a, lda, x, incx)
```

VECLIB8:

```

CHARACTER*1  uplo, trans, diag
INTEGER*8    n, lda, incx
REAL*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 $\text{lenx} = (n-1) \times |\text{incx}| + 1$ containing the right-hand-side n -vector x .

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

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

incx < 0 x is stored backward in array x ; i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$.

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*4 forward elimination using the 75-by-75 unit-diagonal lower-triangular real matrix stored in an array A whose dimensions are 100 by 100, and x is a real vector 75 elements long stored in an array X of dimension 100.

```

CHARACTER*1  UPLO,TRANS,DIAG
INTEGER*4    N,LDA,INCX
REAL*4       A(100,100),X(100)
UPLO = 'L'
TRANS = 'N'
DIAG = 'U'
N = 75
LDA = 100
INCX = 1
CALL STRSV (UPLO,TRANS,DIAG,N,A,LDA,X,INCX)

```

Example 2 Perform REAL*4 back substitution using the 75-by-75 nonunit-diagonal, upper-triangular real matrix stored in an array A whose dimensions are 100 by 100, and x is a real vector 75 elements long stored in an array X of dimension 100.

```

INTEGER*4  N,LDA
REAL*4     A(100,100),X(100)
N = 75
LDA = 100
CALL STRSV ('UPPER','NONTRANS','NONUNIT',N,A,LDA,X,1)

```

Name XERBLA
Error Handler

Purpose This subprogram is the error handler for many of the subprograms in this chapter, as indicated in the “Notes” section in the applicable subprogram descriptions. As supplied in VECLIB, XERBLA writes the following error message onto the standard error file:

```
*****
* XERBLA: subprogram name called with invalid value of argument number iarg *
*****
```

where *name* is the name of the subprogram in which the error was detected, and *iarg* is the argument number of the offending argument. For example, in SGEMV, *trans* is argument number 1 and *m* is argument number 2. If the main program is in Fortran and is executed under SPP-UX, a call traceback is also written onto the standard error file (XERBLA does not write a call traceback when used on HP-UX systems). XERBLA then terminates execution with a nonzero exit status.

You may supply a version of XERBLA that alters this action. Be aware that other subprograms, including many in 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 VECLIB:

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

VECLIB8:

```
CHARACTER*6   name
INTEGER*8     iarg
CALL XERBLA(name, iarg)
```

Input *name* The name of the subprogram in which the error was detected.

iarg The number of the argument that was found to be in error.

Notes This subprogram conforms to specifications of the Level 2 and 3 BLAS and LAPACK.

4 Linear Equations

Overview

This chapter describes the LINPACK software library included with VECLIB. The most important subprograms in this library have been upgraded by incorporating the Level 2 and Level 3 BLAS and other algorithmic changes.

The LAPACK software library included with VECLIB is a comprehensive collection of linear equation solvers and subprograms for other linear algebra computations. This software is documented in the *HP MLIB 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 VECLIB 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

The following document provides supplemental material for this chapter:

Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart. *LINPACK Users' Guide*. Philadelphia: Society for Industrial and Applied Mathematics, 1979.

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
D	Double Precision REAL
C	Single Precision COMPLEX
Z	Double 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). ZGEFA calculates the factorization (FA) of a general dense matrix (GE) using the double precision COMPLEX data type (Z).

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	D	C	Z	GE	CO	FA	SL	DI	
S	D	C	Z	GB	CO	FA	SL	DI	
S	D	C	Z	PO	CO	FA	SL	DI	
S	D	C	Z	PB	CO	FA	SL	DI	
S	D	C	Z	PP	CO	FA	SL	DI	
S	D	C	Z	SI	CO	FA	SL	DI	
S	D	C	Z	SP	CO	FA	SL	DI	
			C	Z	HI	CO	FA	SL	DI
			C	Z	HP	CO	FA	SL	DI
S	D	C	Z	TR	CO		SL	DI	
S	D	C	Z	GT			SL		
S	D	C	Z	PT			SL		
S	D	C	Z	CH	DC		UD	DD	EX
S	D	C	Z	QR	DC	SL			
S	D	C	Z	SV	DC				

LINPACK is organized such 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 $\| \cdot \|$ 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," VECLIB 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.

LINPACK Subprograms Not in This Guide

Although VECLIB includes all LINPACK subprograms, only those subprograms optimized for use on supported computers are described in this chapter. Appendix B lists the LINPACK subprograms that are included in VECLIB but are not documented in the *HP MLIB VECLIB Users' Guide*. You may find information for these subprograms in the *LINPACK Users' Guide*.

Subprograms Included in This Chapter

Following are the LINPACK subprograms optimized for use on supported computers.

Name SGBCO/DGBCO/CGBCO/ZGBCO
Estimate Condition

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, SGTSLV, DGTSL, DGTSLV, CGTSL, CGTSLV, ZGTSL, or ZGTSLV. VECLIB 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 VECLIB subprograms whose names depend on the data type:

Data Type	Estimate Condition	Factor	Solve	Determinant
REAL*4	SGBCO	SGBFA	SGBSL	SGBDI
REAL*8	DGBCO	DGBFA	DGBSL	DGBDI
COMPLEX*8	CGBCO	CGBFA	CGBSL	CGBDI
COMPLEX*16	ZGBCO	ZGBFA	ZGBSL	ZGBDI

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, which 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 $\text{ab}(kl+ku+1+i-j, j)$. 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**VECLIB:**

```

INTEGER*4      ldab, n, kl, ku, ipvt(n)
REAL*4         ab(ldab, n), rcond, work(n)
CALL SGBCO(ab, ldab, n, kl, ku, ipvt, rcond, work)

```

```

INTEGER*4      ldab, n, kl, ku, ipvt(n)
REAL*8         ab(ldab, n), rcond, work(n)
CALL DGBCO(ab, ldab, n, kl, ku, ipvt, rcond, work)

```

```

INTEGER*4      ldab, n, kl, ku, ipvt(n)
COMPLEX*8      ab(ldab, n), work(n)
REAL*4         rcond
CALL CGBCO(ab, ldab, n, kl, ku, ipvt, rcond, work)

```

```

INTEGER*4      ldab, n, kl, ku, ipvt(n)
COMPLEX*16     ab(ldab, n), work(n)
REAL*8         rcond
CALL ZGBCO(ab, ldab, n, kl, ku, ipvt, rcond, work)

```

VECLIB8:

```

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 that 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 such that the logical expression

$$1.0 + \mathbf{rcond} \text{ .EQ. } 1.0$$
 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*4 LDAB,N,KL,KU,IPVT(10)
REAL*8    AB(8,10),RCOND,WORK(10)
LDAB = 8
N = 9
KL = 2
KU = 3
CALL DGBCO (AB,LDAB,N,KL,KU,IPVT,RCOND,WORK)
IF ( 1.0D0 + RCOND .EQ. 1.0D0 ) THEN
    handle singular matrix
END IF
```

Name SGBDI/DGBDI/CGBDI/ZGBDI
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*4	SGBCO	SGBFA	SGBDI
REAL*8	DGBCO	DGBFA	DGBDI
COMPLEX*8	CGBCO	CGBFA	CGBDI
COMPLEX*16	ZGBCO	ZGBFA	ZGBDI

The companion subprograms are documented elsewhere in this chapter.

Usage

VECLIB:

```

INTEGER*4      ldab, n, kl, ku, ipvt(n)
REAL*4         ab(ldab, n), det(2)
CALL SGBDI(ab, ldab, n, kl, ku, ipvt, det)

```

```

INTEGER*4      ldab, n, kl, ku, ipvt(n)
REAL*8         ab(ldab, n), det(2)
CALL DGBDI(ab, ldab, n, kl, ku, ipvt, det)

```

```

INTEGER*4      ldab, n, kl, ku, ipvt(n)
COMPLEX*8      ab(ldab, n), det(2)
CALL CGBDI(ab, ldab, n, kl, ku, ipvt, det)

```

```

INTEGER*4      ldab, n, kl, ku, ipvt(n)
COMPLEX*16     ab(ldab, n), det(2)
CALL ZGBDI(ab, ldab, n, kl, ku, ipvt, det)

```

VECLIB8:

```

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 $ldab \geq 2kl + ku + 1$.

n The order of matrix A , $n \geq 0$.

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 $\det(A) = \det(1) \times 10^{\det(2)}$. This expression may underflow or overflow if evaluated; on HP systems, underflows automatically flush to zero, but overflows normally terminate execution. For REAL*4 and COMPLEX*8, overflow cannot occur if $\det(2) \leq 37$. 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 $\det(A) = \det(1) * 10.0 ** \text{INT}(\det(2))$ The value stored in $\det(2)$ is an integer in REAL or COMPLEX form. $\det(1)$ is normalized such that either $\det(1) = 0$ or $1 \leq Re(\det(1)) + Im(\det(1)) < 10$, 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," VECLIB 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*4 LDAB,N,KL,KU,IPVT(10),IER
REAL*8    AB(8,10),DET(2),DETA
LDAB = 8
N = 9
KL = 2
KU = 3
CALL DGBFA (AB,LDAB,N,KL,KU,IPVT,IER)
IF ( IER .EQ. 0 ) THEN
  CALL DGBDI (AB,LDAB,N,KL,KU,IPVT,DET)
  IF ( DET(1) .EQ. 0.0D0 ) THEN
    DETA = 0.0D0
  ELSE IF ( DET(2) .LE. 306 ) THEN
    DETA = DET(1) * 10.0D0 ** INT(DET(2))
  ELSE
    the determinant of A is too large to evaluate
    without overflow
  END IF
ELSE
  DETA = 0.0D0
END IF
```

Name SGBFA/DGBFA/CGBFA/ZGBFA
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, SGTSLV, DGTSL, DGTSLV, CGTSL, CGTSLV, ZGTSL, or ZGTSLV. VECLIB 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 VECLIB subprograms whose names depend on the data type:

Data Type	Factor	Estimate Condition	Solve	Determinant
REAL*4	SGBFA	SGBCO	SGBSL	SGBDI
REAL*8	DGBFA	DGBCO	DGBSL	DGBDI
COMPLEX*8	CGBFA	CGBCO	CGBSL	CGBDI
COMPLEX*16	ZGBFA	ZGBCO	ZGBSL	ZGBDI

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 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, which is done by presenting the original matrix to the subprogram in an array large enough to satisfy the additional storage requirements. Thus, for the previous 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 $\text{ab}(kl+ku+1+i-j,j)$. 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

VECLIB:

```

INTEGER*4      ldab, n, kl, ku, ipvt(n), ier
REAL*4         ab(ldab, n)
CALL SGBFA(ab, ldab, n, kl, ku, ipvt, ier)

INTEGER*4      ldab, n, kl, ku, ipvt(n), ier
REAL*8         ab(ldab, n)
CALL DGBFA(ab, ldab, n, kl, ku, ipvt, ier)

INTEGER*4      ldab, n, kl, ku, ipvt(n), ier
COMPLEX*8     ab(ldab, n)
CALL CGBFA(ab, ldab, n, kl, ku, ipvt, ier)

INTEGER*4      ldab, n, kl, ku, ipvt(n), ier
COMPLEX*16    ab(ldab, n)
CALL ZGBFA(ab, ldab, n, kl, ku, ipvt, ier)

```

VECLIB8:

```

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)

```


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 $ldab \geq 2kl+ku+1$.
	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, $0 \leq ku < n$. These subprograms are more efficient if $kl \leq ku$. 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 = $k \neq 0$	If $u_{kk} = 0$. (u_{kk} is the k -th element on the diagonal of upper triangular matrix U). Technically, this is not an error condition for these subprograms, but it does indicate that A is computationally singular and that a division by zero 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 previously, in array *AB* whose dimensions are 8 by 10.

```
INTEGER*4 LDAB,N,KL,KU,IPVT(10),IER
REAL*8    AB(8,10)
LDAB = 8
N = 9
KL = 2
KU = 3
CALL DGBFA (AB,LDAB,N,KL,KU,IPVT,IER)
IF ( IER .NE. 0 ) THEN
    handle singular matrix
END IF
```

Name SGBSL/DGBSL/CGBSL/ZGBSL
Solve Band Linear Equations

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 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*4	SGBCO	SGBFA	SGBSL
REAL*8	DGBCO	DGBFA	DGBSL
COMPLEX*8	CGBCO	CGBFA	CGBSL
COMPLEX*16	ZGBCO	ZGBFA	ZGBSL

The companion subprograms are documented elsewhere in this chapter.

Usage

VECLIB:

```

INTEGER*4      ldab, n, kl, ku, ipvt(n), job
REAL*4         ab(ldab, n), b(n)
CALL SGBSL(ab, ldab, n, kl, ku, ipvt, b, job)

```

```

INTEGER*4      ldab, n, kl, ku, ipvt(n), job
REAL*8         ab(ldab, n), b(n)
CALL DGBSL(ab, ldab, n, kl, ku, ipvt, b, job)

```

```

INTEGER*4      ldab, n, kl, ku, ipvt(n), job
COMPLEX*8      ab(ldab, n), b(n)
CALL CGBSL(ab, ldab, n, kl, ku, ipvt, b, job)

```

```

INTEGER*4      ldab, n, kl, ku, ipvt(n), job
COMPLEX*16     ab(ldab, n), b(n)
CALL ZGBSL(ab, ldab, n, kl, ku, ipvt, b, job)

```

VECLIB8:

```

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 $ldab \geq 2kl + ku + 1$.
n	The order of matrix A , $n \geq 0$.
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.	
	<pre> INTEGER*4 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 DGBCO (AB,LDAB,N,KL,KU,IPVT,RCOND,WORK) IF (1.0D0 + RCOND .NE. 1.0D0) THEN CALL DGBSL (AB,LDAB,N,KL,KU,IPVT,B,JOB) ELSE handle singular matrix END IF </pre>	

Name SGECO/DGECO/CGECO/ZGECO
Estimate Condition

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 they 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 VECLIB subprograms whose names depend on the data type:

Data Type	Estimate Condition	Factor	Solve	Determinant or inverse
REAL*4	SGECO	SGEFA	SGESL	SGEDI
REAL*8	DGECO	DGEFA	DGESL	DGEDI
COMPLEX*8	CGECO	CGEFA	CGESL	CGEDI
COMPLEX*16	ZGECO	ZGEFA	ZGESL	ZGEDI

The companion subprograms are documented elsewhere in this chapter.

Usage VECLIB:

```

      INTEGER*4      lda, n, ipvt(n)
      REAL*4         a(lda, n), rcond, work(n)
      CALL SGECO(a, lda, n, ipvt, rcond, work)

```

```

INTEGER*4      lda, n, ipvt(n)
REAL*8         a(lda, n), rcond, work(n)
CALL DGECO(a, lda, n, ipvt, rcond, work)

```

```

INTEGER*4      lda, n, ipvt(n)
COMPLEX*8      a(lda, n), work(n)
REAL*4         rcond
CALL CGECO(a, lda, n, ipvt, rcond, work)

```

```

INTEGER*4      lda, n, ipvt(n)
COMPLEX*16     a(lda, n), work(n)
REAL*8         rcond
CALL ZGECO(a, lda, n, ipvt, rcond, work)

```

VECLIB8:

```

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 such that the logical expression

$$1.0 + \text{rcond} \text{ .EQ. } 1.0$$

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 they are compatible with the VECLIB 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*4 LDA,N,IPVT(10)
REAL*8    A(10,10),RCOND,WORK(10)
LDA = 10
N = 6
CALL DGECO (A,LDA,N,IPVT,RCOND,WORK)
IF ( 1.0D0 + RCOND .EQ. 1.0D0 ) THEN
    handle singular matrix
END IF

```


Name SGEDI/DGEDI/CGEDI/ZGEDI
Determinant and Inverse

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*4	SGECO	SGEFA	SGEDI
REAL*8	DGECO	DGEFA	DGEDI
COMPLEX*8	CGECO	CGEFA	CGEDI
COMPLEX*16	ZGECO	ZGEFA	ZGEDI

The companion subprograms are documented elsewhere in this chapter.

Usage VECLIB:

```

INTEGER*4      lda, n, ipvt(n), job
REAL*4         a(lda, n), det(2), work(n)
CALL SGEDI(a, lda, n, ipvt, det, work, job)

```

```

INTEGER*4      lda, n, ipvt(n), job
REAL*8         a(lda, n), det(2), work(n)
CALL DGEDI(a, lda, n, ipvt, det, work, job)

```

```

INTEGER*4      lda, n, ipvt(n), job
COMPLEX*8      a(lda, n), det(2), work(n)
CALL CGEDI(a, lda, n, ipvt, det, work, job)

```

```

INTEGER*4      lda, n, ipvt(n), job
COMPLEX*16     a(lda, n), det(2), work(n)
CALL ZGEDI(a, lda, n, ipvt, det, work, job)

```

VECLIB8:

```

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 $lda \geq \max(n, 1)$.

n The order of matrix A , $n \geq 0$.

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 $\det(A) = \det(1) \times 10^{\det(2)}$. This expression may underflow or overflow if evaluated; on Hewlett-Packard systems, underflows automatically flush to zero, but overflows normally terminate execution. For REAL*4 and COMPLEX*8, overflow cannot occur if $\det(2) \leq 37$. 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 $\det(A) = \det(1) * 10.0 ** \text{INT}(\det(2))$ Refer to "Example 2." The value stored in $\det(2)$ is an integer in REAL or COMPLEX form. $\det(1)$ is normalized such that either $\det(1) = 0$ or $1 < Re(\det(1)) + Im(\det(1)) < 10$, 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 " $\det(A) \neq 0$ " to mean " A is nonsingular," VECLIB 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*4 LDA,N,IPVT(10),JOB
REAL*8    A(10,10),DET(2),RCOND,WORK(10)
LDA = 10
N = 6
JOB = 1
CALL DGECC (A,LDA,N,IPVT,RCOND,WORK)
IF ( 1.0D0 + RCOND .NE. 1.0D0 ) THEN
    CALL DGEDI (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*4 LDA,N,IPVT(10),IER,JOB
REAL*8    A(10,10),DET(2),DETA,WORK(10)
LDA = 10
N = 6
JOB = 10
CALL DGEFA (A,LDA,N,IPVT,IER)
IF ( IER .EQ. 0 ) THEN
    CALL DGEDI (A,LDA,N,IPVT,DET,WORK,JOB)
    IF ( DET(1) .EQ. 0.0D0 ) THEN
        DETA = 0.0D0
    ELSE IF ( DET(2) .LE. 306 ) THEN
        DETA = DET(1) * 10.0D0 ** INT(DET(2))
    ELSE
        the determinant of A is too large to evaluate
        without overflow
    END IF
ELSE
    DETA = 0.0D0
END IF

```

Name SGEFA/DGEFA/CGEFA/ZGEFA
LU Factorization

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 VECLIB subprograms whose names depend on the data type:

Data Type	Factor	Estimate Condition	Solve	Determinant or inverse
REAL*4	SGEFA	SGECO	SGESL	SGEDI
REAL*8	DGEFA	DGECO	DGESL	DGEDI
COMPLEX*8	CGEFA	CGECO	CGESL	CGEDI
COMPLEX*16	ZGEFA	ZGECO	ZGESL	ZGEDI

The companion subprograms are documented elsewhere in this chapter.

Usage VECLIB:

```

INTEGER*4      lda, n, ipvt(n), ier
REAL*4         a(lda, n)
CALL SGEFA(a, lda, n, ipvt, ier)

INTEGER*4      lda, n, ipvt(n), ier
REAL*8         a(lda, n)
CALL DGEFA(a, lda, n, ipvt, ier)

```

```

INTEGER*4      lda, n, ipvt(n), ier
COMPLEX*8      a(lda, n)
CALL CGEFA(a, lda, n, ipvt, ier)

```

```

INTEGER*4      lda, n, ipvt(n), ier
COMPLEX*16     a(lda, n)
CALL ZGEFA(a, lda, n, ipvt, ier)

```

VECLIB8:

```

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 $lda \geq \max(n, 1)$.

n The order of matrix A , $n \geq 0$.

Output

a The triangular factors replace the input matrix; the strict lower triangle of **a** contains the strict lower triangle of L and the upper triangle of **a** contains U . **a** must be preserved between the factorization call and any solve, determinant, or inverse call.

ipvt The pivot information necessary to construct the permutation matrix P . **ipvt** must be preserved between the factorization call and any solve, determinant, or inverse call.

ier Status response:

ier = 0 Normal return.

ier = $k \neq 0$ If $u_{kk} = 0$. (u_{kk} is the k -th element on the diagonal of upper triangular matrix U). Technically, this is not an error condition for these subprograms, but it does indicate that A is computationally singular and that a division by zero 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 VECLIB 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*4 LDA,N,IPVT(10),IER
REAL*8    A(10,10)
LDA = 10
N = 6
CALL DGEFA (A,LDA,N,IPVT,IER)
IF ( IER .NE. 0 ) THEN
    handle singular matrix
END IF
```

Name SGESL/DGESL/CGESL/ZGESL
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*4	SGECO	SGEFA	SGESL
REAL*8	DGECO	DGEFA	DGESL
COMPLEX*8	CGECO	CGEFA	CGESL
COMPLEX*16	ZGECO	ZGEFA	ZGESL

The companion subprograms are documented elsewhere in this chapter.

Usage

VECLIB:

```

INTEGER*4      lda, n, ipvt(n), job
REAL*4         a(lda, n), b(n)
CALL SGESL(a, lda, n, ipvt, b, job)

```

```

INTEGER*4      lda, n, ipvt(n), job
REAL*8         a(lda, n), b(n)
CALL DGESL(a, lda, n, ipvt, b, job)

```

```

INTEGER*4      lda, n, ipvt(n), job
COMPLEX*8      a(lda, n), b(n)
CALL CGESL(a, lda, n, ipvt, b, job)

```

```

INTEGER*4      lda, n, ipvt(n), job
COMPLEX*16     a(lda, n), b(n)
CALL ZGESL(a, lda, n, ipvt, b, job)

```

VECLIB8:

```

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 $lda \geq \max(n, 1)$.

n	The order of matrix A , $n \geq 0$.
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 $\neq$ 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 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*4 LDA,N,IPVT(10),JOB
REAL*8     A(10,10),B(10),RCOND,WORK(10)
LDA = 10
N = 6
JOB = 0
CALL DGECCO (A,LDA,N,IPVT,RCOND,WORK)
IF ( 1.0D0 + RCOND .NE. 1.0D0 ) THEN
    CALL DGESL (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*4 LDA,N,IPVT(10),IER,JOB
REAL*8    A(10,10),B(10)
LDA = 10
N = 6
JOB = 1
CALL DGEFA (A,LDA,N,IPVT,IER)
IF ( IER .EQ. 0 ) THEN
    CALL DGESL (A,LDA,N,IPVT,B,JOB)
ELSE
    handle singular 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 .

Name SGTSL/DGTSL/CGTSL/ZGTSL
Solve Tridiagonal Linear Equations

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.

These subprograms perform the same function as subprograms SGTSLV, DGTSLV, CGTSLV, and ZGTSLV, except that the latter are designed for use on tridiagonal matrices that are known to be diagonally dominant. A matrix is diagonally dominant if

$$|a_{ii}| \geq \sum_{j \neq i} |a_{ij}|, \quad i = 1, 2, \dots, n.$$

When a matrix is diagonally dominant, partial pivoting is not required to ensure numerical stability. Eliminating partial pivoting admits parallelized algorithms, so SGTSLV, DGTSLV, CGTSLV, and ZGTSLV may be faster than these subprograms. For efficiency, use SGTSLV, DGTSLV, CGTSLV, or ZGTSLV when it is known that partial pivoting may be safely omitted.

Matrix Storage The following example illustrates the storage of general tridiagonal matrices. Consider the 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**VECLIB:**

```

INTEGER*4      n, ier
REAL*4         dl(n), d(n), du(n), b(n)
CALL SGTSL(n, dl, d, du, b, ier)

```

```

INTEGER*4      n, ier
REAL*8         dl(n), d(n), du(n), b(n)
CALL DGTSL(n, dl, d, du, b, ier)

```

```

INTEGER*4      n, ier
COMPLEX*8      dl(n), d(n), du(n), b(n)
CALL CGTSL(n, dl, d, du, b, ier)

```

```

INTEGER*4      n, ier
COMPLEX*16     dl(n), d(n), du(n), b(n)
CALL ZGTSL(n, dl, d, du, b, ier)

```

VECLIB8:

```

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, $dl(i) = a_{i,i-1}$, $i = 2, 3, \dots, n$. On return, **dl** is destroyed, including **dl(1)**.

	d	Array containing the principal diagonal of the tridiagonal matrix, $d(i) = a_{ii}$, $i = 1, 2, \dots, n$. On return, d is destroyed.
	du	Array containing the superdiagonal of the tridiagonal matrix, $du(i) = a_{i,i+1}$, $i = 1, 2, \dots, n-1$. On return, du is destroyed, including du(n) .
	b	The right-hand-side vector b .
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*4 N,IER
REAL*8    DL(10),D(10),DU(10),B(10)
N = 7
CALL DGTSL (N,DL,D,DU,B,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
END IF

```

Name SGTSLV/DGTSLV/CGTSLV/ZGTSLV
Solve Tridiagonal Linear Equations

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.

These subprograms perform the same function as the LINPACK subprograms SGTSL, DGTSL, CGTSL, and ZGTSL, except that they are designed for use on tridiagonal matrices that are known to be diagonally dominant. A matrix is diagonally dominant if

$$|a_{ii}| \geq \sum_{j \neq i} |a_{ij}|, i = 1, 2, \dots, n.$$

When a matrix is diagonally dominant, partial pivoting is not required to ensure numerical stability. Eliminating partial pivoting admits vectorized and parallelized algorithms, so these subprograms are faster than their LINPACK counterparts. Use SGTSL, DGTSL, CGTSL, or ZGTSL unless it is known that partial pivoting may be safely omitted.

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

VECLIB:

```

INTEGER*4      n, ier
REAL*4         dl(n), d(n), du(n), b(n)
CALL SGTSLV(n, dl, d, du, b, ier)

```

```

INTEGER*4      n, ier
REAL*8         dl(n), d(n), du(n), b(n)
CALL DGTSLV(n, dl, d, du, b, ier)

```

```

INTEGER*4      n, ier
COMPLEX*8      dl(n), d(n), du(n), b(n)
CALL CGTSLV(n, dl, d, du, b, ier)

```

```

INTEGER*4      n, ier
COMPLEX*16     dl(n), d(n), du(n), b(n)
CALL ZGTSLV(n, dl, d, du, b, ier)

```

VECLIB8:

```

INTEGER*8      n, ier
REAL*8         dl(n), d(n), du(n), b(n)
CALL SGTSLV(n, dl, d, du, b, ier)

```

```

INTEGER*8      n, ier
COMPLEX*16     dl(n), d(n), du(n), b(n)
CALL CGTSLV(n, dl, d, du, b, ier)

```

Input

n The order of matrix A , $n > 0$.

dl Array containing the subdiagonal of the tridiagonal matrix, $dl(i) = a_{i,i-1}$, $i = 2, 3, \dots, n$. On return, **dl** is destroyed, including **dl**(1).

	d	Array containing the principal diagonal of the tridiagonal matrix, $\mathbf{d}(i) = a_{ii}, i = 1, 2, \dots, n$. On return, d is destroyed.
	du	Array containing the superdiagonal of the tridiagonal matrix, $\mathbf{du}(i) = a_{i,i+1}, i = 1, 2, \dots, n-1$. On return, du is destroyed, including du(n) .
	b	The right-hand-side vector b .
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 = -1	$n \leq 0$.
	ier = -2	A divide-by-zero was encountered; not available in all implementations.

Notes These subprograms are usage compatible with the standard LINPACK subprograms respectively named SGTSL, DGTSL, CGTSL, and ZGTSL, with three exceptions: they have different names, the **ier** status responses are different, and they do not employ partial pivoting to ensure numerical stability. You should prefer these subprograms over the corresponding LINPACK subprograms only when you know that pivoting is unnecessary.

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*4 N, IER
      REAL*8    DL(10), D(10), DU(10), B(10)
      N = 7
      CALL DGTSLV (N, DL, D, DU, B, IER)
      IF ( IER .NE. 0 ) THEN
         handle error condition
      END IF

```

Name SPBCO/DPBCO/CPBCO/ZPBCO
Estimate Condition

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, DPTSL, CPTSL, and ZPTSL.

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 VECLIB subprograms whose names depend on the data type:

Data Type	Estimate Condition	Factor	Solve	Determinant
REAL*4	SPBCO	SPBFA	SPBSL	SPBDI
REAL*8	DPBCO	DPBFA	DPBSL	DPBDI
COMPLEX*8	CPBCO	CPBFA	CPBSL	CPBDI
COMPLEX*16	ZPBCO	ZPBFA	ZPBSL	ZPBDI

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 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

VECLIB:

```

    INTEGER*4      ldab, n, kd, ier
    REAL*4         ab(ldab, n), rcond, work(n)
    CALL SPBCO(ab, ldab, n, kd, rcond, work, ier)

    INTEGER*4      ldab, n, kd, ier
    REAL*8         ab(ldab, n), rcond, work(n)
    CALL DPBCO(ab, ldab, n, kd, rcond, work, ier)

    INTEGER*4      ldab, n, kd, ier
    COMPLEX*8      ab(ldab, n), work(n)
    REAL*4         rcond
    CALL CPBCO(ab, ldab, n, kd, rcond, work, ier)

    INTEGER*4      ldab, n, kd, ier
    COMPLEX*16     ab(ldab, n), work(n)
    REAL*8         rcond
    CALL ZPBCO(ab, ldab, n, kd, rcond, work, ier)

```

VECLIB8:

```

    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 $0 \leq j-i \leq kd$, 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** $\geq$ **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, $0 \leq kd < n$.

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 $1.0 + \text{rcond} \text{ .EQ. } 1.0$ is true, A can be regarded as singular to working precision.
	ier	Status response: ier = 0 ier = $k \neq 0$ Normal return—factorization complete. 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*4 LDAB,N,KD,IER
REAL*8    AB(5,10),RCOND,WORK(10)
LDAB = 5
N = 7
M = 2
CALL DPBCO (AB,LDAB,N,KD,RCOND,WORK,IER)
IF ( IER .NE. 0 ) THEN
    handle indefinite matrix
ELSE IF ( 1.0D0 + RCOND .EQ. 1.0D0 ) THEN
    handle singular matrix
END IF

```

Name SPBDI/DPBDI/CPBDI/ZPBDI
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*4	SPBCO	SPBFA	SPBDI
REAL*8	DPBCO	DPBFA	DPBDI
COMPLEX*8	CPBCO	CPBFA	CPBDI
COMPLEX*16	ZPBCO	ZPBFA	ZPBDI

The companion subprograms are documented elsewhere in this chapter.

Usage VECLIB:

```

      INTEGER*4      ldab, n, kd
      REAL*4         ab(ldab, n), det(2)
      CALL SPBDI(ab, ldab, n, kd, det)

```

```

INTEGER*4      ldab, n, kd
REAL*8         ab(ldab, n), det(2)
CALL DPBDI(ab, ldab, n, kd, det)

```

```

INTEGER*4      ldab, n, kd
COMPLEX*8      ab(ldab, n)
REAL*4         det(2)
CALL CPBDI(ab, ldab, n, kd, det)

```

```

INTEGER*4      ldab, n, kd
COMPLEX*16     ab(ldab, n)
REAL*8         det(2)
CALL ZPBDI(ab, ldab, n, kd, det)

```

VECLIB8:

```

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 $\text{ldab} \geq n+1$.
	n	The order of matrix A , $n \geq 0$.
	kd	The half bandwidth of A , i.e., the number of diagonals above the principal diagonal in the band, $0 \leq \text{kd} < n$.
Output	det	The determinant of A , in the form $\det(A) = \det(1) \times 10^{\det(2)}$. This expression may underflow or overflow if evaluated; on HP systems, underflows automatically flush to zero, but overflows normally terminate execution. For REAL*4 and COMPLEX*8, overflow cannot occur if $\det(2) \leq 37$. For REAL*8 and COMPLEX*16, overflow cannot occur

if $\text{det}(2) \leq 306$. If evaluation is safe, an efficient way to do it is with the statement

$$\text{det}(A) = \text{det}(1) * 10.0 ** \text{INT}(\text{det}(2))$$

The value stored in $\text{det}(2)$ is an integer in REAL form.

$\text{det}(1)$ is normalized such that $\text{det}(1) = 0$ or

$1 \leq \text{det}(1) < 10$.

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 " $\text{det}(A) \neq 0$ " to mean "A is nonsingular," VECLIB 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*4 LDAB,N,KD,IER
REAL*8    AB(5,10),DET(2),DETA
LDAB = 5
N = 7
KD = 2
CALL DPBFA (AB,LDAB,N,KD,IER)
IF ( IER .EQ. 0 ) THEN
    CALL DPBDI (AB,LDAB,N,KD,DET)
    IF ( DET(1) .EQ. 0.0D0 ) THEN
        DETA = 0.0D0
    ELSE IF ( DET(2) .LE. 306 ) THEN
        DETA = DET(1) * 10.0D0 ** INT(DET(2))
    ELSE
        the determinant of A is too large to evaluate
        without overflow
    END IF
ELSE
    DETA = 0.0D0
END IF

```


Name SPBFA/DPBFA/CPBFA/ZPBFA
Cholesky Factorization

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, DPTSL, CPTSL, and ZPTSL.

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 VECLIB subprograms whose names depend on the data type:

Data Type	Factor	Estimate Condition	Solve	Determinant
REAL*4	SPBFA	SPBCO	SPBSL	SPBDI
REAL*8	DPBFA	DPBCO	DPBSL	DPBDI
COMPLEX*8	CPBFA	CPBCO	CPBSL	CPBDI
COMPLEX*16	ZPBFA	ZPBCO	ZPBSL	ZPBDI

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

VECLIB:

```

INTEGER*4      ldab, n, kd, ier
REAL*4         ab(ldab, n)
CALL SPBFA(ab, ldab, n, kd, ier)

INTEGER*4      ldab, n, kd, ier
REAL*8         ab(ldab, n)
CALL DPBFA(ab, ldab, n, kd, ier)

```

```

INTEGER*4      ldab, n, kd, ier
COMPLEX*8      ab(ldab, n)
CALL CPBFA(ab, ldab, n, kd, ier)

```

```

INTEGER*4      ldab, n, kd, ier
COMPLEX*16     ab(ldab, n)
CALL ZPBFA(ab, ldab, n, kd, ier)

```

VECLIB8:

```

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 $0 \leq j-i \leq kd$, then a_{ij} is stored in $ab(kd+1+i-j, j)$. 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 $ldab \geq kd+1$.
	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, $0 \leq kd < n$.
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 = $k \neq 0$	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 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*4 LDAB,N,KD,IER
REAL*8    AB(5,10)
LDAB = 5
N = 7
KD = 2
CALL DPBFA (AB,LDAB,N,KD,IER)
IF ( IER .NE. 0 ) THEN
    handle singular or indefinite matrix
END IF
```

Name SPBSL/DPBSL/CPBSL/ZPBSL
Solve Linear Equations

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*4	SPBCO	SPBFA	SPBSL
REAL*8	DPBCO	DPBFA	DPBSL
COMPLEX*8	CPBCO	CPBFA	CPBSL
COMPLEX*16	ZPBCO	ZPBFA	ZPBSL

The companion subprograms are documented elsewhere in this chapter.

Usage VECLIB:

```
INTEGER*4      ldab, n, kd
REAL*4         ab(ldab, n), b(n)
CALL SPBSL(ab, ldab, n, kd, b)
```

```
INTEGER*4      ldab, n, kd
REAL*8         ab(ldab, n), b(n)
CALL DPBSL(ab, ldab, n, kd, b)
```

```
INTEGER*4      ldab, n, kd
COMPLEX*8      ab(ldab, n), b(n)
CALL CPBSL(ab, ldab, n, kd, b)
```

```

      INTEGER*4      ldab, n, kd
      COMPLEX*16     ab(ldab, n), b(n)
      CALL ZPBSL(ab, ldab, n, kd, b)

```

VECLIB8:

```

      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 $\text{ldab} \geq \text{kd} + 1$.
	n	The order of matrix A , $n \geq 0$.
	kd	The half bandwidth of A , i.e., the number of diagonals above the principal diagonal in the band, $0 \leq \text{kd} < n$.
	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.	

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*4 LDAB,N,KD,IER
      REAL*8    AB(5,10),B(10),RCOND,WORK(10)
      LDAB = 5
      N = 7
      KD = 2
      CALL DPBCO (AB,LDAB,N,KD,RCOND,WORK,IER)
      IF ( IER .NE. 0 ) THEN
         handle indefinite matrix
      ELSE IF ( 1.0D0 + RCOND .NE. 1.0D0 ) THEN
         CALL DPBSL (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.

Name SPOCO/DPOCO/CPOCO/ZPOCO
Estimate Condition

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 VECLIB subprograms whose names depend on the data type:

Data Type	Estimate Condition	Factor	Solve	Determinant or inverse
REAL*4	SPOCO	SPOFA	SPOSL	SPODI
REAL*8	DPOCO	DPOFA	DPOSL	DPODI
COMPLEX*8	CPOCO	CPOFA	CPOSL	CPODI
COMPLEX*16	ZPOCO	ZPOFA	ZPOSL	ZPODI

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

VECLIB:

```
INTEGER*4      lda, n, ier
REAL*4         a(lda, n), rcond, work(n)
CALL SPOCO(a, lda, n, rcond, work, ier)
```

```
INTEGER*4      lda, n, ier
REAL*8         a(lda, n), rcond, work(n)
CALL DPOCO(a, lda, n, rcond, work, ier)
```

```
INTEGER*4      lda, n, ier
COMPLEX*8      a(lda, n), work(n)
REAL*4         rcond
CALL CPOCO(a, lda, n, rcond, work, ier)
```

```
INTEGER*4      lda, n, ier
COMPLEX*16     a(lda, n), work(n)
REAL*8         rcond
CALL ZPOCO(a, lda, n, rcond, work, ier)
```

VECLIB8:

```
INTEGER*8      lda, n, ier
REAL*8         a(lda, n), rcond, work(n)
CALL SPOCO(a, lda, n, rcond, work, ier)
```

```
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 $lda \geq \max(n, 1)$.

n The order of matrix A , $n \geq 0$.

Working storage	work	An array of size n , used for work space.
Output	a	The Cholesky factor <i>R</i> replaces the input matrix <i>A</i> 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 $1.0 + \text{rcond} \text{ .EQ. } 1.0$ is true, then <i>A</i> can be regarded as singular to working precision.
	ier	Status response: ier = 0 Normal return—factorization complete. ier = <i>k</i> ≠ 0 The leading submatrix of order <i>k</i> 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*4 LDA,N,IER
REAL*8    A(10,10),RCOND,WORK(10)
LDA = 10
N = 6
CALL DPOCO (A,LDA,N,RCOND,WORK,IER)
IF ( IER .NE. 0 ) THEN
    handle indefinite matrix
ELSE IF ( 1.0D0 + RCOND .EQ. 1.0D0 ) THEN
    handle singular matrix
END IF

```

Name SPODI/DPODI/CPODI/ZPODI
Determinant and Inverse

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*4	SPOCO	SPOFA	SPODI
REAL*8	DPOCO	DPOFA	DPODI
COMPLEX*8	CPOCO	CPOFA	CPODI
COMPLEX*16	ZPOCO	ZPOFA	ZPODI

The companion subprograms are documented elsewhere in this chapter.

Usage VECLIB:

```

INTEGER*4      lda, n, job
REAL*4         a(lda, n), det(2)
CALL SPODI(a, lda, n, det, job)

INTEGER*4      lda, n, job
REAL*8         a(lda, n), det(2)
CALL DPODI(a, lda, n, det, job)

```

```

INTEGER*4      lda, n, job
COMPLEX*8      a(lda, n)
REAL*4         det(2)
CALL CPODI(a, lda, n, det, job)

```

```

INTEGER*4      lda, n, job
COMPLEX*16     a(lda, n)
REAL*8         det(2)
CALL ZPODI(a, lda, n, det, job)

```

VECLIB8:

```

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 $lda \geq \max(n, 1)$.

n The order of matrix A , $n \geq 0$.

job Option flag:

job = 1	Computer 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

$\det(A) = \det(1) \times 10^{\det(2)}$. This expression may underflow or overflow if evaluated; on HP systems, underflows automatically flush to zero, but overflows normally terminate execution. For REAL*4 and COMPLEX*8, overflow cannot occur if $\det(2) \leq 37$. 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

$$\det(A) = \det(1) * 10.0 ** \text{INT}(\det(2))$$

Refer to "Example 2."

The value stored in $\det(2)$ is an integer in REAL form.

$\det(1)$ is normalized such 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 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," VECLIB 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*4 LDA,N,IER,JOB
REAL*8    A(10,10),DET(2),RCOND,WORK(10)
LDA = 10
N = 6
JOB = 1
CALL DPOCO (A,LDA,N,RCOND,WORK,IER)
IF ( IER .NE. 0 ) THEN
    handle indefinite matrix
ELSE IF ( 1.0D0 + RCOND .NE. 1.0D0 ) THEN
    CALL DPODI (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*4 LDA,N,IER,JOB
REAL*8    A(10,10),DET(2),DETA
LDA = 10
N = 6
JOB = 10
CALL DPOFA (A,LDA,N,IER)
IF ( IER .EQ. 0 ) THEN
    CALL DPODI (A,LDA,N,DET,JOB)
    IF ( DET(1) .EQ. 0.0D0 ) THEN
        DETA = 0.0D0
    ELSE IF ( DET(2) .LE. 306 ) THEN
        DETA = DET(1) * 10.0D0 ** INT(DET(2))
    ELSE
        the determinant of A is too large to evaluate
        without overflow
    END IF
ELSE
    DETA = 0.0D0
END IF

```

Name SPOFA/DPOFA/CPOFA/ZPOFA
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 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 VECLIB subprograms whose names depend on the data type:

Data Type	Factor	Estimate Condition	Solve	Determinant or inverse
REAL*4	SPOFA	SPOCO	SPOSL	SPODI
REAL*8	DPOFA	DPOCO	DPOSL	DPODI
COMPLEX*8	CPOFA	CPOCO	CPOSL	CPODI
COMPLEX*16	ZPOFA	ZPOCO	ZPOSL	ZPODI

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

VECLIB:

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

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

```
INTEGER*4      lda, n, ier
COMPLEX*8      a(lda, n)
CALL CPOFA(a, lda, n, ier)
```

```
INTEGER*4      lda, n, ier
COMPLEX*16     a(lda, n)
CALL ZPOFA(a, lda, n, ier)
```

VECLIB8:

```
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 $lda \geq \max(n, 1)$.

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*4 LDA,N,IER
REAL*8    A(10,10)
LDA = 10
N = 6
CALL DPOFA (A,LDA,N,IER)
IF ( IER .NE. 0 ) THEN
    handle indefinite matrix
END IF
```

Name SPOSL/DPOSL/CPOSL/ZPOSL
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*4	SPOCO	SPOFA	SPOSL
REAL*8	DPOCO	DPOFA	DPOSL
COMPLEX*8	CPOCO	CPOFA	CPOSL
COMPLEX*16	ZPOCO	ZPOFA	ZPOSL

The companion subprograms are documented elsewhere in this chapter.

Usage VECLIB:

```

      INTEGER*4      lda, n
      REAL*4         a(lda, n), b(n)
      CALL SPOSL(a, lda, n, b)

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

```

```

INTEGER*4      lda, n
COMPLEX*8      a(lda, n), b(n)
CALL CPOS(L(a, lda, n, b)

```

```

INTEGER*4      lda, n
COMPLEX*16     a(lda, n), b(n)
CALL ZPOS(L(a, lda, n, b)

```

VECLIB8:

```

INTEGER*8      lda, n
REAL*8         a(lda, n), b(n)
CALL SPOS(L(a, lda, n, b)

```

```

INTEGER*8      lda, n
COMPLEX*16     a(lda, n), b(n)
CALL CPOS(L(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 $lda \geq \max(n, 1)$.
	n	The order of matrix A , $n \geq 0$.
	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*4 LDA,N,IER
REAL*8    A(10,10),B(10),RCOND,WORK(10)
LDA = 10
N = 6
CALL DPOCO (A,LDA,N,RCOND,WORK,IER)
IF ( IER .NE. 0 ) THEN
    handle indefinite matrix
ELSE IF ( 1.0D0 + RCOND .NE. 1.0D0 ) THEN
    CALL DPOS(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 .

Name SPTSL/.../ZPTSL

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

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

VECLIB:

```

INTEGER*4      n
REAL*4         d(n), e(n-1), b(n)
CALL SPTSL(n, d, e, b)

```

```

INTEGER*4      n
REAL*8         d(n), e(n-1), b(n)
CALL DPTSL(n, d, e, b)

```

```

INTEGER*4      n
COMPLEX*8      d(n), e(n-1), b(n)
CALL CPTSL(n, d, e, b)

```

```

INTEGER*4      n
COMPLEX*16     d(n), e(n-1), b(n)
CALL ZPTSL(n, d, e, b)

```

VECLIB8:

```

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, $d(i) = a_{ii}$, $i = 1, 2, \dots, n$. 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, $e(i) = a_{i,i+1}$, $i = 1, 2, \dots, n-1$.

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*4 N
REAL*8    D(10),E(10),B(10)
N = 7
CALL DPTSL (N,D,E,B)
```


5 Eigenvalues and Eigenvectors

Overview

This chapter describes the EISPACK library included with VECLIB.

Some subprograms in this library have been upgraded by incorporating Level 2 and Level 3 BLAS and other algorithmic improvements.

The LAPACK software library included with VECLIB is a comprehensive collection of eigenvalue and eigenvector solvers and subprograms for other linear algebra computations. This software is documented in the *HP MLIB 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 VECLIB 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, $Ax = \lambda x$

The following documents provide supplemental material for this chapter:

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.

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 VECLIB 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. Subprograms in the two versions have the same names. To include both versions in VECLIB would require changing the names of one or both versions, which means the EISPACK subprograms included in the VECLIB library would be nonstandard. To avoid this problem, only the double-precision (64-bit) version is included in VECLIB and only the single-precision version (64-bit) is included in VECLIB8.

EISPACK Subprograms Not in This Guide

Although all EISPACK subprograms are included in VECLIB, only upgraded ones are described in this chapter. Appendix B lists the subprograms that are included in VECLIB but not documented in the *HP MLIB VECLIB User's Guide*. You may find information for these subprograms in the *EISPACK Guide* and the *EISPACK Guide Extension*.

Subprograms Included in This Chapter

Following are the EISPACK subprograms included with VECLIB.

Name	RS Eigenvalues and Eigenvectors of a Real Symmetric Matrix	
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 , $i = 1, 2, \dots, n$, 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	VECLIB: <pre> INTEGER*4 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) </pre> VECLIB8: <pre> 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) </pre>	
Input	ldax The leading dimension of arrays a and x as declared in the calling program unit, with $ldax \geq \max(n, 1)$. n The order of matrix A , $n \geq 0$. 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 $\neq$ 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 $\neq$ 0.
	w	The eigenvalues λ_i of <i>A</i> 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 <i>A</i> if ier = 0 is returned. The <i>j</i> -th column of x contains the eigenvector x_j of <i>A</i> corresponding to the eigenvalue in w (<i>j</i>), <i>j</i> = 1, 2, ..., n . Eigenvectors are normalized to have Euclidean length = 1.
	ier	Status response:
	ier = 0	Normal return.
	ier = <i>k</i> , 1 ≤ <i>k</i> ≤ n	If calculation of the <i>k</i> -th eigenvalue failed to converge. w (1), w (2), ..., w (<i>k</i> −1) are eigenvalues but are not necessarily the smallest and are not necessarily sorted. If eigenvectors are requested, the first <i>k</i> −1 columns of x are eigenvectors corresponding to the first <i>k</i> −1 elements of w .
	ier = 10 n	If n > ldax . No eigenvalues or eigenvectors are returned.

Notes This subprogram is usage-compatible with the standard double-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*4 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*4 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
```

Name TQL2
Eigenvalues and Eigenvectors of a Real Symmetric Matrix

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, $\lambda_i, i = 1, 2, \dots, n$, and nonzero vectors, $x_i, i = 1, 2, \dots, n$, 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 VECLIB:
 INTEGER*4 ldx, n, ier
 REAL*8 d(n), e(n), x(ldx, n)
 CALL TQL2(ldx, n, d, e, x, ier)

VECLIB8:

```

      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 $\text{ldx} \geq \max(\text{n}, 1)$.
	n	The order of matrix A , $\text{n} \geq 0$.
	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.
Output	d	Eigenvalues, λ_i , $i = 1, 2, \dots, n$, 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), $j = 1, 2, \dots, n$. Eigenvectors are normalized to have Euclidean length = 1.
	ier	Status response: ier = 0 Normal return. ier = k, $1 \leq k \leq n$ 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. If eigenvectors are requested, 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 double-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*4 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) = 0.0D0
  END DO
  X(J,J) = 1.0D0
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*4 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

```


Name TQLRAT
Eigenvalues of a Real Symmetric Matrix

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 , $i = 1, 2, \dots, n$, for which there exist corresponding nonzero vectors, x_i , $i = 1, 2, \dots, n$, 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

VECLIB:

```

      INTEGER*4      n, ier
      REAL*8         d(n), e2(n)
      CALL TQLRAT(n, d, e2, ier)

```

VECLIB8:

```

      INTEGER*8      n, ier
      REAL*8         d(n), e2(n)
      CALL TQLRAT(n, d, e2, ier)

```

Input

n The order of matrix A , $n \geq 0$.

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 , $i = 1, 2, \dots, n$, of A overwrite the input if $ier = 0$ is returned. Eigenvalues have been sorted into ascending order.

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 double-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*4 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*4 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
```

Name TRED1
Reduce Real Symmetric Matrix to Tridiagonal Form

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 VECLIB:

```

      INTEGER*4      lda, n
      REAL*8         a(lda, n), d(n), e(n), e2(n)
      CALL TRED1(lda, n, a, d, e, e2)

```

VECLIB8:

```

      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 $lda \geq \max(n, 1)$.

n The order of matrix A , $n \geq 0$.

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 double-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*4 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)

```

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*4 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

```

Name	TRED2 Reduce Real Symmetric Matrix to Tridiagonal Form	
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 $X^T A X = T.$	
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	VECLIB: <pre> INTEGER*4 ldax, n REAL*8 a(ldax, n), d(n), e(n), x(ldax, n) CALL TRED2(ldax, n, a, d, e, x) </pre> VECLIB8: <pre> INTEGER*8 ldax, n REAL*8 a(ldax, n), d(n), e(n), x(ldax, n) CALL TRED2(ldax, n, a, d, e, x) </pre>	
Input	ldax	The leading dimension of arrays a and x as declared in the calling program unit, with $ldax \geq \max(n, 1)$.
	n	The order of matrix A , $n \geq 0$.
	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 double-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*4 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)

```

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*4 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

```


6 Sparse Linear Equations

Overview

This chapter describes state-of-the-art software for the direct solution of sparse symmetric linear equations. This package of subprograms provides very efficient use of the Hewlett-Packard architecture in conjunction with powerful techniques for using the sparsity of the problem to reduce the cost of solution. Accuracy is assured through appropriate numerical techniques.

This chapter explains how to use the sparse linear equation subprograms to solve systems of sparse linear equations where the coefficient matrix is symmetric. Subprograms are provided to:

- Solve a single sparse linear system
- Estimate condition number of the coefficient matrix
- Efficiently solve multiple systems of equations

The following documents provide supplemental material for this chapter:

Ashcraft, C.C. "A Vector Implementation of the Multifrontal Method for Large Sparse, Symmetric Positive Definite Linear Systems." *Boeing Computer Services Technical Report ETA-TR-51*. 1987.

Ashcraft, C.C. and R.G. Grimes. "The Influence of Relaxed Supernode Partitions on the Multifrontal Method." *ACM Transactions on Mathematical Software*. December, 1989. Vol. 15, No. 4. pp 291-309.

Ashcraft, C.C., R.G. Grimes, J.G. Lewis, B.W. Peyton, and H.D. Simon. "Progress in Sparse Matrix Methods for Large Linear Systems on Vector Supercomputers." *The International Journal of Supercomputer Applications*. 1987. Vol. 1, No. 4. pp. 10-30.

Duff, I.S., A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Methods*. Oxford, England: Clarendon Press. 1986.

Duff, I.S. and J.K. Reid. "The Multifrontal Solution of Indefinite Sparse Symmetric Linear Equations." *ACM Transactions of Mathematical Software*. September, 1983. Vol. 9, No. 3. pp. 302-325.

George, J.A. and J.W. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Englewood Cliffs, NJ: Prentice-Hall, Inc. 1981.

Chapter Objectives

After you read this chapter you will:

- Understand what a sparse system is
- Understand how to use these subprograms to solve linear systems and/or to estimate condition numbers
- Know some of the issues in choosing an optimal method for a specific problem

This sparse matrix linear equation software is designed so that it is possible to call a single subprogram to solve a single system of sparse symmetric linear equations. However, this requires a particular format for the sparse matrix. This package provides other approaches that provide a general interface to alternative representations of the sparse matrix and also make available underlying capabilities for reducing the cost of solving multiple sparse systems. These optional approaches, however, require the user to call a sequence of subprograms. This is similar to, but significantly more elaborate than, the use of LINPACK as described in Chapter 4.

What You Need to Know to Use These Subprograms

Sparsity and Storage Formats

Sparse matrices are matrices in which most of the entries are zero. The goal of sparse matrix software is to take maximum advantage of these zero entries to reduce storage and arithmetic. Storage is reduced by not storing zero entries; arithmetic is reduced by not performing operations on entries that are known to be zero.

It is easiest to see how to economize on storage. Suppose that an n -by- n matrix A has only nz nonzero entries. Then A could be specified completely by storing each of the nonzero values in an array of length nz that was accompanied by two integer arrays of length nz holding the corresponding row and column indices. Thus, $3nz$ storage suffices where n^2 storage is required for the corresponding dense matrix format.

Consider, for example, the following matrix:

11	0	13	14	0	0
0	22	23	0	25	0
31	32	33	0	35	0
41	0	0	44	45	0
0	52	53	54	55	0
0	0	0	0	0	66

This matrix could be represented in the format described above by three arrays, IROW, JCOL, and MXVALU, as shown in Figure 6-1.

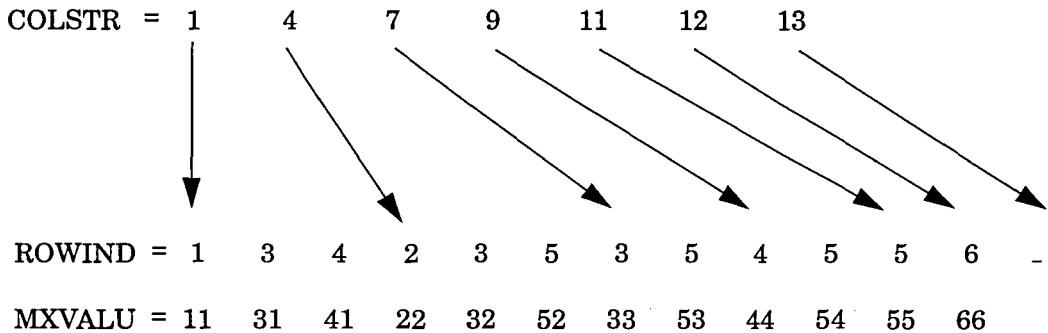
Figure 6-1 Row and Column Index Sparse Matrix Representation

IROW	=	1	3	4	2	3	5	1	2	3	5	1	4	5	2	3	4	5	6
JCOL	=	1	1	1	2	2	2	3	3	3	3	4	4	4	5	5	5	5	6
MXVALU	=	11	31	41	22	32	52	13	23	33	53	14	44	54	25	35	45	55	66

In this example, the matrix entries have been listed in order within each column, and the columns have been listed in order, although the representation does not require that much structure. However, if the entries are required to be ordered by row and column, even less storage is needed. In addition, symmetry in the matrix can be used by storing only the entries, for example, on or below the main diagonal. Other contexts, such as finite element analysis, can make even more concise representations of the locations of the nonzeros.

This package adopts a particular internal format that allows arbitrary symmetric matrices to be stored in $2nz+n+1$ storage locations, where nz represents the number of nonzeros in the lower triangle of the matrix. In essence, the JCOL array, which has repeated entries, is replaced by a shorter array that gives the index of the first element of each column in the MXVALU array and an indication of the number of elements in each column. These two pieces of information per matrix column can be represented in a single array COLSTR of length $n+1$ if the convention is adopted that the number of nonzeros in column j is given by $\text{COLSTR}(j+1) - \text{COLSTR}(j)$ and if $\text{COLSTR}(n+1)$ is set accordingly. Figure 6-2 illustrates this sparse matrix format, known as the *column pointer, row index* representation.

Figure 6-2 Column Pointer, Row Index Sparse Matrix Representation



There are three ways of communicating the coefficient matrix to the package. One is a totally general form, which allows the user to store the matrix outside the package in whatever form is most convenient. The other two ways require that the user store the entire matrix in a form similar to the internal format or at least with all entries in each column contiguous in memory. Any of these three can be used. However, the most general form carries additional overhead in computer time.

Direct Versus Iterative Solution

This package is a *direct* linear equation solver. That is, it computes an explicit factorization of the matrix in a sparse analogy of the dense matrix subprograms DPOFA/DPOSL and DSIFA/DSISL in LINPACK. These capabilities allow for general symmetric sparse matrices and guarantee high accuracy. There are applications where special, factorization-free algorithms can be used effectively. Algorithms appropriate for these cases *iterate* toward a solution, improving an approximate solution at each iteration.

Unfortunately, there is no generally effective iterative algorithm, only a collection of algorithms each effective for particular classes of problems. Used in the appropriate contexts, with only single or a few right-hand sides, and with only limited accuracy required of the solutions, iterative algorithms can be faster than direct methods. Used in the wrong contexts, iterative methods may become inordinately expensive and inaccurate. In contrast, the subprograms described here are designed to function well in general and, additionally, represent the algorithm of choice for many classes of problems.

Fill and Reordering

This linear equation package computes the LDL^T factorization of a matrix A , which is then used to solve the system $Ax = b$. Here, L is a lower triangular

matrix and D is diagonal or quasi-diagonal. However, the sparsity of A is not sufficient to assure that L is sparse. Indeed, L will have nonzeros wherever A 's lower triangle has nonzeros but will also have nonzeros in other positions where A 's entries are zero. These additional entries are known as *fill*. While fill is an intrinsic facet of the factorization process, the amount of fill is often controllable in the following sense. If P is a matrix representing a permutation of the variables of the problem, the factorization of the matrix PAP^T , can often have significantly less fill than does the factorization of A , *provided that the permutation P is chosen appropriately*.

For example, it has been common in solving sparse systems of small order to choose permutations (to *reorder* the matrix) so that its nonzero entries lie close to the main diagonal and then to use subprograms for banded matrices from LINPACK. Banded matrices are a special type of sparse matrix representation. This package is based on more general sparse matrix principles, often resulting in significantly less fill and correspondingly less work in computing the factorization. The principle is the same in both approaches in that the matrix must be reordered so that the factor L is appropriately sparse. The reordering process precedes the actual factorization. The package solves the problem $PAP^T(Px) = (Pb)$. This transformation is invisible to the user.

The subprograms described in this chapter use a very powerful heuristic, the *minimum degree algorithm*, for choosing the reordering. This algorithm is effective in general and is also efficient. In most cases the cost of the reordering process is recovered many times over by the reduction obtained in the cost of the factorization. In some applications, it is common to have several unknowns associated with each grid point of the solution domain. Sparse matrices derived from these applications tend to have some special properties in their structure. A technique known as compression can often improve the performance of the minimum degree reordering. This package provides an option to turn on the compression for these applications.

The numeric factorization algorithm processes columns with like structure simultaneously. This processing yields higher efficiency than older sparse matrix algorithms. The minimum degree ordering is modified to collect columns of the Cholesky factorization which have identical structure. One way to achieve higher computational rates at the cost of performing more numerical operations is to relax the minimum degree ordering so that more columns can be collected together for simultaneous processing while allowing additional fill and hence more floating-point operations. This package provides a relaxation parameter, **maxzer**, which controls the amount of additional fill allowed for each set of collected columns. If **maxzer** = 0 no additional fill will be allowed. Do not use a larger value without verifying that it actually improves performance.

Stability

This package is designed to solve two important classes of symmetric linear systems. One class comprises systems where the coefficient matrix is *positive definite*. The other class comprises the remaining systems where the coefficient matrix is indefinite, but nonsingular. (Negative definite matrices can be treated by the positive definite subprograms by flipping signs.)

The factorization $PAP^T = LDL^T$, with D diagonal, is always stable when A is positive definite. This has the particular effect that the reordering phase can choose any permutation P to maintain sparsity in the factor L . The factorization of an indefinite coefficient matrix is not always stable. However, in a manner closely related to LINPACK subprogram DSIFA, a factorization can always be computed by allowing the matrix D to be a block diagonal matrix, where each block is either of size 1-by-1 or 2-by-2. It is also necessary, as in DSIFA, to incorporate a permutation for *pivoting*, based on the numerical progress of the factorization, to create the proper 2-by-2 blocks that will give stability.

In the dense case, pivoting causes no difficulties. In the sparse case, pivoting can interfere with the permutation chosen to reduce fill. This conflict is dealt with in two ways in this sparse package. First, the effect of pivoting is localized in its modification of the sparsity reordering. Second, a pivoting tolerance is incorporated which allows the user to fine-tune the balance between sparsity and guarantees of stability. This is effected through a parameter, **pvttol**, to the factorization subprogram. **pvttol** is chosen in the range from 0 to 1. A choice of 0 gives a reordering based purely on sparsity; this is the appropriate choice for positive definite matrices. A choice of 1 gives the best guarantee of stability. The value of 0.1 for **pvttol** has been found to be a useful compromise.

Although a purely sparse reordering often works for indefinite matrices, we recommend that you avoid using this option routinely for indefinite problems unless you have already verified its effectiveness on your particular class of problems.

If **pvttol**=0, a faster algorithm is used than when **pvttol**>0. Therefore, you should use **pvttol**=0 if the matrix is known to be positive definite.

We also recommend that matrix condition numbers be computed to assure that sufficient accuracy can be obtained in the solution. Refer to Chapter 4 for more details

Global Communications Array

All of the subprograms in this package use dynamic memory allocation capabilities (refer to Chapter 12) to free users from the often difficult issue of allocating storage for the factors of the matrix. This internal storage is invisible to the user. When the sophisticated lower-level subprograms are used,

knowledge of the internal storage is passed from subprogram to subprogram through a single communications array. This array, called **global** in each of the calling sequences, is a fixed-length double precision array of length 150. It must not be altered by the user because it represents the essential knowledge of the problem. Because it is the only identification for a problem, the user can handle multiple problems simultaneously by having multiple communications arrays with different names.

Error Convention

Each subprogram has an error return flag, **ier**, as one of its arguments. A zero value returned in **ier** is the indication of successful processing. Fatal errors are signaled through negative values. An option allows you to control whether or not this package will print error messages in addition to returning an error flag.

Output Controls

This package differs from most library subroutines in providing optional printed or printable output. The amount of output is controlled by an integer variable, **msglvl**, specifying the *message level*. Setting **msglvl** ≤ 0 suppresses all printed messages, including error messages, and thus should generally be avoided. With **msglvl** = 1, a small amount of runtime statistics and any error messages will be printed. When **msglvl** > 1, the complete set of runtime statistics will be printed. If **msglvl** ≥ 3 , various arrays whose length is on the order of the number of equations will be printed. If **msglvl** ≥ 4 , volumes of output will be produced for debugging purposes.

We recommend that, as a rule, you set **msglvl** = 1. The higher **msglvl** values (≥ 3) are intended for debugging purposes and generate copious amounts of printed output. Further, their use for this purpose may require some knowledge of the data structures being used by the package.

Paths of Control

The key to using this package efficiently is an understanding of possibilities for reusing work. As in solving dense linear systems, factored matrices can be reused as often as needed to solve additional systems. Thus, the subprogram DSLESL can be called repeatedly to solve additional systems with the same coefficient matrix after subprogram DSLEFA or DSLEFS has completed successfully. It is common to encounter sequences of sparse linear systems where the coefficient matrices change, but their sparsity structure, that is, the location of the nonzeros, remains fixed. In such cases, it is necessary to compute a factorization with DSLEFA for each coefficient matrix, but the work of choosing a reordering does not have to be repeated. The possible structures of reuse are illustrated in the following text and Figure 6-3:

What You Need to Know to Use These Subprograms

```
initialization
input matrix structure
reorder matrix
for m = 1 to number_of_structurally_identical_coefficient_matrices do
  input values of matrix entries
  factor matrix
  for r = 1 to
    number_of_right_hand_sides_for_this_coefficient_matrix do
    solve
  endfor
endfor
```

Note that entering matrix values for a new coefficient matrix renders the previously-computed factorization inaccessible. Similarly, entering matrix structure renders any previously-reordered matrix structure inaccessible. However, the save and restart capabilities described in the utility subprograms can be used to save multiple structures or factorizations or to interrupt the package at any point.

Figure 6-3 Paths of Control

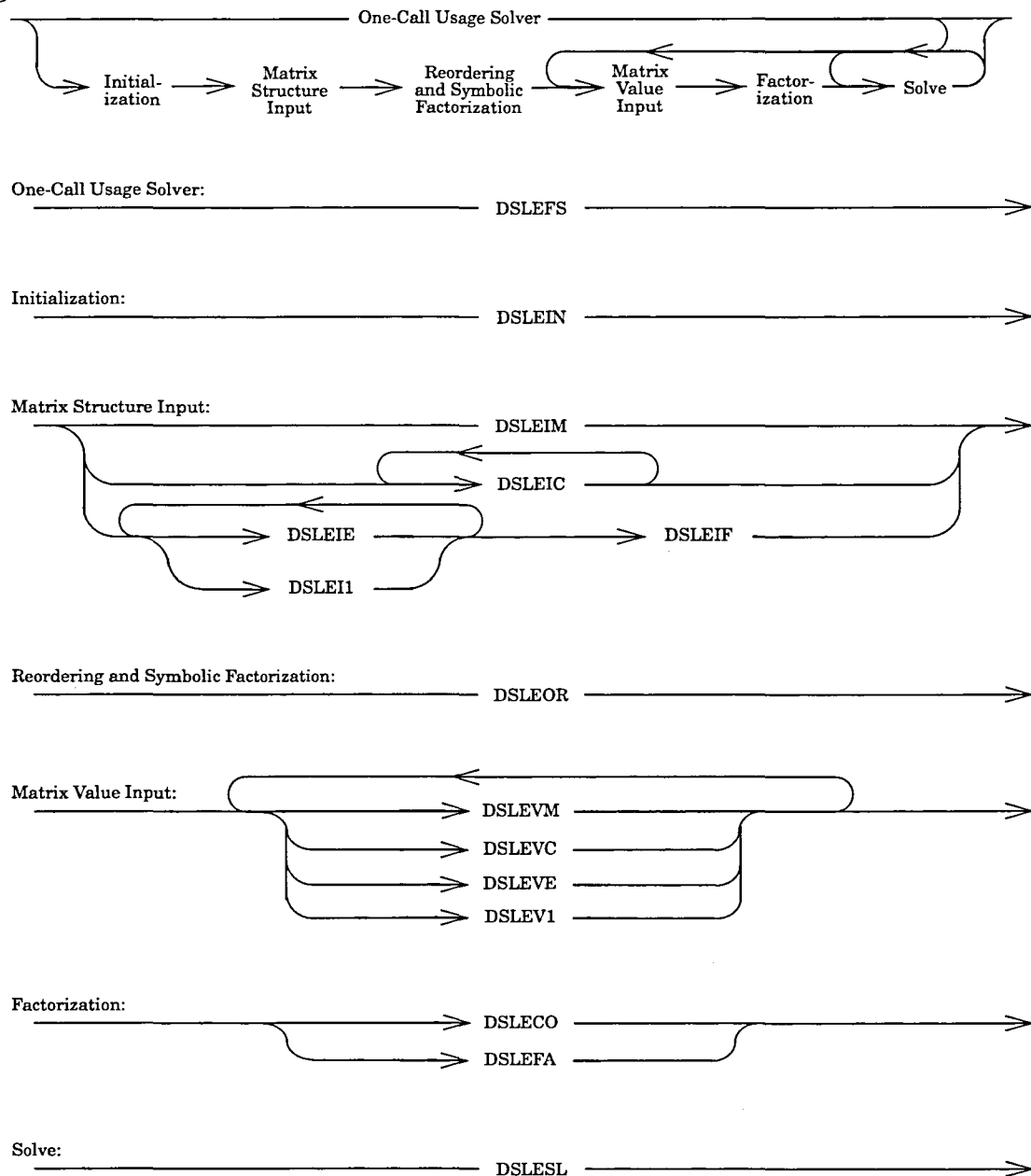
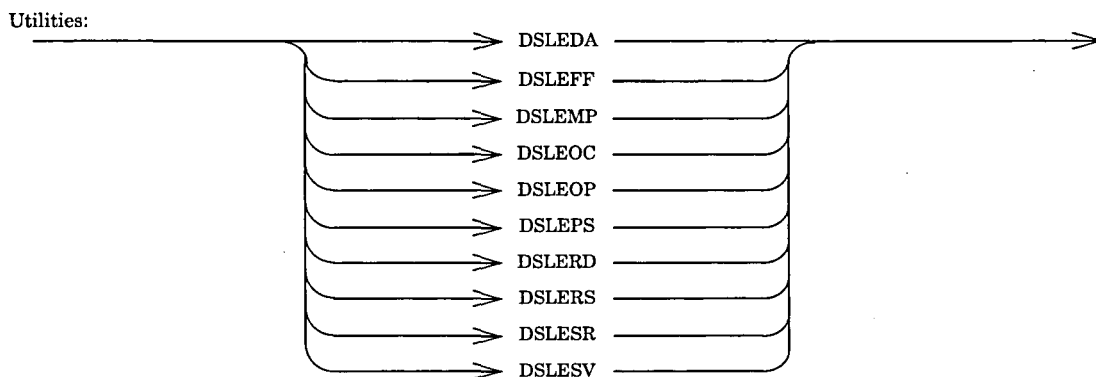


Figure 6-4 Paths of Control (continued)



Sample Program

As illustrated in Figure 6-3, there are many possible paths of control through this sparse matrix package. The following sample program is provided to show one possible path through the package. It is intended to demonstrate that the package is not as difficult to use as Figure 6-3 implies.

In this example, the row and column indices and the corresponding value for each nonzero entry of the matrix are stored in the three arrays `IROW`, `JCOL`, and `MXVALU`. The right-hand-side vector is stored in the array `RHS`. This example demonstrates the use of the subroutines for solving sparse systems of linear equations when subroutine `DSLEFS` cannot be used. The following code assumes that there are `NNZERO` nonzero entries in the matrix, which has `NEQNS` rows and columns. Values of `NNZERO` and `NEQNS` are set prior to this code segment.

```

      INTEGER*4 I,J,K,NEQNS,IROW(NNZERO),JCOL(NNZERO),INRTIA(3),IER
      REAL*8    COND,GLOBAL(150),MXVALU(NNZERO),PVTOL,VALUE
C
C    ... INITIALIZE THE SPARSE MATRIX PACKAGE
C
      CALL DSLEIN (NEQNS,1,6,GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000
C
C    ... INPUT THE MATRIX STRUCTURE
C
      DO 100 K = 1, NNZERO
        I = IROW(K)
        J = JCOL(K)
        CALL DSLEI1 (I,J,GLOBAL,IER)
        IF ( IER .NE. 0 ) GO TO 8000
100 CONTINUE

      CALL DSLEIF (GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000
C
C    ... REORDER THE MATRIX
C
      CALL DSLEOR (0,GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000
C
C    -
C    ... INPUT MATRIX VALUES
C    -
      DO 200 K = 1, NNZERO
        I = IROW(K)
        J = JCOL(K)
        VALUE = MXVALU(K)
        CALL DSLEV1 (I,J,VALUE,GLOBAL,IER)
        IF ( IER .NE. 0 ) GO TO 8000
200 CONTINUE
C
C    -
C    ... FACTOR THE MATRIX AND ESTIMATE ITS CONDITION NUMBER
C    -
      PVTOL = 0.1D0
      CALL DSLECO (PVTOL,COND,INRTIA,GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000
C
C    -
C    ... SOLVE FOR A GIVEN RIGHT HAND SIDE
C    -
      CALL DSLESL (1,RHS,NEQNS,GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000
C

```

Subprograms Included in This Chapter

```
C      ... USE THE SOLUTION STORED IN ARRAY RHS
C
C      .
C      .
C      .
C      ... ERROR TRAP
C
8000 .....
```

Subprograms Included in This Chapter

Following are the subprograms included with VECLIB for the direct solution of sparse symmetric linear equations.

Name DSLEFS
One-Call Usage

Purpose This subprogram provides one-call interface to the sparse linear equation solution package. If this subroutine does not fit your needs, a more flexible interface is available by using the other subroutines in the package.

Usage VECLIB:

```

      INTEGER*4      neqns, maxzer, msglvl, output, colstr(neqns+1),
                     rowind(nnzero), nrhs, ldrhs, inrtia(3), ier
      REAL*8         pvttol, value(nnzero), rhs(ldrhs, nrhs), cond,
                     global(150)
      CALL DSLEFS(neqns, maxzer, pvttol, msglvl, output, colstr, rowind,
                  value, nrhs, rhs, ldrhs, cond, inrtia, global, ier)

```

Input

neqns	Number of equations; neqns > 0.
maxzer	Supernodal relaxation parameter; maxzer ≥ 0. If maxzer > 0, the package allows additional fill for the purpose of forming columns with identical structure to increase the efficiency of the factorization. If maxzer = 0, no additional fill is allowed. Performance studies indicate that a value of 20 to 25 will optimize the factorization execution time with a reduction of 5 to 10%. Because of the additional fill, the solution execution time may increase by 1 to 2%. If many solution vectors are to be computed for each factorization, it is suggested that maxzer = 0 be used. If only a few solution vectors are to be computed maxzer = 25 could be tried but maxzer = 0 is perfectly acceptable. Using values greater than 100 will probably increase the factorization time.
pvttol	Pivoting tolerance; $0 \leq \text{pvttol} \leq 1$. The value 0 gives a reordering which minimizes fill; this value is appropriate for positive definite matrices and provides the greatest efficiency. The value 1 gives the best guarantee of numerical stability for problems not known to be positive definite. The value 0.1 has been found to be a useful compromise between reducing fill and maximizing stability; it is the recommended value when the matrix is not known to be positive definite.

msglvl	Message level for printable output. Options: msglvl ≤ 0 Suppress all output. msglvl = 1 Error messages and summary statistics. msglvl = 2 More complete statistics. msglvl = 3 First stage of debugging output. msglvl = 4 Complete debugging output.
output	Fortran logical unit number to which all printable output will be written.
colstr	colstr(j) gives the index in rowind of the first nonzero in the lower triangular part of column <i>j</i> of the matrix. All of the nonzeros for column <i>j</i> are found, in ascending order, in rowind(colstr(j)), rowind(colstr(j)+1), ..., rowind(colstr(j+1)-1). colstr(neqns+1) must be set to one greater than the total number of nonzeros, nnzero, in the lower triangular part of the matrix.
rowind	List of row indices for all nonzeros, in ascending order within each column, in the lower triangle part of the matrix <i>A</i> .
value	List of values corresponding in position to the indices in rowind .
nrhs	Number of right-hand sides.
ldrhs	The leading dimension of array rhs as declared in the calling program unit, with ldrhs $\geq$ neqns .

Updated	rhs	On input, rhs contains the nrhs right-hand sides. On output, rhs has been overwritten with the computed solutions.
Output	cond	Estimate of the 1-norm condition number.
	inrtia	Array of length 3 containing, respectively, the number of positive eigenvalues, the number of negative eigenvalues, and an indicator if there are zero eigenvalues.
	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
	ier	Status response: ier = 0 Normal return. ier = -101 Error in dynamic storage allocation during matrix structure input. ier = -102 neqns ≤ 0. ier = -106 Unacceptable matrix structure. ier = -201 Error in dynamic storage allocation during ordering. ier = -301 Error in dynamic storage allocation during symbolic factorization. ier = -302 Internal error. ier = -401 Error in dynamic storage allocation during value input. ier = -501 Error in dynamic storage allocation during factorization. ier = -502 Error in dynamic storage allocation during factorization. ier = -503 Exactly zero pivot encountered during factorization; matrix is singular. ier = -602 nrhs ≤ 0. ier = -603 ldrhs < neqns .
Notes	Calling DSLEFS is equivalent to calling DSLEIN, DSLEIM, DSLEOR, DSLECO, and DSLESL in sequence. You may follow the call to DSLEFS with other calls to subprograms in the package to solve additional right-hand sides, to input new matrix values, to input a new matrix structure, or to perform utility functions.	

Example Solve the small (order 6) sparse system of linear equations where the matrix A and right-hand side b are

$$\begin{array}{rcc}
 & \begin{matrix} 4.0 & 0.0 & 1.0 & 1.0 & 0.0 & 0.0 \end{matrix} & \begin{matrix} 1.0 \\ 0.0 \\ 1.0 \\ 1.0 \\ 0.0 \\ 0.0 \end{matrix} \\
 A = & \begin{matrix} 0.0 & 4.0 & 1.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 1.0 & 4.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 4.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 1.0 & 1.0 & 4.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 1.0 \end{matrix} & b = \begin{matrix} 1.0 \\ 0.0 \\ 1.0 \\ 1.0 \\ 0.0 \\ 0.0 \end{matrix}
 \end{array}$$

A supernode relaxation factor of 0 (no extra fill allowed) and a pivoting tolerance of 0.1 will be used. Printed output will be to Fortran logical unit 6, and only printed error messages and some runtime statistics will be printed.


```

INTEGER*4 COLSTR(7),ROWIND(13),INRTIA(3),IER
REAL*8     VALUES(13),RHS(6),COND,GLOBAL(150)

COLSTR(1) = 1
COLSTR(2) = 4
COLSTR(3) = 7
COLSTR(4) = 9
COLSTR(5) = 12
COLSTR(6) = 13
COLSTR(7) = 14

ROWIND(1) = 1
ROWIND(2) = 3
ROWIND(3) = 4
ROWIND(4) = 2
ROWIND(5) = 3
ROWIND(6) = 5
ROWIND(7) = 3
ROWIND(8) = 5
ROWIND(9) = 4
ROWIND(10) = 5
ROWIND(11) = 6
ROWIND(12) = 5
ROWIND(13) = 6

VALUES(1) = 4.0
VALUES(2) = 1.0
VALUES(3) = 1.0
VALUES(4) = 4.0
VALUES(5) = 1.0
VALUES(6) = 1.0
VALUES(7) = 4.0
VALUES(8) = 1.0
VALUES(9) = 4.0
VALUES(10) = 1.0
VALUES(11) = 1.0
VALUES(12) = 4.0
VALUES(13) = 1.0

RHS(1) = 1.0
RHS(2) = 0.0
RHS(3) = 1.0
RHS(4) = 1.0
RHS(5) = 0.0
RHS(6) = 0.0

CALL DSLEFS (6,0,0.1D0,1,6,COLSTR,ROWIND,VALUE,
             1,RHS,6,COND,INRTIA,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF

```

Name	DSLEIN Initialize Sparse Linear Equations	
Purpose	This subprogram provides the necessary information to begin processing with the sparse linear equation solution package.	
Usage	VECLIB: <pre> INTEGER*4 neqns, msglvl, output, ier REAL*8 global(150) CALL DSLEIN(neqns, msglvl, output, global, ier) </pre>	
Input	neqns	Order of matrix (number of degrees of freedom); neqns > 0.
	msglvl	Message level for printable output:
		msglvl ≤ 0 Suppress all output.
		msglvl = 1 Error messages and summary statistics.
		msglvl = 2 More complete statistics.
		msglvl = 3 First stage of debugging output.
		msglvl = 4 Complete debugging output.
Output	output	Fortran logical unit number to which all printable output will be written.
	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
	ier	Status response:
		ier = 0 Normal return.
		ier = -100 Too many systems initialized.
		ier = -101 Error in dynamic storage allocation.
		ier = -102 neqns ≤ 0.
Example	Prepare to solve a system of equations of order 10,000. Obtain error messages and standard minimal output on Fortran logical unit 6. <pre> INTEGER*4 NEQNS, IER REAL*8 GLOBAL(150) NEQNS = 10000 CALL DSLEIN (NEQNS, 1, 6, GLOBAL, IER) IF (IER .NE. 0) THEN handle error condition ENDIF </pre>	

Name	DSLEI1 Matrix Structure Input by Single Entry	
Purpose	This subprogram adds a single entry in the (irow , jcol) position in the lower triangle to the set of known nonzeros for the sparse matrix.	
Usage	VECLIB: <pre> INTEGER*4 irow, jcol, ier REAL*8 global(150) CALL DSLEI1(irow, jcol, global, ier) </pre>	
Input	irow	Row index of the nonzero entry; $jcol \leq irow \leq neqns$, where neqns is the number of equations specified in the call to DSLEIN. Input of diagonal entries is optional.
	jcol	Column index of the nonzero entry; $1 \leq jcol \leq neqns$, where neqns is the number of equations specified in the call to DSLEIN.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: <pre> ier = 0 Normal return. ier = -100 Incorrect processing path; DSLEIN not called or matrix input already finished. ier = -101 Error in dynamic storage allocation. ier = -104 Illegal value for jcol. ier = -105 Illegal value for irow. </pre>
Notes	Calls to DSLEI1 and DSLEIE can be intermixed. DSLEIC and DSLEIM cannot be used if DSLEI1 or DSLEIE are used.	
Example	Add the entry in row 3035, column 1024 to the list of nonzeros in the matrix.	

```

      INTEGER*4 I,J,IER
      REAL*8    GLOBAL(150)
      I = 3035
      J = 1024
      CALL DSLEI1 (I,J,GLOBAL,IER)
      IF ( IER .NE. 0 ) THEN
         handle error condition
      ENDIF
  
```

Name	DSLEIC Matrix Structure Input by Column	
Purpose	This subprogram adds a list of indices in a single column in the lower triangle to the set of known nonzeros for the sparse matrix.	
Usage	VECLIB: <pre> INTEGER*4 jcol, nzcol, jrowin(nzcol), ier REAL*8 global(150) CALL DSLEIC(jcol, nzcol, jrowin, global, ier) </pre>	
Input	jcol	Column index; $1 \leq \text{jcol} \leq \text{neqns}$. Columns must be presented in order from 1 to neqns , except that columns with no entries can be skipped.
	nzcol	Number of nonzeros in column jcol of the matrix; nzcol ≥ 0 .
	jrowin	List of row indices for all nonzeros, in ascending order, in the lower triangular part of column jcol of the matrix; jcol $\leq \text{jrowin}(1) < \text{jrowin}(2) < \dots < \text{jrowin}(\text{nzcol}) \leq \text{neqns}$. Input of diagonal entries is optional
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: <pre> ier = 0 Normal return. ier = -100 Incorrect processing path; DSLEIN not called or matrix input already finished. ier = -101 Error in dynamic storage allocation. ier = -103 Illegal value for nzcol. ier = -104 Illegal value for jcol, or out of order. ier = -106 Illegal value for at least one entry in jrowin or entries out of order. </pre>
Notes	The entire matrix structure must be input with DSLEIC if it is used. Its use is not compatible with DSELE1, DSLEIE, or DSLEIM. If the matrix entries are available by column, using DSLEIC is more efficient than using DSLEI1.	

Example Column 4519 has entries in rows 1, 2735, 4519, 4520, 4521, 6000, 6002 and 6004. Add all five entries in the lower triangular part to the list of nonzeros in the matrix. Columns 1 to 4518 already have been input to the package using DSLEIC.

```
INTEGER*4 J,NZCOL,JROWIN(100),IER
REAL*8    GLOBAL(150)

J = 4519
NZCOL = 5
JROWIN(1) = 4520
JROWIN(2) = 4521
JROWIN(3) = 6000
JROWIN(4) = 6002
JROWIN(5) = 6004

CALL DSLEIC (J,NZCOL,JROWIN,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name	DSLEIE Matrix Structure Input by Finite Element	
Purpose	This subprogram adds indices to the set of known nonzeros in the lower triangle of the sparse matrix corresponding to a finite element or clique.	
Usage	VECLIB: <pre> INTEGER*4 nnode, nodlst(nnode), ier REAL*8 global(150) CALL DSLEIE(nnode, nodlst, global, ier) </pre>	
Input	nnode nodlst	Number of nodes in the finite element or clique. List of nodes. All pairs (nodlst (<i>i</i>), nodlst (<i>j</i>)) in the lower triangle are added to the sparsity structure of the matrix.
Updated	nodlst global	The order of the values in nodlst may be changed. Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: <pre> ier = 0 Normal return. ier = -100 Incorrect processing path; DSLEIN not called or matrix input already finished. ier = -101 Error in dynamic storage allocation. ier = -107 Illegal value for nnode. ier = -108 Illegal value for at least one entry in nodlst. </pre>
Notes	Calls to DSLEIE can be intermixed with calls to DSLEI1. DSLEIC and DSLEIM cannot be used if DSLEI1 or DSLEIE is used.	

Example Rows and columns 345, 346, 347 and 989 form a small dense submatrix of A. Add the positions consisting of all pairs of numbers from this set to the list of nonzeros in the matrix.

```
INTEGER*4 NNODE,NODLST(10),IER
REAL*8    GLOBAL(150)
NNODE = 4
NODLST(1) = 345
NODLST(2) = 346
NODLST(3) = 347
NODLST(4) = 989

CALL DSLEIE (NNODE,NODLST,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name	DSLEIM Matrix Structure Input by Matrix	
Purpose	This subprogram specifies locations of all nonzeros in the lower triangle of the sparse matrix.	
Usage	VECLIB: <pre> INTEGER*4 neqns, nnzero, colstr(neqns+1), rowind(nnzero), ier REAL*8 global(150) CALL DSLEIM(colstr, rowind, global, ier) </pre>	
Input	colstr	colstr(j) gives the index in rowind of the first nonzero in the lower triangular part of column <i>j</i> of the matrix. All of the nonzeros for column <i>j</i> are found, in ascending order, in rowind(colstr(j)), rowind(colstr(j)+1), ..., rowind(colstr(j+1)-1). colstr(neqns+1) must be set to one greater than the total number of nonzeros, nnzero, in the lower triangular part of the matrix.
	rowind	List of row indices for all nonzeros in ascending order within each column in the lower triangle part of the matrix. Input of diagonal entries is optional.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: <pre> ier = 0 Normal return. ier = -100 Incorrect processing path; DSLEIN not called or matrix input already finished. ier = -101 Error in dynamic storage allocation. ier = -106 Illegal value for at least one entry in colstr (not increasing or negative) or invalid entries in rowind (entries out of order within a column or out of the lower triangle). </pre>

Notes This is the most efficient mechanism for specifying the nonzero structure, but the entire matrix structure must be input with DSLEIM if it is used. Its use is not compatible with DSLEI1, DSLEIE, or DSLEIC.

Example Input a small (order 6) sparse matrix with this structure:

```

      x  0  x  x  0  0
      0  x  x  0  x  0
      x  x  x  0  x  0
      x  0  0  x  x  0
      0  x  x  x  x  0
      0  0  0  0  0  x

```

```

INTEGER*4 COLSTR(7),ROWIND(6),IER
REAL*8    GLOBAL(150)

COLSTR(1) = 1
COLSTR(2) = 3
COLSTR(3) = 5
COLSTR(4) = 6
COLSTR(5) = 7
COLSTR(6) = 7
COLSTR(7) = 7

ROWIND(1) = 3
ROWIND(2) = 4
ROWIND(3) = 3
ROWIND(4) = 5
ROWIND(5) = 5
ROWIND(6) = 5

CALL DSLEIM (COLSTR,ROWIND,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF

```

Name	DSLEIF End of Matrix Structure Input	
Purpose	This subprogram specifies the end of structure input for the matrix. DSLEIF is used only if routines DSLEI1 and/or DSLEIE were the mechanism by which the structure was input.	
Usage	VECLIB: <pre> INTEGER*4 ier REAL*8 global(150) CALL DSLEIF(global, ier) </pre>	
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: <pre> ier = 0 Normal return. ier = -100 Incorrect processing path; DSLEIN not called or matrix input already finished. ier = -101 Error in dynamic storage allocation. </pre>
Example	The nonzero structure of a pair of finite element matrices was passed to the package using repeated calls to subroutine DSLEIE. Signal that no more nonzeros will be added to the matrix. <pre> INTEGER*4 IER REAL*8 GLOBAL(150) CALL DSLEIF (GLOBAL, IER) IF (IER .NE. 0) THEN handle error condition ENDIF </pre>	

Name	DSLEOR Reordering and Symbolic Factorization	
Purpose	This subprogram reorders the matrix whose pattern of nonzeros has been input to the package to obtain an efficient sparse factorization. It then constructs data structures that represent the factorization.	
Usage	VECLIB: <pre> INTEGER*4 maxzer, ier REAL*8 global(150) CALL DSLEOR(maxzer, global, ier) </pre>	
Input	maxzer	Supernodal relaxation parameter; maxzer ≥ 0 . If maxzer > 0 , the package allows additional fill for the purpose of forming columns with identical structure to increase the efficiency of the factorization. If maxzer = 0 no additional fill is allowed. Performance studies indicate that a value of 20 to 25 will optimize the factorization execution time with a reduction of 5 to 10%. Because of the additional fill, the solution execution time may increase by 1 to 2%. If many solution vectors are to be computed for each factorization, it is suggested that maxzer = 0 be used. If only a few solution vectors are to be computed maxzer = 25 could be tried but maxzer = 0 is perfectly acceptable. Using values greater than 100 will probably increase the factorization time.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: <pre> ier = 0 Normal return. ier = -200 Incorrect processing path; structure input not completed or value input subroutines already called. ier = -201 Error in dynamic storage allocation. </pre>

Example The sparse matrix has been communicated to the page using DSLEIN and DSLEI1, DSLEIE, and DSLEIF, and DSLEIC or DSLEIM. The next step is obtaining good reordering and performing symbolic factorization for the numeric factorization phase. The value of MAXZER = 0 is used so that no extra fill will occur in the factorization.

```
INTEGER*4 IER
REAL*8    GLOBAL(150)
CALL DSLEOR (0,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name	DSLEV1 Matrix Value Input by Single Entry	
Purpose	This subprogram adds to the value of the entry in the (irow, jcol) position in the lower triangle of the sparse matrix.	
Usage	VECLIB: <pre> INTEGER*4 irow, jcol, ier REAL*8 value, global(150) CALL DSLEV1(irow, jcol, value, global, ier) </pre>	
Input	<pre> irow jcol value </pre>	Row index of the nonzero entry, $jcol \leq irow \leq neqns$. Column index of the nonzero entry, $1 \leq jcol \leq neqns$. Numeric value that will be added to any previous values input for this location.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: <pre> ier = 0 Normal return. ier = -400 Incorrect processing path; DSLEOR not called. ier = -401 Error in dynamic storage allocation. ier = -402 Subscript pair (irow, jcol) was not specified in structure input. No room for value. </pre>
Notes	Calls to DSLEV1, DSLEVC, DSLEVE, and DSLEVM can be intermixed.	
Example	Store the value 4.523×10^{-5} as the nonzero entry in row 3035, column 1024 of the matrix.	

```

      INTEGER*4 I, J, IER
      REAL*8    VALUE, GLOBAL(150)
      I = 3035
      J = 1024
      VALUE = 4.523D-5
      CALL DSLEV1 (I, J, VALUE, GLOBAL, IER)
      IF ( IER .NE. 0 ) THEN
         handle error condition
      ENDIF

```

Name	DSLEVC Matrix Value Input by Column	
Purpose	This subprogram adds to the values of a list of nonzero entries in the lower triangle of a single column of the sparse matrix.	
Usage	VECLIB: INTEGER*4 jcol, nzcol, jrowin(nzcol), ier REAL*8 values(nzcol), global(150) CALL DSLEVC(jcol, nzcol, jrowin, values, global, ier)	
Input	jcol	Column index; $1 \leq \text{jcol} \leq \text{neqns}$.
	nzcol	Number of nonzeros in the list of nonzeros for column jcol of the matrix; $\text{nzcol} \geq 0$.
	jrowin	List of ascending order row indices for nonzeros in the lower triangular part of column jcol of the matrix; $\text{jcol} \leq \text{jrowin}(1) < \text{jrowin}(2) < \dots < \text{jrowin}(\text{nzcol}) \leq \text{neqns}$.
	values	List of values corresponding to the positions specified by jcol and jrowin.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: ier = 0 Normal return. ier = -400 Incorrect processing path; DSLEOR not called. ier = -401 Error in dynamic storage allocation. ier = -402 At least one subscript pair (jrowin(k),jcol) was not specified in structure input. No room for value.
Notes	Calls to DSLEV1, DSLEVC, DSLEVE and DSLEVM can be intermixed. It is not necessary that all entries in the column jcol be included in jrowin. If the matrix entries are available by column, using DSLEVC is more efficient than using DSLEV1.	

Example Column 4519 has entries in rows 1, 2735, 4519, 4520, 4521, 6000, 6002, and 6004. Add the value 1.0 to each of these positions in the matrix.

```
INTEGER*4 J,NZCOL,JROWIN(100),IER
REAL*8    VALUES(100),GLOBAL(150)
J = 4519
NZCOL = 6
JROWIN(1) = 4519
JROWIN(2) = 4520
JROWIN(3) = 4521
JROWIN(4) = 6000
JROWIN(5) = 6002
JROWIN(6) = 6004

VALUES(1) = 1.0D0
VALUES(2) = 1.0D0
VALUES(3) = 1.0D0
VALUES(4) = 1.0D0
VALUES(5) = 1.0D0
VALUES(6) = 1.0D0

CALL DSLEVC (J,NZCOL,JROWIN,VALUES,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name	DSLEVE Matrix Value Input by Finite Element	
Purpose	This subprogram adds to the values of a set of nonzero entries in the lower triangle of the sparse matrix corresponding to a finite element or clique.	
Usage	VECLIB: <pre> INTEGER*4 nnode, ldelmx, nodlst(nnode), ier REAL*8 elmmtx(ldelmx, nnode), global(150) CALL DSLEVE(nnode, nodlst, elmmtx, ldelmx, global, ier) </pre>	
Input	nnode	Number of nodes in the finite element or clique.
	nodlst	List of nodes in element or clique. Values for all pairs (nodlst(i),nodlst(j)) are added to the values of the matrix.
	elmmtx	Array containing values to be added to the matrix. Only the lower triangle (including the diagonal) of elmmtx is referenced. The value in elmmtx(k,l) is added to the value in position (nodlst(k),nodlst(l)) in the sparse matrix.
	ldelmx	The leading dimension of array elmmtx as declared in the calling program unit, with ldelmx $\geq$ nnode .
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: <pre> ier = 0 Normal return. ier = -400 Incorrect processing path; DSLEOR not called. ier = -401 Error in dynamic storage allocation. ier = -402 At least one subscript pair (nodlst(k),nodlst(l)) was not specified in structure input. No room for value. </pre>
Notes	Calls to DSLEV1, DSLEVC, DSLEVE, and DSLEVM can be intermixed. Using DSLEVE is more efficient in finite element contexts, where the matrix is created as the sum of elemental matrices, if the user has stored the elemental matrices rather than the assembled matrix.	

Example Rows and columns 345, 346, 347, and 989 form a small dense submatrix of A . Add the value 1.0 to the values in the positions consisting of all pairs of numbers from this set to the list of nonzeros in the matrix.

```

      INTEGER*4 NNODE,NODLST(10),IER
      REAL*8     ELMMTX(10,10),GLOBAL(150)
      NNODE = 4
      NODLST(1) = 345
      NODLST(2) = 346
      NODLST(3) = 347
      NODLST(4) = 989
      DO 200 K = 1, NNODE
        DO 100 L = K, NNODE
          ELMMTX(K,L) = 1.0D0
100      CONTINUE
200 CONTINUE
      CALL DSLEVE (NNODE,NODLST,ELMMTX,10,GLOBAL,IER)
      IF ( IER .NE. 0 ) THEN
        handle error condition
      ENDIF

```

Name	DSLEVM Matrix Value Input by Matrix	
Purpose	This subprogram adds to the values of many or all of the nonzero entries in the lower triangle of the sparse matrix.	
Usage	VECLIB: <pre> INTEGER*4 neqns, nnzero, colstr(neqns+1), rowind(nnzero), ier REAL*8 values(nnzero), global(150) CALL DSLEVM(colstr, rowind, values, global, ier) </pre>	
Input	colstr	colstr(<i>j</i>) gives the index in rowind of the first nonzero in the lower triangular part of column <i>j</i> of the matrix. All of the nonzeros for column <i>j</i> are found, in ascending order, in rowind(colstr(<i>j</i>)), rowind(colstr(<i>j</i>)+1), ..., rowind(colstr(<i>j</i> +1)-1). colstr(neqns+1) must be set to one greater than the total number of nonzeros in the lower triangular part of the matrix.
	rowind	List of row indices for many or all of the nonzeros in ascending order within each column in the lower triangle part of the matrix. Diagonal entries may be present as well.
	values	List of values corresponding in position to the indices in rowind. These values will be added to any values already present in the matrix.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: <pre> ier = 0 Normal return. ier = -400 Incorrect processing path; DSLEOR not called. ier = -401 Error in dynamic storage allocation. ier = -402 At least one subscript pair was not specified in structure input. No room for value. </pre>

Notes This is the most efficient mechanism for specifying the nonzero values. Normally, DSLEVM is used in conjunction with DSLEIM. Additional entries or modifications can be entered with DSLEV1, DSLEVC, or DSLEVE.

Example Input the values for the following small (order 6) sparse matrix problem.

4.0	0.0	1.0	1.0	0.0	0.0
0.0	4.0	1.0	0.0	1.0	0.0
1.0	1.0	4.0	0.0	1.0	0.0
1.0	0.0	0.0	4.0	1.0	0.0
0.0	1.0	1.0	1.0	4.0	0.0
0.0	0.0	0.0	0.0	0.0	4.0

Use DSLEVM and DSLEV1 to input this matrix.

```

INTEGER*4 COLSTR(7),ROWIND(6),IER
REAL*8    VALUES(6),DIAGVL,GLOBAL(150)

COLSTR(1) = 1
COLSTR(2) = 3
COLSTR(3) = 5
COLSTR(4) = 6
COLSTR(5) = 7
COLSTR(6) = 7
COLSTR(7) = 7

ROWIND(1) = 3
ROWIND(2) = 4
ROWIND(3) = 3
ROWIND(4) = 5
ROWIND(5) = 5
ROWIND(6) = 5

VALUES(1) = 1.0D0
VALUES(2) = 1.0D0
VALUES(3) = 1.0D0
VALUES(4) = 1.0D0
VALUES(5) = 1.0D0
VALUES(6) = 1.0D0

CALL DSLEVM (COLSTR,ROWIND,VALUES,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF

DIAGVL = 4.0
DO 100 I = 1, NEQNS
    CALL DSLEV1 (I,I,DIAGVL,GLOBAL,IER)
    IF ( IER .NE. 0 ) THEN
        handle error condition
    ENDIF
100 CONTINUE

```

This matrix could also have been input strictly with DSLEVM by expanding COLSTR, ROWIND, and VALUES to include the diagonal. This action would have avoided the calls to DSLEV1. Including the diagonal entries in COLSTR and ROWIND is also acceptable to DSLEIM.

Name	DSLECO Numeric Factorization and Condition Number Estimation	
Purpose	This subprogram computes the numeric factorization and condition number estimate of the sparse symmetric matrix input to the package.	
Usage	VECLIB: INTEGER*4 inrtia(3), ier REAL*8 pvttol, cond, global(150) CALL DSLECO(pvttol, cond, inrtia, global, ier)	
Input	pvttol	Pivoting tolerance; $0 \leq \text{pvttol} \leq 1$. The value 0 gives a reordering which minimizes fill; this value is appropriate for positive definite matrices and provides the greatest efficiency. The value 1 gives the best guarantee of numerical stability for problems not known to be positive definite. The value 0.1 has been found to be a useful compromise between reducing fill and maximizing stability; it is the recommended value when the matrix is not known to be positive definite.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	cond	Estimate of the 1-norm condition number.
	inrtia	Array of length 3 containing, respectively, the number of positive eigenvalues, the number of negative eigenvalues, and an indicator if there are zero eigenvalues.
	ier	Status response: ier = 0 Normal return. ier = -500 Incorrect processing path; value input not performed. ier = -501 Error in dynamic storage allocation during factorization. ier = -502 Error in dynamic storage allocation during factorization. ier = -503 Exactly zero pivot encountered during factorization; matrix is singular.

Example Factor the matrix input to the package and estimate its condition number. Use a pivot tolerance of 0.1.

```
INTEGER*4 INRTIA(3),IER
REAL*8    COND,GLOBAL(150)
CALL DSLECO (0.1D0,COND,INRTIA,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name	DSLEFA Numeric Factorization	
Purpose	This subprogram computes the numeric factorization of the sparse symmetric matrix input to the package. No condition number estimation is performed.	
Usage	VECLIB: <pre> INTEGER*4 inrtia(3), ier REAL*8 pvttol, global(150) CALL DSLEFA(pvttol, inrtia, global, ier) </pre>	
Input	pvttol	Pivoting tolerance; $0 \leq \text{pvttol} \leq 1$. The value 0 gives a reordering which minimizes fill; this value is appropriate for positive definite matrices and provides the greatest efficiency. The value 1 gives the best guarantee of numerical stability for problems not known to be positive definite. The value 0.1 has been found to be a useful compromise between reducing fill and maximizing stability; it is the recommended value when the matrix is not known to be positive definite.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	inrtia	Array of length 3 containing, respectively, the number of positive eigenvalues, the number of negative eigenvalues, and an indicator if there are zero eigenvalues.
	ier	Status response: <pre> ier = 0 Normal return. ier = -500 Incorrect processing path; value input not performed. ier = -501 Error in dynamic storage allocation during factorization. ier = -502 Error in dynamic storage allocation during factorization. ier = -503 Exactly zero pivot encountered during factorization; matrix is singular. </pre>

Example Factor the matrix input to the package. Use a pivot tolerance of 0.1.

```
INTEGER*4 INRTIA(3),IER
REAL*8    GLOBAL(150)
CALL DSLEFA (0.1D0,COND,INRTIA,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```


Name	DSLESL Solve	
Purpose	This subprogram computes the solution for a set of right-hand side vectors given a numeric factorization performed by DSLECO or DSLEFA.	
Usage	VECLIB: <pre> INTEGER*4 nrhs, ldrhs, ier REAL*8 rhs(ldrhs, nrhs), global(150) CALL DSLESL(nrhs, rhs, ldrhs, global, ier) </pre>	
Input	nrhs	Number of right-hand sides; $\text{nrhs} > 0$.
	ldrhs	The leading dimension of array rhs as declared in the calling program unit, with $\text{ldrhs} \geq \text{neqns}$.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
	rhs	On input, rhs contains the nrhs right-hand sides. On output, rhs has been overwritten with computed solutions.
Output	ier	Status response: <pre> ier = 0 Normal return. ier = -600 Incorrect processing path; numeric factorization has not been performed. ier = -601 Error in dynamic storage allocation. ier = -602 nrhs ≤ 0. ier = -603 ldrhs < neqns. </pre>
Example	Given the following right-hand side vector, compute the solution given the numerical factorization computed by DSLECO or DSLEFA. <pre> 1.0 0.0 rhs = 1.0 1.0 0.0 0.0 </pre>	

```
INTEGER*4 IER
REAL*8    RHS(6),GLOBAL(150)

RHS(1) = 1.0
RHS(2) = 0.0
RHS(3) = 1.0
RHS(4) = 1.0
RHS(5) = 0.0
RHS(6) = 0.0

CALL DSLESL (1,RHS,6,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name	DSLEDA Deallocate Working Storage	
Purpose	Deallocate working storage at the end of processing. If the program using the package is to continue execution after use of the package is completed, it is recommended that the user deallocate the dynamically-allocated working storage with subroutine DSLEDA.	
Usage	VECLIB: INTEGER*4 ier REAL*8 global(150) CALL DSLEDA(global, ier)	
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: ier = 0 Normal return. ier = -200 No system exists for this global communications array.
Example	Deallocate working storage after use of package. REAL*8 GLOBAL(150) CALL DSLEDA (GLOBAL, IER) IF (IER .NE. 0) THEN handle error condition ENDIF	

Name	DSLEFF Specify Singularity Treatment	
Purpose	This subprogram provides a mechanism to deal with singular (or nearly singular) matrices by specifying a course of action to follow when zero or small diagonal elements are encountered in the factorization during subsequent calls to DSLEFA or DSLECO which specify <code>pvtol = 0</code> .	
Usage	VECLIB: <pre> INTEGER*4 iabort, ier REAL*8 fudge, tol, global(150) CALL DSLEFF(iabort, fudge, tol, global, ier) </pre>	
Input	iabort	Flag to specify when to terminate the factorization, as follows: iabort = 1 Terminate the factorization if a diagonal element becomes smaller than <code>-tol</code> , i.e., when a diagonal element is too negative. iabort = 2 Terminate the factorization if the absolute value of a diagonal element becomes smaller than <code>tol</code> , i.e., when a diagonal element is near zero. iabort = 3 Do not terminate the factorization on negative or near zero diagonal values. This is the default if DSLEFF has not been called.
	fudge	User specified value of the fudge factor, <code>fudge > 0</code> . Used when <code>iabort = 1</code> or <code>iabort = 3</code> . When a pivot that is smaller in absolute value than <code>tol</code> is encountered in DSLEFA or DSLECO, the near-zero pivot value is replaced by <code>fudge</code> . The index of the first such pivot is returned in non-permuted order by the <code>ier</code> argument of DSLEFA or DSLECO. The location of all of the fudged pivots can be determined if <code>fudge ≤ tol</code> by calling DSLERD to get the diagonal, and then searching it for values matching <code>fudge</code> .

	tol	User-specified value of the tolerance. Those diagonal elements whose absolute values are less than tol during the numerical factorization will be replaced by fudge . Default value is 0 if DSLEFF has not been called
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: ier = 0 Normal return. ier = -1000 Incorrect processing path; should be called before DSLEFA or DSLECO. ier = -1001 tol < 0.

Name	DSLEOC Output Control	
Purpose	This subprogram may be called at any point after subprogram DSLEIN to alter either the output message level or the Fortran output unit number for message output.	
Usage	VECLIB: <pre> INTEGER*4 msglvl, output REAL*8 global(150) CALL DSLEOC(msglvl, output, global) </pre>	
Input	<pre> msglvl output </pre>	Message level for printable output: <pre> msglvl ≤ 0 Suppress all output. msglvl = 1 Error messages and summary statistics. msglvl = 2 More complete statistics. msglvl = 3 First stage of debugging output. msglvl = 4 Complete debugging output. </pre> Fortran logical unit number to which all output will be written.
Updated	<pre>global</pre>	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Example	Increase message level from 1 to 2 while leaving the output unit the same at 6. <pre> REAL*8 GLOBAL(150) CALL DSLEOC (2,6,GLOBAL) </pre>	

Name	DSLEOP Select the Reordering Scheme	
Purpose	This subprogram selects the reordering scheme used by the solver. The default is to use the Multiple Minimum Degree (MMD) reordering.	
Usage	VECLIB: <pre> CHARACTER*2 order INTEGER*4 ier REAL*8 global(150) CALL DSLEOP(order, global, ier) </pre>	
Input	order	The reordering scheme used by the solver: order = 'MM' or 'mm' Use the Multiple Minimum Degree ordering. order = 'NO' or 'no' No internal reordering is performed; the original ordering is used in all phases of the solver.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: ier = 0 Normal return. ier = -1100 Incorrect processing path; should be called before DSLEOR. ier = -1101 Invalid value for order.
Notes	Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the CALL statement may be improved by coding the order argument as 'MMD Reordering' or 'no reordering'.	

Name	DSLEPS Print Statistics
Purpose	This subprogram prints available statistics about the original matrix, amount of work space in use, amount required for the next phase, and maximum amount used thus far. Also, this subprogram prints storage and arithmetic requirements, CPU time used, and computational rate for Cholesky factorization. The amount of information printed depends on the stage of execution. The number of lines of output ranges from 8 to 30, with the width of the lines being less than 80 characters.
Usage	VECLIB: REAL*8 global(150) CALL DSLEPS(global)
Input	global Global communications array for this problem.
Example	Print the statistics after the package has completed a numeric solution (either after DSLESL or DSLEFS). REAL*8 GLOBAL(150) CALL DSLEPS (GLOBAL)

Name	DSLERD Return the Diagonal Elements of the Matrix in its Current Form		
Purpose	This subprogram is called after the matrix value input is complete. It returns the diagonal elements of the sparse matrix in its current form: • If the matrix has not been factored, diag (A) is returned. • If the matrix has been factored, diag (D) is returned. The diagonal elements can be returned in the original order or in the permuted order if reordering has been completed.		
Usage	VECLIB: <pre> CHARACTER*1 job INTEGER*4 ier REAL*8 diag(neqns), global(150) CALL DSLERD(job, diag, global, ier) </pre>		
Input	job	Option flag:	
		job = 'O' or 'o'	Return the diagonal elements of the current form of the matrix in the original order.
		job = 'P' or 'p'	Return the diagonal elements of the current form of the matrix in the permuted order determined by the reordering algorithm.
	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.	
Output	diag	Array of diagonal elements of the current form of the sparse matrix.	
	ier	Status response:	
		ier = 0	Normal return.
		ier = -800	Incorrect processing path; matrix value input not completed. Diagonal values are not returned.
		ier = -801	Invalid value for job.

Notes

Actual character arguments in a subroutine call may be longer than the corresponding dummy arguments. Therefore, readability of the CALL statement may be improved, for example, by coding the job argument as 'Original Order' for 'O', or 'Permuted Order' for 'P'.

Name	DSLERS Restore Problem State from a Savefile	
Purpose	This subprogram restores the working problem from the state stored on the user-specified I/O file by subprogram DSLESV.	
Usage	VECLIB: <pre> INTEGER*4 svfile, ier REAL*8 global(150) CALL DSLERS(svfile, global, ier) </pre>	
Input	svfile	Fortran logical unit number of a file containing the saved problem state as created by DSLESV.
Output	global	Global communications array restored from svfile .
	ier	Status response:
	ier = 0	Normal return.
	ier = -900	Failed to allocate systems.
	ier = -901	Error in dynamic storage allocation.
	ier = -902	I/O error detected by Fortran.
Example	Restart the solution process from the save file on Fortran logical unit 42 created by subroutine DSLESV. <pre> REAL*8 GLOBAL(150) CALL DSLERS (42,GLOBAL,IER) IF (IER .NE. 0) THEN handle error condition ENDIF </pre>	

Name	DSLESR Retrieve Runtime Statistics	
Purpose	This subprogram retrieves certain runtime statistics from the global communications array.	
Usage	VECLIB: INTEGER*4 inuse, mxused REAL*8 global(150), time(6), opcnts(2) CALL DSLESR(global, inuse, mxused, time, opcnts)	
Input	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	inuse	Amount of dynamically-allocated storage currently in use in units of REAL*8 words.
	mxused	Maximum amount of dynamically-allocated storage allocated in units of REAL*8 words.
	time	Array which, on return, holds the CPU time spent in matrix structure input, ordering, symbolic factorization, value input, numeric factorization, and numeric solution, in seconds.
	opcnts	Array which, on return, holds the operation counts for the numeric factorization and numeric solution.
Notes	The time reported for numeric solution is the time for all right-hand sides. If a phase has been repeated, time for the last repetition is reported. Time for structure and value input includes all the time, including the user's, from the start of input until it is completed.	
Example	Retrieve the statistics after the package has completed a numeric solution (either after DSLESL or DSLEFS).	

```

INTEGER*4  INUSE, MXUSED, OPCNTS (2)
REAL*8     GLOBAL (150), TIME (6)
CALL DSLESR (GLOBAL, INUSE, MXUSED, TIME, OPCNTS)

```

Name	DSLESV Save Problem State to a Savefile	
Purpose	This subprogram saves the current problem for restarting later at the current state. The global communication array and all working storage are written onto the user-specified I/O file using standard Fortran unformatted sequential write statements. The file is rewound before and after use.	
Usage	VECLIB: <pre> INTEGER*4 svfile, ier REAL*8 global(150) CALL DSLESV(svfile, global, ier) </pre>	
Input	svfile	Fortran logical unit number of the file onto which the state is to be saved. The file will be rewound before and after use.
	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: <pre> ier = 0 Normal return. ier = -902 I/O error detected by Fortran. </pre>
Example	Save the current state on Fortran logical unit 42. <pre> REAL*8 GLOBAL(150) CALL DSLESV (42,GLOBAL,IER) IF (IER .NE. 0) THEN handle error condition ENDIF </pre>	

7 Sparse Eigenvalues and Eigenvectors

Overview

This chapter describes state-of-the-art software for solving sparse symmetric and generalized symmetric eigenvalue problems. This package of subprograms provides efficient use of the Hewlett-Packard architecture in conjunction with powerful techniques for using the sparsity of the problem to reduce the cost of solution. Accuracy is assured through appropriate numerical techniques.

This chapter explains how to use sparse eigenvalue subprograms to solve sparse eigenvalue problems where the matrix or matrices are symmetric. Subprograms are provided to:

- Solve ordinary symmetric sparse eigenvalue problems of the form $Ax=\lambda x$
- Solve certain generalized symmetric sparse eigenvalue problems of the form $Ax=\lambda Bx$

This sparse matrix eigenvalue software is designed so that it is possible to call a single subprogram to solve a sparse symmetric eigenvalue problem. However, this requires a particular format for the sparse matrix. This package provides other approaches that provide a very general interface to alternative representations of the sparse matrix. These optional approaches, however, require the user to call a sequence of subprograms. This is similar to, but significantly more elaborate than, the use of EISPACK as described in Chapter 5.

This chapter is valid only for Hewlett-Packard C Series and Exemplar architectures. The subprograms described herein are not supported on Hewlett-Packard workstations.

The following documents provide supplemental material for this chapter:

Grimes, R.G., J.G. Lewis and H.D. Simon. "Eigenvalue Problems and Algorithms in Structural Engineering." *Large Scale Eigenvalue Problems*. North-Holland. 1986. pp. 81-95

Grimes, R.G., J.G. Lewis and H.D. Simon. "The Implementation of a Block Shifted and Inverted Lanczos Algorithm for Eigenvalue Problems in Structural Engineering." *Boeing Computer Services Technical Report, ETA-TR-39*. August, 1986.

Nour-Omid, B., B.N. Parlett, T. Ericsson and P.S. Jensen. "How to Implement the Spectral Transformation," *Mathematics of Computation*. 1987. 48, 178, pp. 663-673.

Parlett, B.N. *The Symmetric Eigenvalue Problem*. Englewood Cliffs, NJ: Prentice-Hall, Inc. 1980.

Chapter Objectives

After you read this chapter you will:

- Understand what a sparse matrix is
- Understand how to use these subprograms to compute some of the eigenvalues and eigenvectors of sparse symmetric matrices

What You Need to Know to Use These Subprograms

Generalized Symmetric Eigenproblems

This package is capable of solving ordinary symmetric eigenvalue problems $Ax=\lambda x$ and certain generalized symmetric eigenvalue problems $Ax=\lambda Bx$. In the latter case, both A and B are symmetric. The ordinary symmetric eigenproblem has the property that all eigenvalues are real and that there are n real orthogonal real eigenvectors. This latter condition is equivalent to $x_i^T x_i = 1$ and $x_i^T x_j = 0$ for all eigenvectors x_i and x_j with $i \neq j$.

To have similar properties in the generalized case, more conditions are needed than just symmetry. When B is singular, there are fewer than n finite eigenvalues. If, however, there exists a positive definite matrix $C=\alpha A+\beta B$ for some scalars α and β , then all finite eigenvalues are real and the corresponding eigenvectors are real and C -orthogonal. That is, $x_i^T C x_i = 1$ and $x_i^T C x_j = 0$ for all eigenvectors x_i and x_j with $i \neq j$.

Unfortunately, there is no general and sparse algorithm for determining the necessary scalars α and β for a general pair of matrices A and B . The transformations used in this package require explicit knowledge of C , so this

package is restricted to cases where the user can explicitly transform the problem to be in terms of C , or to the two standard cases where C is obvious; that is, when B or A alone is a positive definite matrix. The standard names associated with these cases in structural engineering are:

- Vibration analysis — B is positive definite ($A=K$, the stiffness matrix; $B=M$, the mass matrix)
- Buckling analysis — A is positive definite and B is indefinite ($A=K$, the stiffness matrix; $B=K_g$, the geometric or differential stiffness matrix)

In both cases, the package allows the matrix subject to the definiteness condition to be a semi-definite matrix. For example, the mass matrix in vibration analysis is required only to have non-negative masses, not positive masses.

Allowing semi-definiteness makes the package more generally useful, but also allows the user to specify a problem that is not a well-posed generalized symmetric eigenproblem. In most contexts, the user will know from scientific considerations that this is or is not a problem. Well-posed vibration problems with singular mass matrices are quite common. Normally, the package will return warnings about inconsistent inertia counts in ill-posed cases. However, supplying an indefinite matrix (for example, a mass matrix with negative masses) will usually result in a fatal message describing a failure in the reorthogonalization modules or in inconsistent inertia counts.

Sparsity and Storage Formats

Sparse matrices are matrices in which most of the entries are zero. The goal of sparse matrix software is to take maximum advantage of these zero entries to reduce storage and arithmetic. Storage is reduced by not storing zero entries; arithmetic is reduced by not performing operations on entries that are known to be zero.

It is easiest to see how to economize on storage. Suppose that an n -by- n matrix A has only nz nonzero entries. Then A could be specified completely by storing each of the nonzero values in an array of length nz that was accompanied by two integer arrays of length nz holding the corresponding row and column indices. Thus, $3nz$ storage suffices where n^2 storage is required for the corresponding dense matrix format. Consider, for example, the following matrix:

11	0	13	14	0	0
0	22	23	0	25	0
31	32	33	0	35	0
41	0	0	44	45	0
0	52	53	54	55	0
0	0	0	0	0	66

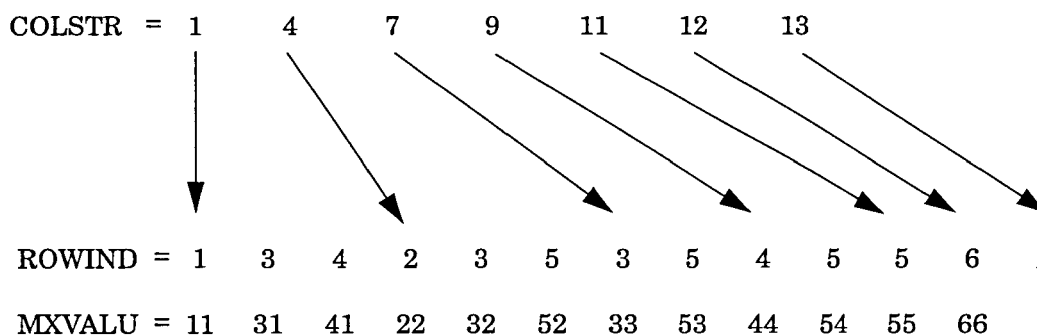
This matrix could be represented in the format described above by three arrays, IROW, JCOL, and MXVALU, as shown in Figure 7-1.

Figure 7-1 Row and Column Index Sparse Matrix Representation

IROW	=	1	3	4	2	3	5	1	2	3	5	1	4	5	2	3	4	5	6
JCOL	=	1	1	1	2	2	2	3	3	3	3	4	4	4	5	5	5	5	6
MXVALU	=	11	31	41	22	32	52	13	23	33	53	14	44	54	25	35	45	55	66

In this example, the matrix entries are listed in order within each column, and the columns are listed in order, although the representation does not require that much structure. However, if the entries are required to be ordered by row and column, even less storage is needed. In addition, symmetry in the matrix can be used by storing only the entries, for example, on or below the main diagonal. Other contexts, such as finite element analysis, can make even more concise representations of the locations of the nonzeros.

This package adopts a particular internal format that allows arbitrary symmetric matrices to be stored in $2nz+n+1$ storage locations, where nz represents the number of nonzeros in the lower triangle of the matrix. In essence, the JCOL array, which has repeated entries, is replaced by a shorter array that gives the index of the first nonzero of each column in the MXVALU array and an indication of the number of elements in each column. These two pieces of information per matrix column can be represented in a single array COLSTR of length $n+1$ if the convention is adopted that the number of nonzeros in column j is given by $\text{COLSTR}(j+1) - \text{COLSTR}(j)$ and if $\text{COLSTR}(n+1)$ is set accordingly. This sparse matrix format, known as the *column pointer, row index* representation, is illustrated in Figure 7-2.

Figure 7-2 Column Pointer, Row Index Sparse Matrix Representation

There are three ways of communicating the coefficient matrix or matrices to the package. One is a totally general form, which allows the user to store the matrices outside the package in whatever form is most convenient. The other two ways require that the user store the matrices in a form similar to the internal format or at least with all entries in each column contiguous in memory. Any of these three can be used. However, the most general form carries additional overhead in computer time.

Description of Sparse Eigenvalue Problems

Sparsity in the matrix does not guarantee sparsity in the matrix of eigenvectors. Indeed, it is rare that any eigenvectors of a sparse matrix are sparse. This implies that computing all the eigenvalues and eigenvectors of a sparse matrix is an enormous computation requiring storage of a large dense matrix. The common requirement for sparse eigenanalysis is to compute a subset, usually small, of the eigenvalues and vectors. This section describes how to specify the appropriate subsets to the sparse eigenanalysis package.

For notational purposes, assume that the eigenvalues λ_i are ordered $\lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_n$. The results from subroutines in this section normally form a contiguous set of eigenvalues, say, $\lambda_i, \lambda_{i+1}, \dots, \lambda_j$. The user must designate which subset is wanted through the following kinds of constraints, which usually relate to the application-specific meaning of the eigenvalues:

1. Eigenvalues can be restricted to lie in a finite or semi-infinite interval (find only eigenvalues in $[a, b]$).
2. Eigenvalues can be described by count (find only P eigenvalues) together with a location descriptor based either on the magnitude (either the smallest/lowest or largest/highest) or on the proximity to a fixed value (closest to c).

These two characterizations can be used together or separately. Examples of standard cases include:

- Find the lowest 10 eigenvalues
- Find all eigenvalues in [0.0, 100.0]
- Find the 3 eigenvalues closest to 53.1

This structure also permits combinations such as:

- Find the lowest 15 eigenvalues in [20.1, 75.3]
- Find the 10 eigenvalues closest to 15.0 in [10.0, 20.0]

The terms 'lowest' and 'highest' have two standard and different meanings in algebra. One is size (for example, least magnitude), the other is ordinal position (algebraically least). These subprograms follow a common engineering usage, where the interpretation is magnitude, not order. Thus, lowest means closest to zero and highest farthest from zero. The two interpretations are the same when all eigenvalues are positive and the reverse of one another when all eigenvalues are negative. The greatest confusion arises when the eigenvalues of interest may include both positive and negative numbers, that is, when the interval specifications include zero in the interior of the interval. To help clarify the interpretation of 'lowest' in an interval containing zero in the interior, printed output from the package will describe this as 'nearest' to zero.

When zero lies in the interior of the interval, the eigenvalues highest in magnitude may be drawn from both the algebraically least and the algebraically greatest. These eigenvalues could form two disjoint sets of contiguous eigenvalues rather than one. To avoid problems associated with this, the subprograms currently allow finding the highest eigenvalues only with an interval specification [a,b] that lies to one side of zero (for example, [1.0,100.0], [-9.99,-1.0] or [0.0, 25.0], but not [-100.0,100.0]). This restriction on the interval does not apply to the other descriptors (all, lowest, or nearest).

When you want the algebraically least or greatest eigenvalues, the problem must be transformed into finding the eigenvalues nearest one or the other endpoint of the interval.

Trust Regions and Matrix Inertias (Sturm Sequence Counts)

This sparse eigenanalysis package includes an independent facility for verifying that the eigenvalues computed are the eigenvalues requested. To know if a given set of 10 eigenvalues comprise the least 10 eigenvalues of a matrix, it is necessary either to compute all of the eigenvalues or to obtain information on eigenvalue counts and location through some other means.

This package uses the fact that if $A - \sigma I = LDL^T$, then the number of negative eigenvalues of D is the same as the number of eigenvalues of A that are smaller

than σ . This eigenvalue package uses the sparse symmetric linear equation package to compute an LDL^T factorization, where D is a diagonal or quasi-diagonal matrix.

It is trivial to count the number of eigenvalues of each sign for D . (Technically, the three numbers giving the number of negative, zero and positive eigenvalues of D are the inertia of D and $A - \sigma I$. A related technique for tridiagonal matrices yielding so-called Sturm sequences can be extended to give the same information; this name has been carried over in certain engineering contexts to describe the inertia of the matrix.)

The matrix inertias are used in the following way: The difference between the number of negative eigenvalues for the associated D matrices for $A - \sigma_2 I$ and for $A - \sigma_1 I$ is the number of eigenvalues in the interval $[\sigma_1, \sigma_2]$. If this number agrees with the number of eigenvalues that the package has computed in this interval, all of the eigenvalues in this subinterval have been computed. We call such a subinterval a *trust region*. The goal of the package is to compute the number of eigenvalues requested by the user and to find a trust region including these eigenvalues. To this end, whenever factorizations are computed, the inertias of the associated matrix are evaluated and saved. Additional factorizations may be computed to provide trust region information, so that all returned eigenvalues are confirmed as being properly described.

This use of matrix inertia is subject to rounding errors. A particular difficulty arises when the shift value σ is actually equal to an eigenvalue. In this case $A - \sigma I$ should have at least one zero eigenvalue. In practice, the numerical value is probably very small, but nonzero. However, the matrix inertia considers only the sign, and so an incorrect count may be returned even when the factorization is stable. The eigenvalue location counts derived from the inertias may differ from what has been computed. In such cases a warning is given to the user. The algorithm attempts to avoid such situations when creating trust regions but often must use user-specified interval bounds for its final trust region. Specifying eigenvalues in an interval where the endpoints are very close to eigenvalues is likely to cause warnings.

Convention for Returning Eigenvalues and Eigenvectors

The number of eigenvalues and vectors that will be computed by the sparse eigenanalysis package is often not known before the computation. This is obviously the case when all of the eigenvalues in an interval are requested, but it may also occur when an interval contains fewer than the number of eigenvalues requested. The eigenvector matrix is presumed to be dense, so it would occupy a large amount of memory if the number of eigenvalues computed is large. In some contexts, it may not be necessary or desirable to hold the entire eigenvector matrix in main memory. Thus, rather than request that the user supply the space for the eigenvector matrix, the eigenvectors are returned to the user through a two-pass process. The user first calls one or more

subprograms to compute the eigenvalues and eigenvectors. These inform the user of the number computed. In the second pass, the user can request that either specified single or groups of eigenvectors be returned, permitting more flexible use of storage.

This convention, requiring a second subroutine to retrieve the eigenvectors, also permits the user to have selective access to additional data that would otherwise be discarded by the algorithm. It is common for the package to compute a few more eigenvectors than the user requested. For example, if the user requests the lowest 10 eigenvalues and the tenth, eleventh and twelfth eigenvalues are very close to one another, the package will have to compute all twelve eigenvalues in order to establish a reliable trust region containing the first 10 eigenvalues. There are other contexts, as when the heuristic shifting strategy used is not perfect, or when some failure of the algorithm causes the computation to be cut short, where the package will compute more or different eigenvalues and vectors than requested.

The package allows the retrieval of these otherwise discarded eigenvectors if the user wants them. The main eigenanalysis subprogram returns two values: the number of eigenvalues and vectors computed to the specified accuracy that lie in a single reliable trust region and the number of eigenvalues and vectors computed to the specified accuracy that do not lie in this single reliable trust region. These additional eigenvalues may or may not lie in a trust region that is an extension of the final trust region. To determine whether this is the case, the user will have to make an interpretation of the matrix inertia data, which is printed (in English) for each factorization if the output message level is set to one or greater.

Global Communications Array

All of the subprograms in this package use dynamic memory allocation capabilities (refer to Chapter 12) to free users from the often difficult issue of allocating storage for the factors of the matrix. This internal storage is invisible to the user. When the sophisticated lower-level routines are used, knowledge of the internal storage is passed from subprogram to subprogram through a single communications array. This array, called **global** in each of the calling sequences, is a fixed-length double precision vector of length 150. It must not be altered by the user, as it represents the essential knowledge of the problem. Because it is the only identification for a problem, the user can handle multiple problems simultaneously by having multiple communications arrays with different names.

Error Convention

Each subprogram has an error return flag, **ier**, as one of its arguments. A zero value returned in **ier** is the indication of successful processing. Fatal errors are signaled through negative values; each is a negative integer in the range -1000

$\leq \text{ier} \leq -100$. The hundreds digit indicates the phase of processing in which the error occurred; the other digits specify the error itself. A positive error return indicates that some number of eigenvalues have been computed but a fatal error was encountered which caused early termination of the eigensolution phase.

In addition to the error return flag, the eigensolution phase has an additional warning return, **warning**, to indicate when the computed results are not exactly what was requested by the user. A warning will occur, for example, when the user requested the lowest 20 eigenvalues in a finite interval which contains only 11 eigenvalues.

Output Controls

This package differs from most library subroutines in providing optional printed or printable output. The amount of output is controlled by an integer variable, **msglvl**, specifying the message level. Setting **msglvl** ≤ 0 suppresses all printed messages, including error messages, and thus should generally be avoided. However, this will be the proper message level when the contents of the output file must originate solely from the application program that is using the sparse matrix package. When **msglvl** = 1, a small amount of runtime statistics, including matrix inertias necessary for interpreting 'discardable' results, and any error messages will be printed. When **msglvl** > 1 , the complete set of runtime statistics will be printed. If **msglvl** ≥ 3 , various arrays whose length is on the order of the number of equations will be printed. If **msglvl** ≥ 4 , volumes of output will be produced for debugging purposes.

We recommend that, as a rule, you set **msglvl** = 1. The higher **msglvl** values (≥ 3) are intended for debugging purposes and generate copious amounts of printed output. Further, their use for this purpose may require some knowledge of the data structures being used by the package.

Paths of Control

This package provides both a single standard format for sparse matrices and a completely general interface that allows the user to represent the sparse matrices in the most convenient form for use outside this package. The possible paths through the package are illustrated in Figure 7-3.

What You Need to Know to Use These Subprograms

Figure 7-3 **Paths of Control**

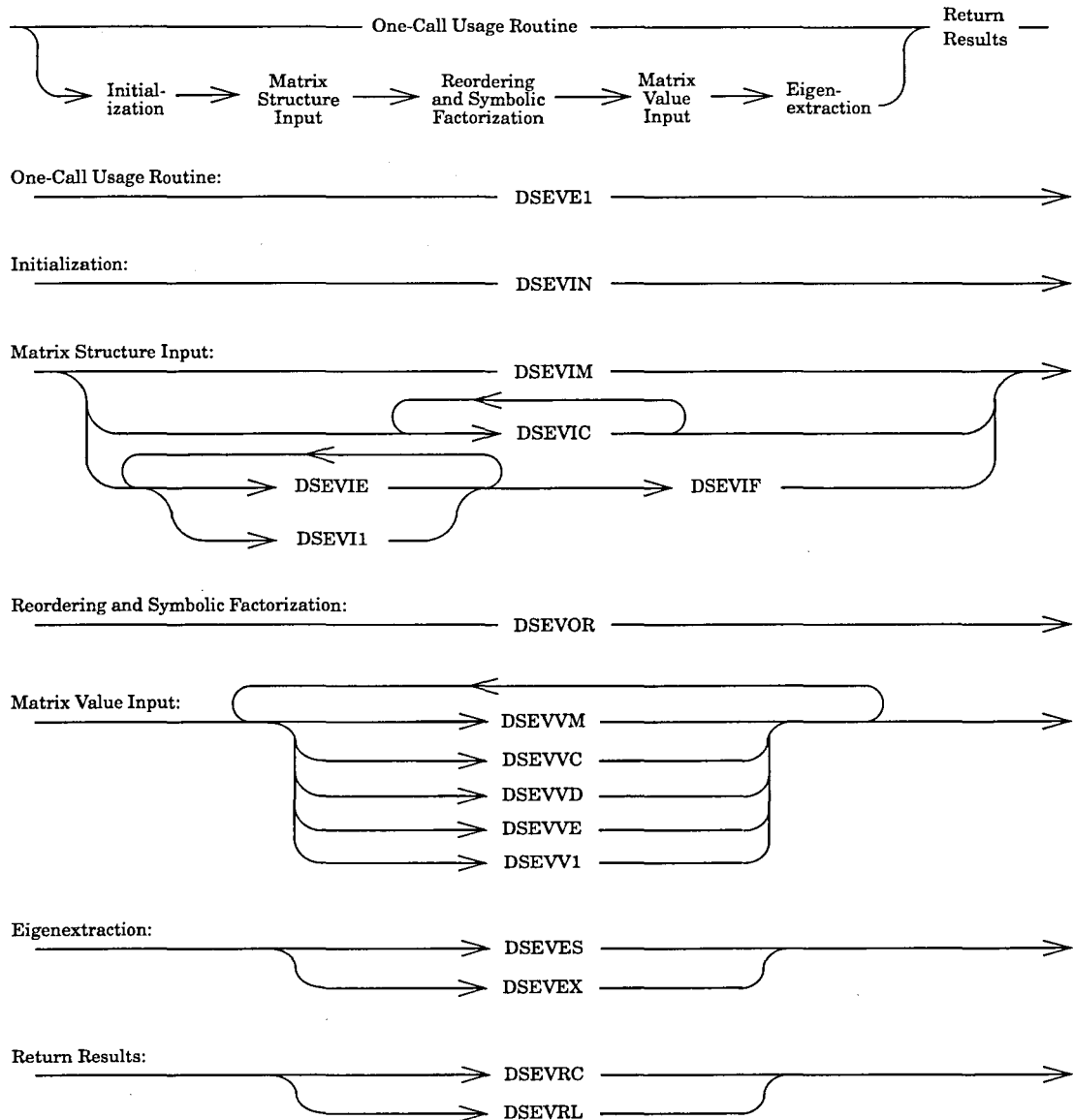
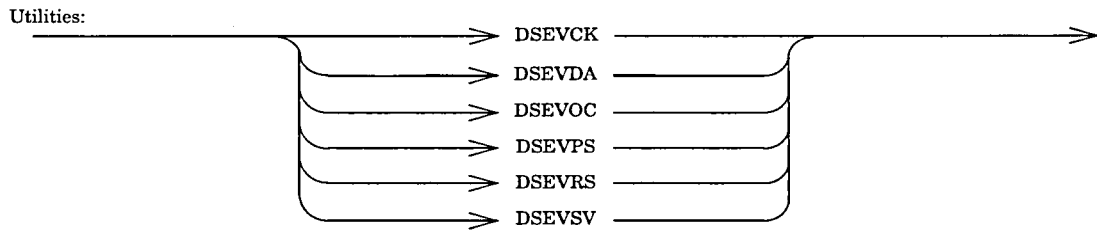


Figure 7-4 Paths of Control (continued)

To use the general interface, the user must call a sequence of subprograms that perform the following operations.

- Initialization
- Input of matrix structure
- Reordering
- Input of values of matrix entries
- Eigenextraction
- Retrieval of eigenvalues and vectors

It is possible to use this package to solve several different eigenvalue problems associated with the same matrix. This uncommon situation can be handled by calling the eigenextraction subprogram with different specifications for the desired eigenvalues. Unlike the sparse linear equations package described in Chapter 6, the only mechanism for solving eigenvalue problems for a sequence of matrices with different values, but the same sparsity pattern, is to use the save and restart facilities immediately after the reordering phase.

Sample Program

As illustrated in Figure 7-3, there are many possible paths of control through this sparse eigenvalue package. The following sample program is provided to show one possible path through the package, where the problem to be solved is the ordinary eigenvalue problem $Ax = \lambda x$. It is intended to demonstrate that the package is not as difficult to use as Figure 7-3 implies.

Sample Program

In this example, the row and column indices and the corresponding value for each nonzero entry of the matrix are stored in the three arrays IROW, JCOL, and MXVALU. This example demonstrates the use of the subroutines for solving a sparse eigenproblem when the subroutine DSEVES cannot be used. The eigenproblem posed here is to find the 10 lowest eigenvalues of the matrix A represented by the three arrays. Hence, this is an ordinary eigenproblem. The code assumes that there are NNZERO nonzero entries in the matrix, which has NEQNS rows and columns.

```

      INTEGER*4 IROW(NNZERO),JCOL(NNZERO),WARNNG
      LOGICAL*4 ORTWRN
      REAL*8 DISCRP,DUMMY,VALUE
      REAL*8 MXVALU(NNZERO),GLOBAL(150),EVALUE(10),EVECTR(NEQNS,10)

C
C   ... INPUT THE MATRIX STRUCTURE
C

      CALL DSEVIN (NEQNS,'I',1,6,GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000

      DO 100 K = 1, NNZERO
        I = IROW(K)
        J = JCOL(K)
        CALL DSEV11 ('A',I,J,GLOBAL,IER)
        IF ( IER .NE. 0 ) GO TO 8000
100 CONTINUE

      CALL DSEVIF (GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000

C
C   ... REORDER THE MATRIX
C

      CALL DSEVOR (GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000

C
C   -
C   ... INPUT MATRIX VALUES
C   -

      DO 200 K = 1, NNZERO
        I = IROW(K)
        J = JCOL(K)
        VALUE = MXVALU(K)
        CALL DSEVV1 ('A',I,J,VALUE,GLOBAL,IER)
        IF ( IER .NE. 0 ) GO TO 8000
200 CONTINUE

C
C   ... COMPUTE THE EIGENVALUES AND EIGENVECTORS
C

      CALL DSEVES (10,'L','O',.FALSE.,DUMMY,.FALSE.,DUMMY,
1      DUMMY,NFOUND,NDISCD,GLOBAL,IER,WARNNG)
      IF ( IER .NE. 0 ) GO TO 8000

C
C   ... CHECK ACCURACY
C

      CALL DSEVCK (.TRUE.,ORTWRN,DISCRP,GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000
      IF ( ORTWRN ) PRINT *, 'EIGENVALUE/EIGENVECTOR DISCREPANCY
      =',DISCRP

```

Subprograms Included in This Chapter

```
C      -  
C      ... RETRIEVE EIGENVALUES AND EIGENVECTORS  
C      -  
C      CALL DSEVRC (20,EVALUE,1,NFOUND,EVECTR,NEQNS,GLOBAL,IER)  
C      IF ( IER .NE. 0 ) GO TO 8000  
  
C      ... USE THE EIGENVALUES AND EIGENVECTORS  
C      .  
C      .  
C      .  
C      ... ERROR TRAP  
C  
8000  .....
```

Subprograms Included in This Chapter

Following are the subprograms included with VECLIB for solving sparse symmetric and generalized symmetric eigenvalue problems.

Name	DSEVE1 One Call Usage	
Purpose	This subprogram provides a one-call usage for the sparse eigenvalue package. If this subroutine does not fit your needs, a more flexible interface is available by using the other subroutines in the package described in this chapter.	
Usage	VECLIB: CHARACTER*1 bmxtyp, which, pbtype INTEGER*4 annzer, bnnzer, norder, msglvl, output, acolst(norder+1), arowin(annzer), bcolst(norder+1), browin(bnnzer), neigvl, nfound, ndiscd, ier, warnng LOGICAL*4 lfnit, rfnit REAL*8 avalue(annzer), bvalue(bnnzer), lftend, rhtend, center, global(150) CALL DSEVE1(norder, msglvl, output, acolst, arowin, avalue, bmxtyp, bcolst, browin, bvalue, neigvl, which, pbtype, lfnit, lftend, rfnit, rhtend, center, nfound, ndiscd, global, ier, warnng)	
Input	norder	Order of matrices (number of degrees of freedom), norder > 0.
	msglvl	Message level for printable output: msglvl ≤ 0 Suppress all output. msglvl = 1 Error messages, summary statistics and inertia information. msglvl = 2 More complete statistics. msglvl = 3 First stage of debugging output. msglvl = 4 Complete debugging output.
	output	Fortran logical unit number for file to which printable output will be written.
	acolst	acolst(j) gives the address in arowin of the first nonzero in the lower triangular part of column j of the matrix A. All of the nonzeros for column j are found in ascending order in arowin(acolst(j)), arowin(acolst(j)+1), ..., arowin(acolst(j+1)-1). acolst(norder+1) must be set to one greater than the total number of nonzeros, annzer, in the lower triangular part of the matrix.

arowin	List of row indices for all nonzeros, in ascending order within each column, in the lower triangle part of the matrix A .
avalue	List of values corresponding in position to the indices in arowin .
bmxtyp	A character string specifying the type of sparsity found in the right-hand-side B matrix in a generalized eigenvalue problem $Ax = \lambda Bx$. Only the first character is significant, but longer input can be used for clarity (for example, 'Identity' or 'Diagonal'). 'I' or 'i' or '1' B is an identity matrix. Use this option for a standard (ordinary) eigenproblem $Ax = \lambda x$, where there is no second matrix. 'A' or 'a' or 'K' or 'k' The sparsity structure of B is the same as that of A or K. 'D' or 'd' B is a diagonal matrix but not necessarily the identity. Any other letter B is a general sparse matrix whose structure is specified independently from A.
bcolst	If B is specified as a general sparse matrix with structure different than A, then bcolst(j) gives the ordinal in browin of the first nonzero in the lower triangular part of column j of the matrix B. All of the nonzeros for column j are found in ascending order in browin(bcolst(j)), browin(bcolst(j)+1), ..., browin(bcolst(j)+1)-1). bcolst(norder+1) must be set to one greater than the total number of nonzeros, bnnzer in the lower triangular part of the matrix. Not referenced unless the matrix B is specified as anything other than a general sparse matrix with structure different from A.
browin	List of row indices for all nonzeros in ascending order within each column in the lower triangle part of the matrix B .

	Not referenced unless the matrix B is specified as anything other than a general sparse matrix with structure different from A .
bvalue	If B is a general sparse matrix, then bvalue contains the list of values corresponding in position to the indices in arowin or browin depending on whether B has the same structure as A or not. If B is diagonal, then bvalue contains the diagonal entries. Not referenced if B is the identity.
neigvl	Number of eigenvalues to be found, neigvl > 0.
which	Character string indicating which eigenvalues are to be computed. Only the first character is significant, but longer input can be used for clarity (e.g., 'Lowest' or 'All').
	'L' or 'l' The lowest (smallest magnitude) eigenvalues.
	'H' or 'h' The highest (greatest magnitude) eigenvalues.
	'C' or 'c' or 'N' or 'n' The eigenvalues nearest center .
	'A' or 'a' All eigenvalues in the specified interval.
pdtype	Character string indicating type of problem. Only the first character is significant, but longer input can be used for clarity (e.g., 'Vibration' or 'Ordinary').
	'V' or 'v' Generalized symmetric (vibration) problem $Kx = \lambda Mx$, with M positive semi-definite.
	'B' or 'b' Generalized symmetric (buckling) problem $Kx = \lambda K_g x$, with K positive semi-definite and K_g possibly indefinite.
	'O' or 'o' Ordinary symmetric eigenproblem $Kx = \lambda x$.
lfinit	If .TRUE. , the value of the argument lftend is to be used as a restriction on the location of computed eigenvalues. If .FALSE. , no lower bound is placed on the value of computed eigenvalues; equivalently, the left endpoint is negative infinity.

	lftend	Left endpoint of an interval in which the desired eigenvalues lie; not used if lfinit is .FALSE.
	rfinit	If .TRUE. , the value of the argument rhtend is to be used as a restriction on the location of computed eigenvalues. If .FALSE. , no upper bound is placed on the value of computed eigenvalues; equivalently, the right endpoint is positive infinity.
	rhtend	Right endpoint of an interval in which the desired eigenvalues lie; not used if rfinit is .FALSE.
	center	A center or critical value for use when eigenvalues nearest some particular value are desired. Referenced only when which specifies 'centered' or 'nearest'.
Output	nfound	The number of eigenvalues computed and confirmed to meet the problem description constraints by inertia (Sturm sequence) computations.
	ndiscd	The number of other eigenvalues computed during the course of the computation, but discarded because they were outside of the specifications for eigenvalues or because they were not confirmed by inertia computations.
	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
	ier	Status response: ier = 0 Normal return. ier < 0 Fatal error, no useful results returned. ier > 0 Fatal error, but some eigenvalues and vectors computed. ier = -101 Error in dynamic storage allocation during matrix structure input. ier = -102 norder ≤ 0. ier = -201 Error in dynamic storage allocation during ordering. ier = -301 Error in dynamic storage allocation. ier = -302 Internal error. ier = -401 Error in dynamic storage allocation. ier = ±701 Error in storage allocation.

ier = ±702	Illegal specification for lftend and rhtend .
ier = ±703	Illegal specification for pbtype .
ier = ±704	Illegal specification for which .
ier = ±705	which = highest for an interval that spans zero.
ier = ±706	Insufficient dynamic memory for factorization.
ier = ±707	Internal error during factorization.
ier = ±710	Factorization save error in finite interval analysis.
ier = ±720	Error in storage allocation for first set of Lanczos recurrence vectors.
ier = ±721	Error in storage allocation for second set of Lanczos recurrence vectors.
ier = ±722	Error in storage allocation for eigenvectors.
ier = ±723	Insufficient storage during Lanczos iteration.
ier = ±724	QL iteration did not converge.
ier = ±725	Number of eigenvalues computed exceeded storage limitations.
ier = ±726	Internal error involving access of Lanczos recurrence vectors.
ier = ±727	Singular Value Decomposition did not converge.
ier = ±728	Gram-Schmidt reorthogonalization did not converge.
ier = ±729	Three numeric factorizations in a row failed. Probable cause is that this is not a symmetric generalized eigenproblem.
ier = ±730	Number of trust regions formed exceeded storage limitations.
ier = ±740	Error in deallocation of storage for first set of Lanczos recurrence vectors.

ier = ± 741 Error in deallocation of storage for second set of Lanczos recurrence vectors.

warnng Warning status. **warnng** includes several different warnings encoded in a three digit decimal integer:

low-order digit:

warnng = xx0 Normal return.

warnng = xx1 Fewer modes computed than requested (interval does not include requested number).

warnng = xx2 Fewer modes computed than requested because eigenvalues are wildly different in size.

warnng = xx3 Both of the above conditions occurred.
ten's digit:

warnng = x0x Normal return.

warnng = x1x At some point in the process, an inconsistent inertia count was found.

hundred's digit:

warnng = 0xx Normal return.

warnng = 1xx Interval had to be expanded because one or both endpoints were very close to eigenvalues.

Notes

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **potype** argument as 'vibration' or 'buckling'.

Example In a small (order 6) sparse generalized eigenvalue problem, the matrices A and B are as follows:

$$A = \begin{bmatrix} 4.0 & 0.0 & 1.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 4.0 & 1.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 1.0 & 4.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 4.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 1.0 & 1.0 & 4.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 1.0 \end{bmatrix}$$

$$B = \begin{bmatrix} 1.0 & 0.0 & 1.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 1.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 1.0 & 1.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 1.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 1.0 \end{bmatrix}$$

The eigenproblem is to compute the two eigenvalues nearest zero.

```
INTEGER*4  MSGVLVL, OUTPUT, ACOLST(7), AROWIN(13), IER
REAL*8     GLOBAL(150), AVALUE(13), BVALUE(13)
```

```
MSGVLVL = 1
OUTPUT = 6
```

```
ACOLST(1) = 1
ACOLST(2) = 4
ACOLST(3) = 7
ACOLST(4) = 9
ACOLST(5) = 12
ACOLST(6) = 13
ACOLST(7) = 14
```

```
AROWIN(1) = 1
AROWIN(2) = 3
AROWIN(3) = 4
AROWIN(4) = 2
AROWIN(5) = 3
AROWIN(6) = 5
AROWIN(7) = 3
AROWIN(8) = 5
AROWIN(9) = 4
AROWIN(10) = 5
AROWIN(11) = 6
AROWIN(12) = 5
AROWIN(13) = 6
```

```
AVALUE(1) = 4.0
AVALUE(2) = 1.0
AVALUE(3) = 1.0
```

```
AVALUE(4) = 4.0
AVALUE(5) = 1.0
AVALUE(6) = 1.0
AVALUE(7) = 4.0
AVALUE(8) = 1.0
AVALUE(9) = 4.0
AVALUE(10) = 1.0
AVALUE(11) = 1.0
AVALUE(12) = 4.0
AVALUE(13) = 1.0
```

```
BVALUE(1) = 1.0
BVALUE(2) = 1.0
BVALUE(3) = 1.0
BVALUE(4) = 1.0
BVALUE(5) = 1.0
BVALUE(6) = 1.0
BVALUE(7) = 1.0
BVALUE(8) = 1.0
BVALUE(9) = 1.0
BVALUE(10) = 1.0
BVALUE(11) = 1.0
BVALUE(12) = 1.0
BVALUE(13) = 1.0
```

```
CALL DSEVE1 (6,MSGLVL,OUTPUT,ACOLST,AROWIN,AVALUE,
             'A',IDUMMY,IDUMMY,BVALUE,2,'LOWEST',
             'VIBRATION',.FALSE.,-1.0D30,.FALSE.,1.0D30,0.0D0,
             NFOUND,NDISCD,GLOBAL,IER,WARNNG)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name	DSEVIN Initialize Sparse Eigenvalues/Eigenvectors	
Purpose	This subprogram provides information to start the sparse eigenvalue package.	
Usage	VECLIB: <pre> CHARACTER*1 bmxtyp INTEGER*4 norder, msglvl, output, ier REAL*8 global(150) CALL DSEVIN(norder, bmxtyp, msglvl, output, global, ier) </pre>	
Input	norder	Order of matrices (number of degrees of freedom), norder > 0.
	bmxtyp	A character string specifying the type of sparsity found in the right-hand-side B matrix in a generalized eigenvalue problem $Ax = \lambda Bx$. 'I' or 'i' or '1' B is an identity matrix. Use this option for a standard (ordinary) eigenproblem $Ax = \lambda x$, where there is no second matrix. 'A' or 'a' or 'K' or 'k' The sparsity structure of B is the same as that of A or K . 'D' or 'd' B is a diagonal matrix, but not necessarily the identity. Any other letter B is a general sparse matrix whose structure is specified independently from A .
	msglvl	Message level for printable output: msglvl ≤ 0 Suppress all output. msglvl = 1 Error messages, summary statistics and inertia information. msglvl = 2 More complete statistics. msglvl = 3 First stage of debugging output. msglvl = 4 Complete debugging output.
	output	Fortran logical unit number to which all printable output will be written.

Output	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.						
	ier	Status response: <table border="0" style="margin-left: 20px;"> <tr> <td>ier = 0</td> <td>Normal return.</td> </tr> <tr> <td>ier = -101</td> <td>Error in dynamic storage allocation.</td> </tr> <tr> <td>ier = -102</td> <td>norder not positive.</td> </tr> </table>	ier = 0	Normal return.	ier = -101	Error in dynamic storage allocation.	ier = -102	norder not positive.
ier = 0	Normal return.							
ier = -101	Error in dynamic storage allocation.							
ier = -102	norder not positive.							

Notes Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **bmxtyp** argument as 'identity' or 'diagonal'.

Example Prepare to compute some of the eigenvalues and eigenvectors of a 10,000 by 10,000 matrix (ordinary symmetric for eigenproblem). Obtain error messages and standard minimal output on Fortran logical unit 6.

```

INTEGER*4 NORDER, IER
REAL*8    GLOBAL(150)
NORDER = 10000
CALL DSEVIN (NORDER, 'IDENTITY', 1, 6, GLOBAL, IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF

```

Name	DSEVI1 Matrix Structure Input by Single Entry	
Purpose	This subprogram adds a single entry in the (irow , jcol) position in the lower triangle to the set of known nonzeros for one of the sparse matrices.	
Usage	VECLIB: <pre> CHARACTER*1 matrix INTEGER*4 irow, jcol, ier REAL*8 global(150) CALL DSEVI1(matrix, irow, jcol, global, ier) </pre>	
Input	matrix	Character string denoting the matrix for which the nonzero location is being input: 'A' or 'a' or 'K' or 'k' Add nonzero to the matrix on the left side. 'B' or 'b' or 'M' or 'm' Add nonzero to the matrix on the right side.
	irow	Row index of the nonzero entry, $jcol \leq irow \leq norder$, where norder is the matrix order specified in the call to DSEVIN. Input of diagonal entries is optional.
	jcol	Column index of the nonzero entry, $1 \leq jcol \leq norder$.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: ier = 0 Normal return. ier = -100 Incorrect processing path; DSEVIN not called or matrix input already finished. ier = -101 Error in dynamic storage allocation. ier = -104 Illegal value for jcol. ier = -105 Illegal value for irow.

ier = -109 ier = -110	Illegal value for matrix designator. Attempt to add nonzero location to <i>B</i> , which was not specified as having a general sparse structure (<i>B</i> is a diagonal matrix, an identity matrix, or has same structure as <i>A</i>).

Notes Calls to DSEVI1 and DSEVIE can be intermixed. DSEVIC and DSEVIM cannot be used if DSEVI1 or DSEVIE are used.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example Add the entry in row 3035, column 1024 to the list of nonzeros in the matrix *A*.

```

INTEGER*4 I,J,IER
REAL*8    GLOBAL(150)
I = 3035
J = 1024
CALL DSEVI1 ('A',I,J,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF

```


Name	DSEVIC Matrix Structure Input by Column	
Purpose	This subprogram adds a list of indices in a single column in the lower triangle to the set of known nonzeros for one of the sparse matrices.	
Usage	VECLIB: CHARACTER*1 matrix INTEGER*4 jcol, nzcol, jrowin(nzcol), ier REAL*8 global(150) CALL DSEVIC(matrix, jcol, nzcol, jrowin, global, ier)	
Input	matrix	Character string denoting the matrix for which the nonzero location is being input: 'A' or 'a' or 'K' or 'k' Add nonzero to the matrix on the left side. 'B' or 'b' or 'M' or 'm' Add nonzero to the matrix on the right side.
	jcol	Column index, $1 \leq \mathbf{jcol} \leq \mathbf{norder}$, where norder is the matrix order as specified in the call to DSEVIN. Columns must be presented in order from 1 to norder , except that columns with no entries can be skipped.
	nzcol	Number of nonzeros in column jcol of the matrix, $\mathbf{nzcol} \geq 0$.
	jrowin	List of row indices for all nonzeros, in ascending order, in the lower triangle part of column jcol of the matrix designated by matrix, jcol $\leq \mathbf{jrowin}(1) < \mathbf{jrowin}(2) < \dots < \mathbf{jrowin}(\mathbf{nzcol}) \leq \mathbf{norder}$. Input of diagonal entries is optional.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output	ier	Status response:
	ier = 0	Normal return.
	ier = -100	Incorrect processing path; DSEVIN not called or matrix input already finished.
	ier = -101	Error in dynamic storage allocation.
	ier = -103	Illegal value for nzcol .
	ier = -104	Illegal value for jcol ; either out of range or out of order.
	ier = -106	Illegal value for at least one entry in jrowin or entries out of order.
	ier = -109	Illegal value for matrix designator.
	ier = -110	Attempt to add nonzero location to <i>B</i> , which was not specified as having a general sparse structure (<i>B</i> is a diagonal matrix, an identity matrix, or has same structure as <i>A</i>).

Notes The entire matrix structure must be input with DSEVIC if it is used. Its use is not compatible with DSEVI1, DSEVIE, or DSEVIM.

If the matrix entries are available by column, using DSEVIC is more efficient than using DSEVI1.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example Column 4519 has entries in rows 1, 2735, 4519, 4520, 4521, 6000, 6002, and 6004. Add all five entries in the lower triangular part to the list of nonzeros in the matrix *A*. Columns 1 to 4518 already have been input to the package using DSEVIC.

```

INTEGER*4 J,NZCOL,JROWIN(100),IER
REAL*8    GLOBAL(150)
J = 4519
NZCOL = 5
JROWIN(1) = 4520
JROWIN(2) = 4521
JROWIN(3) = 6000
JROWIN(4) = 6002
JROWIN(5) = 6004
CALL DSEVIC ('A',J,NZCOL,JROWIN,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF

```

Name	DSEVIE Matrix Structure Input by Finite Element	
Purpose	This subprogram adds indices to the set of known nonzeros in the lower triangle of one of the sparse matrices corresponding to a finite element or clique.	
Usage	VECLIB: CHARACTER*1 matrix INTEGER*4 nnode, nodlst(nnode), ier REAL*8 global(150) CALL DSEVIE(matrix, nnode, nodlst, global, ier)	
Input	matrix	Character string denoting the matrix for which the nonzero location is being input: 'A' or 'a' or 'K' or 'k' Add nonzero to the matrix on the left side. 'B' or 'b' or 'M' or 'm' Add nonzero to the matrix on the right side.
	nnode	Number of nodes in the finite element or clique, $1 \leq \mathbf{nnode} \leq \mathbf{norder}$, where norder is the matrix order specified in the call to DSEVIN.
	nodlst	List of nodes in element or clique, $1 \leq \mathbf{nodlst}(i) \leq \mathbf{norder}$ for all i , and $\mathbf{nodlst}(i) \neq \mathbf{nodlst}(j)$ for all i and j with $i \neq j$. All pairs $(\mathbf{nodlst}(i), \mathbf{nodlst}(j))$ in the lower triangle are added to the sparsity structure of the matrix designated by matrix .
Updated	nodlst global	The order of the values in nodlst may be changed. Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: ier = 0 Normal return. ier = -100 Incorrect processing path; DSEVIN not called or matrix input already finished.

ier = -101	Error in dynamic storage allocation.
ier = -107	Illegal value for nnode .
ier = -108	Illegal value for at least one entry in nodlst .
ier = -109	Illegal value for matrix designator.
ier = -110	Attempt to add nonzero location to <i>B</i> , which was not specified as having a general sparse structure (<i>B</i> is a diagonal matrix, an identity matrix, or has same structure as <i>A</i>).

Notes Calls to DSEVIE can be intermixed with calls to DSEVI1. DSEVIC and DSEVIM cannot be used if DSEVI1 or DSEVIE are used.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example Rows and columns 345, 346, 347, and 989 form a small dense submatrix of *A*. Add positions consisting of all pairs of numbers from this set to the list of nonzeros in the matrix *A*.

```

INTEGER*4 NNODE,NODLST(10),IER
REAL*8    GLOBAL(150)
NNODE = 4
NODLST(1) = 345
NODLST(2) = 346
NODLST(3) = 347
NODLST(4) = 989
CALL DSEVIE ('A',NNODE,NODLST,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF

```

Name	DSEVIM Matrix Structure Input by Matrix	
Purpose	This subprogram specifies the locations of all of the nonzeros in the lower triangle of one of the matrices.	
Usage	VECLIB: <pre> CHARACTER*1 matrix INTEGER*4 norder, nnzero, colstr(norder+1), rowind(nnzero), ier REAL*8 global(150) CALL DSEVIM(matrix, colstr, rowind, global, ier) </pre>	
Input	matrix	Character string denoting the matrix for which the nonzero location is being input: 'A' or 'a' or 'K' or 'k' Add nonzero to the matrix on the left side. 'B' or 'b' or 'M' or 'm' Add nonzero to the matrix on the right side.
	colstr	colstr(j) gives the address in rowind of the first nonzero in the lower triangular part of column <i>j</i> of the matrix specified by matrix . All of the nonzeros for column <i>j</i> are found in ascending order in rowind(colstr(j)) , rowind(colstr(j)+1) , ..., rowind(colstr(j+1)-1) . colstr(norder+1) must be set to one greater than the total number of nonzeros, nnzero in the lower triangular part of the matrix, where norder is the matrix order as specified in the call to DSEVIN.
	rowind	List of row indices for all nonzeros in ascending order within each column in the lower triangle part of the matrix designated by matrix . Input of diagonal entries is optional.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output	ier	Status response:
	ier = 0	Normal return.
	ier = -100	Incorrect processing path; DSEVIN not called or matrix input already finished.
	ier = -101	Error in dynamic storage allocation.
	ier = -106	Illegal value for at least one entry in colstr (not increasing or negative) or invalid entries in rowind (entries out of order within a column, or out of the lower triangle).
	ier = -109	Illegal value for matrix designator.
	ier = -110	Attempt to add nonzero location to <i>B</i> , which was not specified as having a general sparse structure (<i>B</i> is a diagonal matrix, an identity matrix, or has same structure as <i>A</i>).

Notes This is the most efficient mechanism for specifying the nonzero structure, but the entire matrix structure for both matrices must be input with DSEVIM if it is used. Its use is not compatible with DSEVI1, DSEVIE, or DSEVIC.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example In a small (order 6) sparse generalized eigenvalue problem, B is a mass matrix with the following structure:

```

x  0  x  x  0  0
0  x  x  0  x  0
x  x  x  0  x  0
x  0  0  x  x  0
0  x  x  x  x  0
0  0  0  0  0  0

```

```

INTEGER*4 COLSTR(7),ROWIND(6),IER
REAL*8    GLOBAL(150)
COLSTR(1) = 1
COLSTR(2) = 3
COLSTR(3) = 5
COLSTR(4) = 6
COLSTR(5) = 7
COLSTR(6) = 7
COLSTR(7) = 7

ROWIND(1) = 3
ROWIND(2) = 4
ROWIND(3) = 3
ROWIND(4) = 5
ROWIND(5) = 5
ROWIND(6) = 5

CALL DSEVIM ('B',COLSTR,ROWIND,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF

```

Name	DSEVIF End of Matrix Structure Input	
Purpose	This subprogram indicates the end of structure input for matrices in the sparse eigenvalue problem. DSEVIF is used only if subprograms DSEVIN and/or DSEVIE were the mechanism by which the structure was input.	
Usage	VECLIB: <pre> INTEGER*4 ier REAL*8 global(150) CALL DSEVIF(global, ier) </pre>	
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: <pre> ier = 0 Normal return. ier = -100 Incorrect processing path; DSEVIN not called or matrix input already finished. ier = -101 Error in dynamic storage allocation. </pre>
Example	The nonzero structure of a pair of finite element matrices was passed to the sparse eigenvalue package using repeated calls to subroutine DSEVIE. Signal that no more nonzeros will be added to either matrix. <pre> INTEGER*4 IER REAL*8 GLOBAL(150) CALL DSEVIF (GLOBAL, IER) IF (IER .NE. 0) THEN handle error condition ENDIF </pre>	

- Name** DSEVOR
Reordering and Symbolic Factorization
- Purpose** This subprogram reorders the matrix whose pattern of nonzeros is given by the locations where either A or B is nonzero, such that an efficient sparse factorization of such a matrix can be obtained. DSEVOR then constructs data structures required for the sparse factorization.
- Usage** VECLIB:

```

      INTEGER*4      ier
      REAL*8         global(150)
      CALL DSEVOR(global, ier)

```
- Output**
- | | |
|---------------|--|
| global | Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package. |
| ier | Status response: |
| | ier = 0 Normal return. |
| | ier = -200 Incorrect processing path; structure input not completed, or value input subroutines already called. |
| | ier = -201 Error in dynamic storage allocation. |
| | ier = -301 Error in dynamic storage allocation. |
| | ier = -302 Illegal error. |
- Example** The structure of the sparse matrices in a sparse eigenvalue problem have been communicated to the eigenanalysis package using DSEVIN followed by DSEVIM or DSEVIC or the following: DSEVI1, DSEVIE and DSEVIF. The next step is the process to obtain good reordering for the factorizations that will be needed in the eigenextraction process.

```

      INTEGER*4 IER
      REAL*8    GLOBAL(150)
      CALL DSEVOR (GLOBAL, IER)
      IF ( IER .NE. 0 ) THEN
        handle error condition
      ENDIF

```

Name	DSEVV1 Matrix Value Input by Single Entry	
Purpose	This subprogram adds to the value of the entry in the (irow , jcol) position in the lower triangle of one of the sparse matrices.	
Usage	VECLIB: <pre> CHARACTER*1 matrix INTEGER*4 irow, jcol, ier REAL*8 value, global(150) CALL DSEVV1(matrix, irow, jcol, value, global, ier) </pre>	
Input	matrix	Character string denoting the matrix for which the nonzero location is being input: 'A' or 'a' or 'K' or 'k' Add to the matrix on the left side. 'B' or 'b' or 'M' or 'm' Add to the matrix on the right side.
	irow	Row index of the nonzero entry, $jcol \leq irow \leq norder$, where norder is the matrix order as specified in the call to DSEVIN.
	jcol	Column index of the nonzero entry, $1 \leq jcol \leq norder$.
	value	Numeric value that will be added to any previous values input for this location.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: ier = 0 Normal return. ier = -400 Incorrect processing path; DSEVOR not called or eigenextraction already begun. ier = -401 Error in dynamic storage allocation. ier = -402 Subscript pair (irow , jcol) is not in lower triangle of matrix. ier = -403 Illegal value for matrix designator.

ier = -404	Subscript pair (irow , jcol) was not specified in structure input. No room for value.
ier = -405	The <i>B</i> matrix was specified as diagonal. Cannot add off-diagonal nonzero value.
ier = -406	The <i>B</i> matrix was specified as an identity matrix. Cannot add nonzero value.

Notes Calls to DSEVV1, DSEVVC, DSEVVD, DSEVVE, and DSEVVM can be intermixed.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example Store the value 4.523×10^{-5} as the nonzero entry in row 3035, column 1024 of the matrix *A*.

```

      INTEGER*4 I,J,IER
      REAL*8    VALUE,GLOBAL(150)
      I = 3035
      J = 1024
      VALUE = 4.523D-5
      CALL DSEVV1 ('A',I,J,VALUE,GLOBAL,IER)
      IF ( IER .NE. 0 ) THEN
         handle error condition
      ENDIF

```

Name	DSEVVC Matrix Value Input by Column	
Purpose	This subprogram adds to the values of a list of nonzero entries in the lower triangle of a single column of one of the sparse matrices.	
Usage	VECLIB: <pre> CHARACTER*1 matrix INTEGER*4 jcol, nzcol, jrowin(nzcol), ier REAL*8 values(nzcol), global(150) CALL DSEVVC(matrix, jcol, nzcol, jrowin, values, global, ier) </pre>	
Input	matrix	Character string denoting the matrix for which the nonzero location is being input: 'A' or 'a' or 'K' or 'k' Add to the matrix on the left side. 'B' or 'b' or 'M' or 'm' Add to the matrix on the right side.
	jcol	Column index, $1 \leq \text{jcol} \leq \text{norder}$, where norder is the matrix order as specified in the call to DSEVIN.
	nzcol	Number of values in the list to be added to column jcol of the matrix, $\text{nzcol} > 0$.
	jrowin	List of row indices in ascending order for values to be added to the lower triangle part of column jcol of the matrix designated by matrix , $\text{jcol} \leq \text{jrowin}(1) < \text{jrowin}(2) < \dots < \text{jrowin}(\text{nzcol}) \leq \text{norder}$. It is not necessary that all entries in column jcol be included in jrowin .
	values	List of values corresponding to positions specified by jcol and jrowin .
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: ier = 0 Normal return. ier = -400 Incorrect processing path; DSEVOR not called or eigenextraction already begun.

ier = -401	Error in dynamic storage allocation.
ier = -402	Illegal value for either nzcol or jcol .
ier = -403	Illegal value for matrix designator.
ier = -404	At least one subscript pair (jrowin(j) , jcol) was not specified in structure input. No room for value.
ier = -405	The <i>B</i> matrix was specified as diagonal. Cannot add off-diagonal nonzero value.
ier = -406	The <i>B</i> matrix was specified as an identity matrix. Cannot add nonzero value.

Notes Calls to DSEVV1, DSEVVC, DSEVVD, DSEVVE and DSEVVM can be intermixed. If the matrix entries are available by column, using DSEVVC is more efficient than using DSEVV1.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example Column 4519 has entries in rows 1, 2735, 4519, 4520, 4521, 6000, 6002, and 6004. Add the value 1.0 to each of these positions in the matrix *A*.

```

INTEGER*4 J,NZCOL,JROWIN(100),IER
REAL*8    VALUES(100),GLOBAL(150)
J = 4519
NZCOL = 6

JROWIN(1) = 4519
JROWIN(2) = 4520
JROWIN(3) = 4521
JROWIN(4) = 6000
JROWIN(5) = 6002
JROWIN(6) = 6004

VALUES(1) = 1.0D0
VALUES(2) = 1.0D0
VALUES(3) = 1.0D0
VALUES(4) = 1.0D0
VALUES(5) = 1.0D0
VALUES(6) = 1.0D0

CALL DSEVVC ('A',J,NZCOL,JROWIN,VALUES,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF

```

Name	DSEVVD Matrix Value Input to Main Diagonal	
Purpose	This subprogram adds to the values of corresponding entries on the main diagonal of one of the sparse matrices.	
Usage	VECLIB: CHARACTER*1 matrix INTEGER*4 norder, ier REAL*8 values(norder), global(150) CALL DSEVVD(matrix, values, global, ier)	
Input	matrix	Character string denoting the matrix for which the nonzero location is being input: 'A' or 'a' or 'K' or 'k' Add to the matrix on the left side. 'B' or 'b' or 'M' or 'm' Add to the matrix on the right side.
	values	Array of values to be added to the corresponding entries on main diagonal of the matrix specified by matrix .
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: ier = 0 Normal return. ier = -400 Incorrect processing path; DSEVOR not called or eigenextraction already begun. ier = -401 Error in dynamic storage allocation. ier = -403 Illegal value for matrix designator. ier = -404 At least one subscript pair (jrowin(j),jcol) was not specified in structure input. No room for value. ier = -406 The <i>B</i> matrix was specified as an identity matrix. Cannot add nonzero value.

Notes Calls to DSEVV1, DSEVVC, DSEVVD, DSEVVE and DSEVVM can be intermixed. If the matrix entries of the main diagonal are available as a vector, using DSEVVD is more efficient than using DSEVV1.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example Create a main diagonal equal to 4.0 for *B*.

```
INTEGER*4 IER
REAL*8    VALUES(100),GLOBAL(150)

VALUES(1) = 4.0D0
VALUES(2) = 4.0D0
VALUES(3) = 4.0D0
VALUES(4) = 4.0D0
VALUES(5) = 4.0D0
VALUES(6) = 4.0D0

CALL DSEVVD ('B',VALUES,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name	DSEVVE Matrix Value Input by Finite Element	
Purpose	This subprogram adds to the values of a set of nonzero entries in the lower triangle of one of the sparse matrices corresponding to a finite element or clique.	
Usage	VECLIB: <pre> CHARACTER*1 matrix INTEGER*4 nnode, ldelmx, nodlst(nnode), ier REAL*8 elmmtx(ldelmx, nnode), global(150) CALL DSEVVE(matrix, nnode, nodlst, elmmtx, ldelmx, global, ier) </pre>	
Input	matrix	Character string denoting the matrix for which the nonzero location is being input: 'A' or 'a' or 'K' or 'k' Add to the matrix on the left side. 'B' or 'b' or 'M' or 'm' Add to the matrix on the right side.
	nnode	Number of nodes in the finite element or clique, $1 \leq \mathbf{nnode} \leq \mathbf{norder}$, where norder is the matrix order as specified in the call to DSEVIN.
	nodlst	List of nodes in element or clique, $1 \leq \mathbf{nodlst}(i) \leq \mathbf{norder}$ for all i , and $\mathbf{nodlst}(i) \neq \mathbf{nodlst}(j)$ for all i and j with $i \neq j$. Values for all pairs $(\mathbf{nodlst}(i), \mathbf{nodlst}(j))$ in the lower triangle are added to the values of the matrix designated by matrix .
	elmmtx	A two-dimensional array containing values to be added to the matrix. Only the lower triangle (including the diagonal) of elmmtx is referenced. The value in elmmtx (k, l) is added to the value in position $(\mathbf{nodlst}(k), \mathbf{nodlst}(l))$ in the sparse matrix.
	ldelmx	The leading dimension of array elmmtx as declared in the calling program unit, with $\mathbf{ldelmx} \geq \mathbf{nnode}$.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output	ier	Status response:
	ier = 0	Normal return.
	ier = -400	Incorrect processing path; DSEVOR not called or eigenextraction already begun.
	ier = -401	Error in dynamic storage allocation.
	ier = -402	Illegal value for either nnode or in nodlst .
	ier = -403	Illegal value for matrix designator.
	ier = -404	At least one subscript pair (nodlst(k) , nodlst(l)) was not specified in structure input. No room for value.
	ier = -405	The <i>B</i> matrix was specified as diagonal. Cannot add off-diagonal nonzero value.
	ier = -406	The <i>B</i> matrix was specified as an identity matrix. Cannot add nonzero value.

Notes Calls to DSEVV1, DSEVVC, DSEVVD, DSEVVE, and DSEVVM can be intermixed. If the matrix entries are available as a sum of elemental matrices, using DSEVVE is more efficient than using DSEVV1.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example Rows and columns 345, 346, 347, and 989 form a small dense submatrix of A. Add the value 1.0 to the values in the positions consisting of all pairs of numbers from this set to the list of nonzeros in matrix A.

```

      INTEGER*4 NNODE,NODLST(10),IER
      REAL*8     ELMMTX(10,10),GLOBAL(150)

      NNODE = 4

      NODLST(1) = 345
      NODLST(2) = 346
      NODLST(3) = 347
      NODLST(4) = 989

      DO 200 K = 1, NNODE
        DO 100 L = K, NNODE
          ELMMTX(K,L) = 1.0D0
100      CONTINUE
200 CONTINUE
      CALL DSEVVE ('A',NNODE,NODLST,ELMMTX,10,GLOBAL,IER)
      IF ( IER .NE. 0 ) THEN
        handle error condition
      ENDIF

```

Name	DSEVVM Matrix Value Input by Matrix	
Purpose	This subprogram adds values to all of the entries in the lower triangle of one of the sparse matrices.	
Usage	VECLIB: <pre> CHARACTER*1 matrix INTEGER*4 norder, nnzero, colstr(norder+1), rowind(nnzero), ier REAL*8 values(nnzero), global(150) CALL DSEVVM(matrix, colstr, rowind, values, global, ier) </pre>	
Input	matrix	Character string denoting the matrix for which the nonzero location is being input: 'A' or 'a' or 'K' or 'k' Add to the matrix on the left side. 'B' or 'b' or 'M' or 'm' Add to the matrix on the right side.
	colstr	colstr(j) gives the address in rowind of the first nonzero in the lower triangular part of column <i>j</i> of the matrix specified by matrix . All of the nonzeros for column <i>j</i> are found, in ascending order, in rowind(colstr(j)) , rowind(colstr(j)+1) , ..., rowind(colstr(j+1)-1) . colstr(norder+1) must be set to one greater than the total number of nonzeros, nnzero , in the lower triangular part of the matrix, where norder is the matrix order as specified in the call to DSEVIN.
	rowind	List of row indices for all nonzeros in ascending order within each column in the lower triangle part of the matrix designated by matrix . Diagonal entries may be present, but are not required.
	values	List of values corresponding in position to the indices in rowind . These values will be added to any values already present in the matrix specified by matrix .
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output	ier	Status response:
	ier = 0	Normal return.
	ier = -400	Incorrect processing path; DSEVOR not called or eigenextraction already begun.
	ier = -401	Error in dynamic storage allocation.
	ier = -402	Illegal value for either nnode or in nodlst .
	ier = -403	Illegal value for matrix designator.
	ier = -404	At least one subscript pair was not specified in structure input. No room for value.
	ier = -405	The B matrix was specified as diagonal. Cannot add off-diagonal nonzero value.
	ier = -406	The B matrix was specified as an identity matrix. Cannot add nonzero value.

Notes This is the most efficient mechanism for specifying nonzero values. Normally, DSEVVM is used in conjunction with DSEVIM. The matrix structure passed to DSEVVM must be identical to the structure passed to DSEVIM.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example In a small (order 6) sparse generalized eigenvalue problem, *B* is the following mass matrix:

4.0	0.0	1.0	1.0	0.0	0.0
0.0	4.0	1.0	0.0	1.0	0.0
1.0	1.0	4.0	0.0	1.0	0.0
1.0	0.0	0.0	4.0	1.0	0.0
0.0	1.0	1.0	1.0	4.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0

Use DSEVVM and DSEVVD to input this as *B*.

```

INTEGER*4 COLSTR(7),ROWIND(6),IER
REAL*8     VALUES(6),DIAGNL(6),GLOBAL(150)

COLSTR(1) = 1
COLSTR(2) = 3
COLSTR(3) = 5
COLSTR(4) = 6
COLSTR(5) = 7
COLSTR(6) = 7
COLSTR(7) = 7

ROWIND(1) = 3
ROWIND(2) = 4
ROWIND(3) = 3
ROWIND(4) = 5
ROWIND(5) = 6
ROWIND(6) = 6

VALUES(1) = 1.0
VALUES(2) = 1.0
VALUES(3) = 1.0
VALUES(4) = 1.0
VALUES(5) = 1.0
VALUES(6) = 1.0

CALL DSEVVM ('B',COLSTR,ROWIND,VALUES,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF

DIAGNL(1) = 4.0
DIAGNL(2) = 4.0
DIAGNL(3) = 4.0
DIAGNL(4) = 4.0
DIAGNL(5) = 4.0
DIAGNL(6) = 0.0
CALL DSEVVD ('B',DIAGNL,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF

```

Note: It would not have been necessary to use DSEVVD if the diagonal values had been held in the same value array as the rest of the sparse matrix.

Name	DSEVES Eigenextraction		
Purpose	This subprogram computes selected eigenvalues and eigenvectors of a sparse symmetric matrix A or of a sparse symmetric matrix pencil (A,B) . The matrix or matrices have already been processed through the structure input, reordering, and value input phases. This is the standard interface; subroutine DSEVEX provides additional control parameters.		
Usage	VECLIB: CHARACTER*1 which, ptype INTEGER*4 neigvl, nfound, ndiscd, ier, warnng LOGICAL*4 lfnit, rfinit REAL*8 lftend, rhtend, center, global(150) CALL DSEVES(neigvl, which, ptype, lfnit, lftend, rfinit, rhtend, center, nfound, ndiscd, global, ier, warnng)		
Input	neigvl	Number of eigenvalues to be found, neigvl > 0.	
	which	Character string indicating which eigenvalues are to be computed.	
		'L' or 'l'	The lowest (smallest magnitude) eigenvalues.
		'H' or 'h'	The highest (greatest magnitude) eigenvalues.
		'C' or 'c' or 'N' or 'n'	The eigenvalues nearest center .
		'A' or 'a'	All eigenvalues in the specified interval.
	ptype	Character string indicating type of problem.	
		'V' or 'v'	Generalized symmetric (vibration) problem $Kx = \lambda Mx$, with M positive semi-definite.
		'B' or 'b'	Generalized symmetric (buckling) problem $Kx = \lambda K_g x$, with K positive semi-definite and K_g possibly indefinite.
		'O' or 'o'	Ordinary symmetric eigenproblem $Kx = \lambda x$.
	lfnit	If .TRUE., the value of the argument lftend is to be used as a restriction on the location of computed	

		eigenvalues. If .FALSE. , no lower bound is placed on the value of computed eigenvalues; equivalently, the left endpoint is negative infinity.
	lftend	Left endpoint of an interval in which the desired eigenvalues lie; not used if lfinit is .FALSE.
	rfinit	If .TRUE. , the value of the argument rhtend is to be used as a restriction on the location of computed eigenvalues. If .FALSE. , no upper bound is placed on the value of computed eigenvalues; equivalently, the right endpoint is positive infinity.
	rhtend	Right endpoint of an interval in which the desired eigenvalues lie; not used if rfinit is .FALSE.
	center	A center or critical value for use when eigenvalues nearest some particular value are desired. Referenced only when which specifies 'centered' or 'nearest'.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	nfound	The number of eigenvalues computed and confirmed to meet the problem description constraints by inertia (Sturm sequence) computations.
	ndiscd	The number of other eigenvalues computed during the course of the computation but discarded because they were outside specifications for eigenvalues or because they were not confirmed by inertia computations.
	ier	Status response: ier = 0 Normal return. ier < 0 Fatal error, no useful results returned. ier > 0 Fatal error, but some eigenvalues and vectors computed. ier = -700 DSEVEX called without first successfully calling earlier stages. ier = ±701 Error in storage allocation. ier = ±702 Illegal specification for lftend and rhtend . ier = ±703 Illegal specification for pdtype . ier = ±704 Illegal specification for which .

ier = ±705	which = highest for an interval that spans zero.
ier = ±706	Insufficient dynamic memory for factorization.
ier = ±707	Internal error during factorization.
ier = ±710	Factorization save error in finite interval analysis.
ier = ±720	Error in storage allocation for first set of Lanczos recurrence vectors.
ier = ±721	Error in storage allocation for second set of Lanczos recurrence vectors.
ier = ±722	Error in storage allocation for eigenvectors.
ier = ±723	Insufficient storage during Lanczos iteration.
ier = ±724	QL iteration did not converge.
ier = ±725	Number of eigenvalues computed exceeded storage limitations.
ier = ±726	Internal error involving access of Lanczos recurrence vectors.
ier = ±727	Singular Value Decomposition did not converge.
ier = ±728	Gram-Schmidt reorthogonalization did not converge.
ier = ±729	Three numeric factorizations in a row failed. Probable cause is that this is not a symmetric generalized eigenproblem.
ier = ±730	Number of trust regions formed exceeded storage limitations.
ier = ±731	Error during restoration of a numeric factorization.
ier = ±740	Error in deallocation of storage for first set of Lanczos recurrence vectors.
ier = ±741	Error in deallocation of storage for second set of Lanczos recurrence vectors.

warnng	Warning status. warnng includes several different warnings encoded in a three-digit decimal integer:
	low order digit:
warnng = xx0	Normal return.
warnng = xx1	Fewer modes computed than requested (interval does not include requested number).
warnng = xx2	Fewer modes computed than requested because eigenvalues are different in size.
warnng = xx3	Both of the above conditions occurred.
	ten's digit:
warnng = x0x	Normal return.
warnng = x1x	At some point in the process, an inconsistent inertia count was found.
	hundred's digit:
warnng = 0xx	Normal return.
warnng = 1xx	Interval had to be expanded because one or both endpoints were very close to eigenvalues.

Notes Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **pctype** argument as 'vibration' or 'buckling'.

Example 1 Compute the lowest 10 eigenvalues of a structural engineering vibration analysis, where matrix *A* is the stiffness matrix *K*, matrix *B* is the mass matrix *M*, and matrices *K* and *M* are positive definite or semi-definite.

```

INTEGER*4 NFOUND,NDISCD,IER,WARNNG
CALL DSEVES (10,'LOWEST','VIBRATION',.FALSE.,-1.0D30,
             .FALSE.,1.0D30,1.0D30,NFOUND,NDISCD,GLOBAL,IER,
             WARNNG)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
IF ( IER .GE. 0 ) THEN
    retrieve eigenvalues and/or eigenvectors
ENDIF

```

- Example 2** Compute the lowest negative eigenvalue and corresponding eigenvector for a structural engineering buckling analysis, where the matrix A is the stiffness matrix K and the matrix B is the geometric stiffness matrix K_g . The matrix K is positive semi-definite.

```

INTEGER*4 NFOUND,NDISCD,IER,WARNNG
CALL DSEVES (1,'LOWEST','BUCKLING',.FALSE.,-1.0D30,
             .TRUE.,0.0D0,0.0D0,NFOUND,NDISCD,GLOBAL,IER,WARNNG)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
IF ( IER .GE. 0 ) THEN
    retrieve eigenvalues and/or eigenvectors
ENDIF

```

- Example 3** Compute the 200 eigenvalues and corresponding eigenvectors closest to 3.14159 for a sparse matrix A .

```

INTEGER*4 NFOUND,NDISCD,IER,WARNNG
CALL DSEVES (1,'NEAREST','ORDINARY',.FALSE.,-1.0D30,
             .FALSE.,1.0D30,3.14159D0,NFOUND,NDISCD,GLOBAL,IER,
             WARNNG)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
IF ( IER .GE. 0 ) THEN
    retrieve eigenvalues and/or eigenvectors
ENDIF

```

Name DSEVEX
Eigenextraction

Purpose This subprogram computes selected eigenvalues and eigenvectors of a sparse symmetric matrix A or of a sparse symmetric matrix pencil (A, B) . The matrix or matrices have already been processed through the structure input, reordering, and value input phases. This is the sophisticated interface; subroutine DSEVES is simpler to use.

DSEVEX allows the user to control three parameters, the block size, the accuracy tolerance, and the shifting scale, that are selected heuristically by DSEVE1. DSEVEX also provides a mechanism for giving initial guesses for the eigenvectors. Each of these parameters is described in more detail in "Usage". Users with a great deal of experience in particular applications may find more efficient settings for these parameters than the general defaults in the code. However, it should be noted that the performance of the package can be degraded significantly by poor choices.

Usage VECLIB:

```

CHARACTER*1  which, pbtype
INTEGER*4    neigvl, mxbksz, nusrv, nfound, ndiscd, ier, warnng
LOGICAL*4    lfinit, rfinit
REAL*8       lftend, rhtend, center, tolact, shfscl, global(150),
              usrvs(norder, nusrv)
CALL DSEVEX(neigvl, which, pbtype, lfinit, lftend, rfinit, rhtend,
center, mxbksz, tolact, shfscl, nusrv, usrvs, nfound, ndiscd, global, ier,
warnng)

```

Input

neigvl	Number of eigenvalues to be found.								
which	Character string indicating which eigenvalues are to be computed:								
	<table> <tbody> <tr> <td>'L' or 'l'</td> <td>The lowest (smallest magnitude) eigenvalues.</td> </tr> <tr> <td>'H' or 'h'</td> <td>The highest (greatest magnitude) eigenvalues.</td> </tr> <tr> <td>'C' or 'c' or 'N' or 'n'</td> <td>The eigenvalues nearest center.</td> </tr> <tr> <td>'A' or 'a'</td> <td>All eigenvalues in the specified interval.</td> </tr> </tbody> </table>	'L' or 'l'	The lowest (smallest magnitude) eigenvalues.	'H' or 'h'	The highest (greatest magnitude) eigenvalues.	'C' or 'c' or 'N' or 'n'	The eigenvalues nearest center.	'A' or 'a'	All eigenvalues in the specified interval.
'L' or 'l'	The lowest (smallest magnitude) eigenvalues.								
'H' or 'h'	The highest (greatest magnitude) eigenvalues.								
'C' or 'c' or 'N' or 'n'	The eigenvalues nearest center.								
'A' or 'a'	All eigenvalues in the specified interval.								

pdtype	Character string indicating type of problem: 'V' or 'v' Generalized symmetric (vibration) problem $Kx = \lambda Mx$, with M positive semi-definite. 'B' or 'b' Generalized symmetric (buckling) problem $Kx = \lambda K_g x$, with K positive semi-definite and K_g possibly indefinite. 'O' or 'o' Ordinary symmetric eigenproblem $Kx = \lambda x$.
lfinit	If .TRUE., the value of the argument lftend is to be used as a restriction on the location of computed eigenvalues. If .FALSE., no lower bound is placed on the value of computed eigenvalues; equivalently, the left endpoint is negative infinity.
lftend	Left endpoint of an interval in which the desired eigenvalues lie; not used if lfinit is .FALSE.
rfinit	If .TRUE., the value of the argument rhtend is to be used as a restriction on the location of computed eigenvalues. If .FALSE., no upper bound is placed on the value of computed eigenvalues; equivalently, the right endpoint is positive infinity.
rhtend	Right endpoint of an interval in which the desired eigenvalues lie; not used if rfinit is .FALSE.
center	A center or critical value for use when eigenvalues nearest some particular value are desired. Referenced only when which specifies 'centered' or 'nearest'.
mxbkosz	A limit on the block size used in the block Lanczos recurrence. In general, it is best to choose a block size as large as, or slightly larger than, the largest multiplicity of the eigenvalues desired. The following constraints also apply: • A block size of 1 is usually less efficient than a block size of 2. • Block sizes larger than 10 are usually less efficient than smaller block sizes. • If mxbkosz is zero or negative, a default block size optimized to the computer system will be chosen.

		In all cases, the Lanczos algorithm may find it necessary to reduce the block size to fit the problem into the storage available.
	tolact	An accuracy tolerance for the eigenvalues, which must be in the range from $\sqrt{\epsilon} \leq \text{tolact} \leq \epsilon$, where ϵ is the machine precision. If a negative or zero value is supplied, a default value of $\epsilon^{2/3}$ will be used. Special care should be taken if the value of tolact is reduced below its default value. In particular, it is important to be sure that the computed eigenvectors remain adequately orthogonal. Subroutine DSEVCK can facilitate such monitoring.
	shfscl	An estimate of the magnitude of the smallest nonzero eigenvalues. The package will compute a heuristic estimate if the user does not supply an estimate (signaled by shfscl ≤ 0.0). This argument is provided only to assist the algorithm in cases where zero lies in the interval of interest or when it is not appropriate to use a shift of 0.0 either because A or K is singular (has rigid body modes) or because this is a buckling analysis.
	nusrvc	Number of user starting vectors provided. Any approximate eigenvectors supplied by the user will be used to create a starting block for the Lanczos algorithm. Provision of starting vectors can be effective when only a very few eigenvalues are to be computed, but it has limited effect otherwise.
	usrsvc	Array containing the starting vectors, one per column. Referenced only if nusrvc is positive. norder is the problem size as set in DSEVIN.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	nfound	The number of eigenvalues computed and confirmed to meet the problem description constraints by inertia (Sturm sequence) computations.
	ndiscd	The number of other eigenvalues computed during the course of the computation but discarded because they

were outside of the specifications for eigenvalues or because they were not confirmed by inertia computations.

ier

Status response:

ier = 0	Normal return.
ier < 0	Fatal error, no useful results returned.
ier > 0	Fatal error, but some eigenvalues and vectors computed.
ier = -700	DSEVEX called without first successfully calling earlier stages.
ier = ±701	Error in storage allocation.
ier = ±702	Illegal specification for lftend and rhtend .
ier = ±703	Illegal specification for pdtype .
ier = ±704	Illegal specification for which .
ier = ±705	which = highest for an interval that spans zero.
ier = ±706	Insufficient dynamic memory for factorization.
ier = ±707	Internal error during factorization.
ier = ±709	Starting block I/O error.
ier = ±710	Factorization save error in finite interval analysis.
ier = ±720	Error in storage allocation for first set of Lanczos recurrence vectors.
ier = ±721	Error in storage allocation for second set of Lanczos recurrence vectors.
ier = ±722	Error in storage allocation for eigenvectors.
ier = ±723	Insufficient storage during Lanczos iteration.
ier = ±724	QL iteration did not converge.
ier = ±725	Number of eigenvalues computed exceeded storage limitations.
ier = ±726	Internal error involving access of Lanczos recurrence vectors.

ier = ± 727	Singular Value Decomposition did not converge.
ier = ± 728	Gram-Schmidt reorthogonalization did not converge.
ier = ± 729	Three numeric factorizations in a row failed. Probable cause is that this is not a symmetric generalized eigenproblem.
ier = ± 730	Number of trust regions formed exceeded storage limitations.
ier = ± 731	Error during restoration of a numeric factorization.
ier = ± 740	Error in deallocation of storage for first set of Lanczos recurrence vectors.
ier = ± 741	Error in deallocation of storage for second set of Lanczos recurrence vectors.
ier = ± 742	Error in deallocation of dynamic storage.

warnng

Warning status. **warnng** includes several different warnings encoded in a three-digit decimal integer: low-order digit:

warnng = xx0	Normal return.
warnng = xx1	Fewer modes computed than requested (interval does not include requested number).
warnng = xx2	Fewer modes computed than requested because eigenvalues are different in size.
warnng = xx3	Both of the above conditions occurred.
ten's digit:	
warnng = x0x	Normal return.
warnng = x1x	At some point in the process, an inconsistent inertia count was found.

hundred's digit:

warnng = 0xx Normal return.

warnng = 1xx Interval had to be expanded because one or both endpoints were very close to eigenvalues.

Notes Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **potype** argument as 'vibration' or 'buckling'.

Example 1 Compute the lowest 10 eigenvalues of a structural engineering vibration analysis, where the matrix *A* is the stiffness matrix *K*, the matrix *B* is the mass matrix *M*, and both matrices *K* and *M* are positive definite or semi-definite.

```

      INTEGER*4 NFOUND,NDISCD,IER,WARNNG
      CALL DSEVEX (10,'LOWEST','VIBRATION',.FALSE.,-1.0D30,
                  .FALSE,.1.0D30,1.0D30,0,0.0D0,0.0D0,0,0.0D0,
                  NFOUND,NDISCD,GLOBAL,IER,WARNNG)
      IF ( IER .NE. 0 ) THEN
         handle error condition
      ENDIF
      IF ( IER .GE. 0 ) THEN
         retrieve eigenvalues and/or eigenvectors
      ENDIF

```

Example 2 Compute the lowest negative eigenvalue and corresponding eigenvector for a structural engineering buckling analysis, where the matrix *A* is the stiffness matrix *K* and the matrix *B* is the geometric stiffness matrix *K_g*. The matrix *K* and *M* is positive semi-definite. A good guess at the eigenvector is available from static analysis and is stored in array USRSVC.

```

      INTEGER*4 NFOUND,NDISCD,IER,WARNNG
      REAL*8    USRSVC(NORDER)
      CALL DSEVEX (1,'LOWEST','BUCKLING',.FALSE.,-1.0D30,
                  .TRUE.,0.0D0,0.0D0,0,0.0D0,0.0D0,1,USRSVC,
                  NFOUND,NDISCD,GLOBAL,IER,WARNNG)
      IF ( IER .NE. 0 ) THEN
         handle error condition
      ENDIF
      IF ( IER .GE. 0 ) THEN
         retrieve eigenvalues and/or eigenvectors
      ENDIF

```


Example 3 Compute the 200 eigenvalues and corresponding eigenvectors closest to 3.14159 for a sparse matrix *A*. Less accuracy is needed than usual, but highly multiple eigenvalues are expected. It is known from previous analyses that eigenvalues closest to zero are about 1.5×10^{-5} .

```
INTEGER*4 NFOUND,NDISCD,IER,WARNNG
CALL DSEVEX (200,'NEAREST','ORDINARY',.FALSE.,-1.0D30,
             .FALSE.,1.0D30,3.14159D0,10,1.0D-8,1.5D-5,
             0,0.0D0,NFOUND,NDISCD,GLOBAL,IER,WARNNG)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
IF ( IER .GE. 0 ) THEN
    check orthogonality with DSEVCK
    retrieve eigenvalues and/or eigenvectors
ENDIF
```

Name	DSEVRC Return Eigenvalue/Eigenvector Results	
Purpose	This subprogram retrieves all or selected eigenvalues and eigenvectors after eigenextraction.	
Usage	VECLIB: INTEGER*4 levalu, fstval, lstval, ldevct, ier REAL*8 evalue(levalu), evectr(ldevct, levalu), global(150) CALL DSEVRC(levalu, evalue, fstval, lstval, evectr, ldevct, global, ier)	
Input	levalu	Length of vector for storing eigenvalues.
	fstval	Ordinal position of first eigenvalue to be returned in evalue.
	lstval	Ordinal position of last eigenvalue to be returned in evalue.
	The sign of fstval and lstval determine whether the eigenvalues returned are from the confirmed set or the discarded set. Positive indices indicate eigenvalues in the final confirmed trust region, while negative indices indicate the discardable set. The following pairs illustrate the meaning of fstval and lstval :	
	fstval	lstval
	1	nfound All final trust-region eigenvalues.
	-1	-ndiscd All other eigenvalues.
	1	-ndiscd All computed eigenvalues.
	<i>i</i>	<i>i</i> <i>i</i> -th final trust-region eigenvalue.
	- <i>i</i>	- <i>i</i> <i>i</i> -th discardable eigenvalue.
	<i>i</i>	<i>j</i> <i>i</i> -th through <i>j</i> -th final trust-region eigenvalues.
	<i>i</i>	<i>-j</i> <i>i</i> -th through nfound -th final trust-region eigenvalues and 1st through <i>j</i> -th discardable eigenvalues.
	ldevct	The leading dimension of array evectr as declared in the calling program unit, with ldevct ≥ norder , where norder is the matrix order as specified in the call to DSEVIN.

Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	evalue	List of selected eigenvalues.
	evectr	Corresponding array of eigenvectors. Each eigenvector is normalized such that
		$x^T B x = x^T M x = 1$ (vibration)
		$x^T A x = x^T K x = 1$ (buckling)
		$x^T x = 1$ (ordinary)
	ier	Status response:
		ier = 0 Normal return.
		ier = -800 Incorrect processing path.
		ier = -801 levalu not large enough.
		ier = -802 ldevct smaller than order of problem.
		ier = -803 System error.
		ier = -804 Error in input, indices out of range.

Example 1 Retrieve all of the final trust-region eigenvalues and eigenvectors.

```

INTEGER*4 NEVALS,LDEVCT,NFOUND,IER
REAL*8     EVALUE(100),EVECTR(10000,100),GLOBAL(150)
NEVALS = 100
LDEVCT = 1000
CALL DSEVRC (NEVALS,EVALUE,1,NFOUND,EVECTR,LDEVCT,GLOBAL,
             IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF

```

Example 2 Retrieve all of the confirmed eigenvalues and the first three discarded eigenvalues, which lie in a cluster with the last confirmed eigenvalue and the corresponding eigenvectors.

```

INTEGER*4 NEVALS,IER
REAL*8     EVALUE(55),EVECTR(10000,100),GLOBAL(150)
NEVALS = 55
LDEVCT = 10000
CALL DSEVRC (NEVALS,EVALUE,1,-3,EVECTR,LDEVCT,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF

```

Name	DSEVRL Return Eigenvalue Results	
Purpose	This subprogram retrieves all or selected eigenvalues after eigenextraction.	
Usage	VECLIB: <pre> INTEGER*4 levalu, fstval, lstval, ier REAL*8 evalue(levalu), global(150) CALL DSEVRL(levalu, evalue, fstval, lstval, global, ier) </pre>	
Input	levalu	Length of vector for storing eigenvalues.
	fstval	Ordinal position of first eigenvalue to be returned in evalue.
	lstval	Ordinal position of last eigenvalue to be returned in evalue.
	The sign of fstval and lstval determine whether eigenvalues returned are from the confirmed set or the discarded set. Positive indices indicate eigenvalues in the final confirmed trust region, while negative indices indicate the discardable set. The following pairs illustrate the meaning of fstval and lstval :	
	fstval	lstval
	1	nfound All final trust-region eigenvalues.
	-1	-ndiscd All other eigenvalues.
	1	-ndiscd All computed eigenvalues.
	<i>i</i>	<i>i</i> <i>i</i> -th final trust-region eigenvalue.
	- <i>i</i>	- <i>i</i> <i>i</i> -th discardable eigenvalue.
	<i>i</i>	<i>j</i> <i>i</i> -th through <i>j</i> -th final trust-region eigenvalues.
	<i>i</i>	- <i>j</i> <i>i</i> -th through nfound -th final trust-region eigenvalues and 1st through <i>j</i> -th discardable eigenvalues.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	evalue	List of eigenvalues.

ier	Status response:
ier = 0	Normal return.
ier = -800	Incorrect processing path.
ier = -801	levalu not large enough.
ier = -803	Input error, indices out of range.

Example 1 Retrieve all of the final trust region eigenvalues.

```

INTEGER*4 NEVALS,NFOUND,IER
REAL*8     EVALUE(NEVALS),GLOBAL(150)
NEVALS = 100
LDEVCT = 1000
CALL DSEVRL (NEVALS,EVALUE,1,NFOUND,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF

```

Example 2 Retrieve all of the computed eigenvalues.

```

INTEGER*4 NEVALS,NDISCD,NFOUND,IER
REAL*8     EVALUE(NEVALS),GLOBAL(150)
IF ( NDISCD .GT. 0 ) THEN
    CALL DSEVRL (NEVALS,EVALUE,1,-NDISCD,GLOBAL,IER)
ELSE
    CALL DSEVRL (NEVALS,EVALUE,1,NFOUND,GLOBAL,IER)
ENDIF
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF

```

Name DSEVCK
Check Accuracy of Results

Purpose This subprogram performs an *a posteriori* check on the accuracy of the computed eigenvalues and eigenvectors. The eigenvectors should satisfy the following pairs of equations:

$$\begin{array}{ll} X^T K X = \text{diag}(\{\lambda_i\}), & X^T M X = I \quad (\text{vibration}) \\ X^T K_g X = \text{diag}(\{1/\lambda_i\}), & X^T K X = I \quad (\text{buckling}) \\ X^T A X = \text{diag}(\{\lambda_i\}), & X^T X = I \quad (\text{ordinary}) \end{array}$$

The subroutine computes the relevant matrix-matrix products to confirm that these equations hold to a reasonable numerical accuracy.

Usage VECLIB:
INTEGER*4 ier
LOGICAL*4 nodscd, ortwrn
REAL*8 discrp, global(150)
CALL DSEVCK(nodscd, ortwrn, discrp, global, ier)

Input nodscd If .TRUE., check only the eigenpairs whose eigenvalues are confirmed to lie in the single final trust region. If FALSE., check all of the computed eigenpairs, including those from 'discardable' eigenvalues.

Updated global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output ortwrn If .TRUE., the maximum relative discrepancy detected by checking the validity of these equations exceeded what was expected for the accuracy tolerance used in the eigenextraction.

discrp Maximum relative discrepancy detected.

ier Status response:

ier = 0 Normal return.

ier = -800 Incorrect processing path.

ier = -801 Error in storage allocation.

ier = -802 Error in retrieving eigenvectors.

ier = -804 System error.

Notes Additional diagnostic output from this subprogram is written onto the Fortran logical unit specified in the last call to DSEVIN or DSEVOC. The amount of output is independent of the message level value set in the last call to DSEVIN or DSEVOC.

Example Check only the final trust region eigenvalues and eigenvectors.

```
LOGICAL*4 ORTWRN
REAL*8 DISCRP,GLOBAL(150)
INTEGER*4 IER
CALL DSEVCK (.TRUE.,ORTWRN,DISCRP,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name	DSEVDA Deallocate Working Storage	
Purpose	This subprogram deallocates working storage at the end of processing. If the program using the package is going to continue execution when use of the package is completed, then it is recommended that the user deallocate the dynamically-allocated working storage to reduce system impact.	
Usage	VECLIB: INTEGER*4 ier REAL*8 global(150) CALL DSEVDA(global, ier)	
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: ier = 0 Normal return. ier = -801 Error in dynamic storage deallocation.
Example	Deallocate working storage after use of package. REAL*8 GLOBAL(150) CALL DSEVDA (GLOBAL, IER) IF (IER .NE. 0) THEN handle error condition ENDIF	

Name	DSEVOC Output Control	
Purpose	This subprogram may be called at any point after subprogram DSEVIN to alter the output message level and the Fortran output unit number for message output.	
Usage	VECLIB: <pre> INTEGER*4 msglvl, output REAL*8 global(150) CALL DSEVOC(msglvl, output, global) </pre>	
Input	<pre> msglvl output </pre>	Message level for printable output: msglvl ≤ 0 Suppress all output. msglvl = 1 Error messages, summary statistics and inertia information. msglvl = 2 More complete statistics. msglvl = 3 First stage of debugging output. msglvl = 4 Complete debugging output. output Fortran logical unit number to which all output will be written.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Example	Increase message level from 1 to 2 while leaving the Fortran logical unit number for the output unit at 6. <pre> REAL*8 GLOBAL(150) CALL DSEVOC (2,6,GLOBAL) </pre>	

Name	DSEVPS Print Statistics
Purpose	This subprogram prints available statistics about the original matrices, amount of work space in use, amount required for the next phase, and maximum amount used thus far. Also, this subprogram prints storage and arithmetic requirements, CPU time used, and computational rate for Cholesky factorization. The amount of information printed depends on the stage of execution. The number of lines of output ranges from 8 to 30, with the width of the lines being less than 80 characters.
Usage	VECLIB: REAL*8 global(150) CALL DSEVPS(global)
Input	global Global communications array for this problem.
Example	Print the statistics after the package has completed a numeric solution (either after DSEVSL or DSEVFS). REAL*8 GLOBAL(150) CALL DSEVPS (GLOBAL)

Name	DSEVRS Restore Problem State from a Savefile	
Purpose	This subprogram restores the working problem from the state stored on the user-specified I/O file by subprogram DSEVSV.	
Usage	VECLIB: <pre> INTEGER*4 svfile, ier REAL*8 global(150) CALL DSEVRS(svfile, global, ier) </pre>	
Input	svfile	Fortran logical unit number of a file containing the saved problem state as created by DSEVSV.
Output	global	Global communications array restored from svfile .
	ier	Status response:
	ier = 0	Normal return.
	ier = -901	Error in storage allocation.
	ier = -902	I/O error detected by Fortran.
Example	Restart the eigenanalysis from the save file on Fortran logical unit 42 created by subroutine DSEVSV. <pre> REAL*8 GLOBAL(150) CALL DSEVRS (42,GLOBAL,IER) IF (IER .NE. 0) THEN handle error condition ENDIF </pre>	

Name	DSEVSV Save Problem State to a Savefile	
Purpose	This subprogram saves the current problem for later restart at its current state. The global communication array and all working storage are written onto the user-specified I/O file using standard Fortran unformatted sequential write statements. The file is rewound before and after use.	
Usage	VECLIB: <pre> INTEGER*4 svfile, ier REAL*8 global(150) CALL DSEVSV(svfile, global, ier) </pre>	
Input	svfile	Fortran logical unit number of the file onto which the state is to be saved. The file will be rewound before and after use.
	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response:
		ier = 0 Normal return.
		ier = -902 I/O error detected by Fortran.
Example	Save the current state on Fortran logical unit 42. <pre> REAL*8 GLOBAL(150) CALL DSEVSV (42,GLOBAL,IER) IF (IER .NE. 0) THEN handle error condition ENDIF </pre>	

8 Skyline Linear Equations

Overview

This chapter describes state-of-the-art software for the direct solution of sparse systems of linear equations with symmetric positive definite matrices. The equations are represented and processed using a so-called "skyline" data structure, also known as a variable-band, envelope, or profile data structure. This package of subprograms provides very efficient use of the Hewlett-Packard architecture in conjunction with powerful techniques for using the sparsity of the problem to reduce the cost of solution. Accuracy is assured through appropriate numerical techniques.

This chapter explains how to use the skyline linear equation subprograms to solve systems of sparse linear equations where the coefficient matrix is symmetric positive definite and is represented with a variable band storage scheme. Since the matrix is positive definite, pivoting is not needed to ensure numerical stability. This package optionally uses pivoting to reduce storage requirements and arithmetic.

The following documents provide supplemental material for this chapter:

Ashcraft, C.C., R.G. Grimes, J.G. Lewis, B.W. Peyton, and H.D. Simon. "Progress in Sparse Matrix Methods for Large Linear Systems on Vector Supercomputers," *The International Journal of Supercomputer Applications*. 1987. Vol. 1, No. 4. pp. 10-30.

Duff, I.S., A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Methods*. Oxford, England: Clarendon Press. 1986.

George, J.A. and J.W. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Englewood Cliffs, NJ: Prentice-Hall, Inc. 1981.

Lewis, J.G. "The Gibbs-Pool-Stockmeyer and Gibbs-King Algorithms for Reordering Sparse Matrices." *ACM Transactions of Mathematical Software*. June, 1982. Vol. 8, No. 2. pp. 190-194.

Chapter Objectives

After you read this chapter you will:

- Understand what a sparse system is
- Understand how to use these subprograms to solve linear systems
- Know some of the issues in choosing an optimal method for a specific problem

This skyline sparse matrix linear equation software is designed so that it is possible to call a single subprogram to solve a single system of skyline symmetric linear equations. However, this requires a particular format for the skyline matrix. This package provides other approaches that provide a general interface to alternative representations of the skyline matrix. These optional approaches, however, require the user to call a sequence of subprograms. This is similar to, but significantly more elaborate than, the use of LINPACK as described in Chapter 4.

What You Need to Know to Use These Subprograms

Sparsity and Storage Formats

Sparse matrices are matrices in which most of the entries are zero. The goal of skyline matrix software is to take maximum advantage of these zero entries to reduce storage and arithmetic. Storage is reduced by not storing zero entries; arithmetic is reduced by not performing operations on entries that are known to be zero.

It is easiest to see how to economize on storage. Suppose that an n -by- n matrix A has only nz nonzero entries. Then A could be specified completely by storing each of the nonzero values in an array of length nz that was accompanied by two integer arrays of length nz holding corresponding row and column indices. Thus, $3nz$ storage suffices where n^2 storage is required for the corresponding dense matrix format. Consider, for example, the following matrix:

11	0	13	14	0	0
0	22	23	0	25	0
31	32	33	0	35	0
41	0	0	44	45	0
0	52	53	54	55	0
0	0	0	0	0	66

This matrix could be represented in the format described above by three arrays, IROW, JCOL, and MXVALU, as shown in Figure 8-1.

Figure 8-1 Row and Column Index Sparse Matrix Representation

IROW	=	1	3	4	2	3	5	1	2	3	5	1	4	5	2	3	4	5	6
JCOL	=	1	1	1	2	2	2	3	3	3	3	4	4	4	5	5	5	5	6
MXVALU	=	11	31	41	22	32	52	13	23	33	53	14	44	54	25	35	45	55	66

In Figure 8-1, the matrix entries have been listed in order within each column, and the columns have been listed in order, although the representation does not require that much structure. However, if the entries are required to be ordered by row and column, even less storage is needed. In addition, symmetry in the matrix can be used by storing only the entries either on and above or on and below the main diagonal. Other contexts, such as finite element analysis, can make more concise representations of the locations of the nonzeros.

One of the disadvantages of representing the sparse matrix in the above manner is that all references to the matrix elements must be done indirectly, through the index arrays. Indirection can slow down the computation. However, if the nonzeros of the matrix all lie fairly close to the diagonal, it is feasible to store the matrix as a band matrix so that rows and columns can be accessed as vectors. When this is done, zeros within the band are both stored and operated upon, so there is usually an increase in storage and computing. But if vector and matrix operations can replace the indirection mentioned above, the solution time may be reduced regardless.

The band storage scheme can be improved, in terms both of storage and computation, by storing only the entries between the first and last nonzero in each row or column. This results in a band-like matrix with ragged edges, which requires a more complicated data structure and algorithm. Such schemes are called *envelope*, *variable-band*, *skyline*, or *profile* methods. Again, symmetry in the matrix can be used by storing only the upper- or lower-triangular portion of the variable band.

This package adopts a particular internal format, called *skyline* or *reverse skyline* representations, that allow banded symmetric matrices to be stored in $nz+n+1$ storage locations, where nz represents the number of values within the band. $DIAG(i)$ indicates the position of the i th diagonal element in the $MXVALU$, where the values in $MXVALU$ have been listed in forward or reverse order within each column, and the columns have been listed in order. $DIAG(i)$ indicates that last position of the i th column for the skyline's representation, as shown in Figure 8-2, and the first position for the reverse skyline's representation, as shown in Figure 8-3. $DIAG(n+1) = nz+1$ conveys the number of values to the package.

Figure 8-2 Skyline Matrix Representation

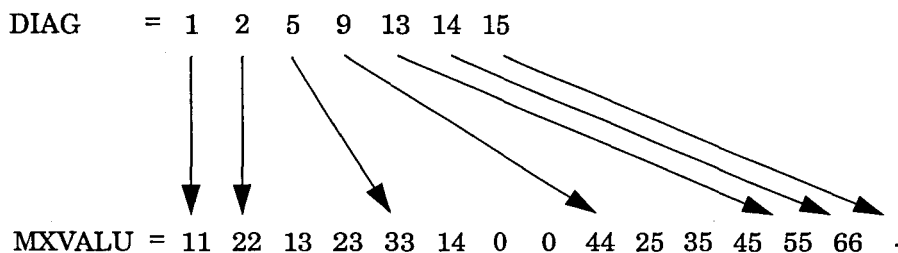
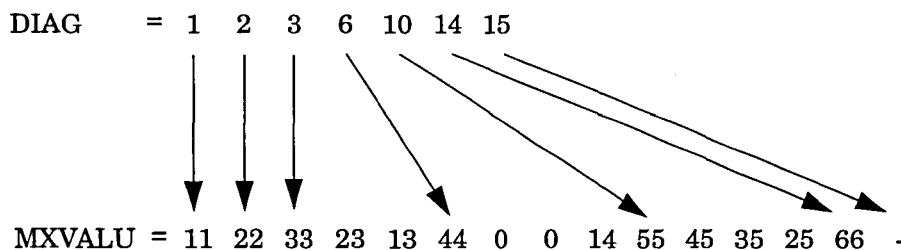


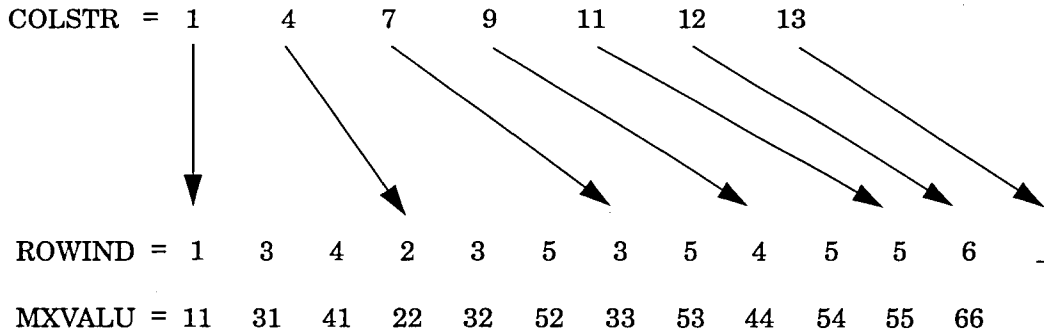
Figure 8-3 Reverse Skyline Matrix Representation



This package also adopts another input format that allows arbitrary symmetric matrices to be stored in $2nz+n+1$ storage locations, where nz represents the number of nonzeros in the lower triangle of the matrix. In essence, the **JCOL** array of Figure 8-1, which has repeated entries, is replaced by a shorter array that gives the index of the first element of each column in the **MXVALU** array, and an indication of the number of elements in each column. These two pieces of information per matrix column can be represented in a single array **COLSTR** of length $n+1$ if the convention is adopted that the number of nonzeros in column j is given by $COLSTR(j+1) - COLSTR(j)$ and if $COLSTR(n+1)$ is set

accordingly. This sparse matrix format, known as the *column pointer, row index* representation, is illustrated in Figure 8-4.

Figure 8-4 **Column Pointer, Row Index Sparse Matrix Representation**



There are three ways of communicating the coefficient matrix to the package. One is a totally general form, which allows the user to store the matrix outside the package in whatever form is most convenient. The other two ways require that the user store the entire matrix in a form similar to the internal format or at least with all entries in each column contiguous in memory. Any of these three can be used. However, the most general form carries additional overhead in computer time.

Direct Versus Iterative Solution

This package is a *direct* linear equation solver. That is, it computes an explicit factorization of the matrix in a sparse analogy of the dense matrix subprograms DPOFA/DPOSL in LINPACK. These capabilities allow for general symmetric sparse matrices and guarantee high accuracy. There are applications where special, factorization-free algorithms can be used effectively. Algorithms appropriate for these cases *iterate* toward a solution, improving an approximate solution at each iteration.

Unfortunately, there is no generally effective iterative algorithm, only a collection of algorithms each effective for particular classes of problems. Used in the appropriate contexts, with only single or a few right-hand sides, and with only limited accuracy required of the solutions, iterative algorithms can be faster than direct methods. Used in the wrong contexts, iterative methods may become inordinately expensive and inaccurate. In contrast, the subprograms described here are designed to function well in general and, additionally, represent the algorithm of choice for many classes of problems.

Fill and Reordering

This linear equation package computes the LDL^T factorization of the matrix A and uses it to solve the system $Ax = b$. Here, L is a unit lower triangular matrix and D is diagonal. However, the sparsity of A is not sufficient to assure that L is sparse. Indeed, L will have nonzeros wherever A 's lower triangle has nonzeros, but it will also have nonzeros in other positions where A 's entries are zero. These additional entries are known as *fill*. While fill is an intrinsic facet of the factorization process, the amount of fill is often controllable in the following sense. If P is a matrix representing a permutation of the variables of the problem, the factorization of the matrix PAP^T , can often have significantly less fill than does the factorization of A , *provided that the permutation P is chosen appropriately*.

This package includes an algorithm to choose a permutation to reorder the matrix so that its nonzero entries lie close to the diagonal but also allows the user to provide the permutation. The principle is the same in both approaches: if the matrix is reordered so that the factor L has fewer nonzero entries, then there is correspondingly less work in computing the factorization. The reordering process precedes the actual factorization. The package solves the problem $PAP^T(Px) = (Pb)$. This solution is invisible to the user.

Stability

This package is designed to solve banded symmetric linear systems where the coefficient matrix is *positive definite*. The factorization $PAP^T = LDL^T$, with D diagonal, is always stable when A is positive definite. This has the particular effect that the reordering phase can choose any permutation P to maintain sparsity in the factor L .

Global Communications Array

All of the subprograms in this package use dynamic memory allocation capabilities (refer to Chapter 12) to free users from the often difficult issue of allocating storage for the factors of the matrix. This internal storage is invisible to the user. When the sophisticated lower-level routines are used, knowledge of the internal storage is passed from subprogram to subprogram through a single communications array. This array, called **global** in each of the calling sequences, is a fixed-length double precision array of length 150. It must not be altered by the user, as it represents the essential knowledge of the problem. Because it is the only identification for a problem, the user can handle multiple problems simultaneously by having multiple communications arrays with different names.

Error Convention

Each subprogram has an error return flag, **ier**, as one of its arguments. A zero value returned in **ier** is the indication of successful processing. Fatal errors are signaled through negative values; each is a negative integer in the range $-1000 \leq \text{ier} \leq -100$. The hundreds digit indicates the phase of processing in which the error occurred; the other digits specify the error itself. Note that an option allows the user to control whether or not this package will print error messages in addition to returning an error flag.

Output Controls

This package differs from most library subroutines in providing optional printed or printable output. The amount of output is controlled by an integer variable, **msglvl**, specifying the *message level*. Setting **msglvl** ≤ 0 suppresses all printed messages, including error messages, and thus should generally be avoided. With **msglvl** = 1, a small amount of runtime statistics and any error messages will be printed. Whenever **msglvl** > 1 , the complete set of runtime statistics will be printed. If **msglvl** ≥ 3 , various arrays whose length is on the order of the number of equations will be printed. If **msglvl** ≥ 4 , then volumes of output will be produced for debugging purposes.

We recommend that, as a rule, the user set **msglvl** = 1. The higher **msglvl** values (≥ 3) are intended for debugging purposes and generate large amounts of printed output. Further, their use for this purpose may require some knowledge of the data structures being used by the package.

Paths of Control

The key to efficiently using this package is an understanding of possibilities for reusing work. As in solving dense linear systems, factored matrices can be used as often as needed to solve additional systems. Thus, the subprogram DSKYSL can be called repeatedly to solve additional systems with the same coefficient matrix after subprogram DSKYFA or DSKYFS has completed successfully. It is common to encounter sequences of sparse linear systems where the coefficient matrices change, but their sparsity structure, that is, the location of the nonzeros, remains fixed. In such cases, it is necessary to compute a factorization with DSKYFA for each coefficient matrix, but the work of choosing a reordering does not have to be repeated. The possible structures of reuse are illustrated in Figure 8-5 and below:

What You Need to Know to Use These Subprograms

```
initialization
input matrix structure
reorder matrix
for m = 1 to number_of_structurally_identical_coefficient_matrices do
  input values of matrix entries
  factor matrix
  for r = 1 to number_of_right_hand_sides_for_this_coefficient_matrix do
    solve
  endfor
endfor
```

Note that entering matrix values for a new coefficient matrix renders inaccessible the previously computed factorization. Similarly, entering matrix structure renders any previously reordered matrix structure inaccessible. However, the save and restart capabilities described in the utility subprograms can be used to save multiple structures or factorizations or to interrupt the package at any point.

Figure 8-5 **Paths of Control**

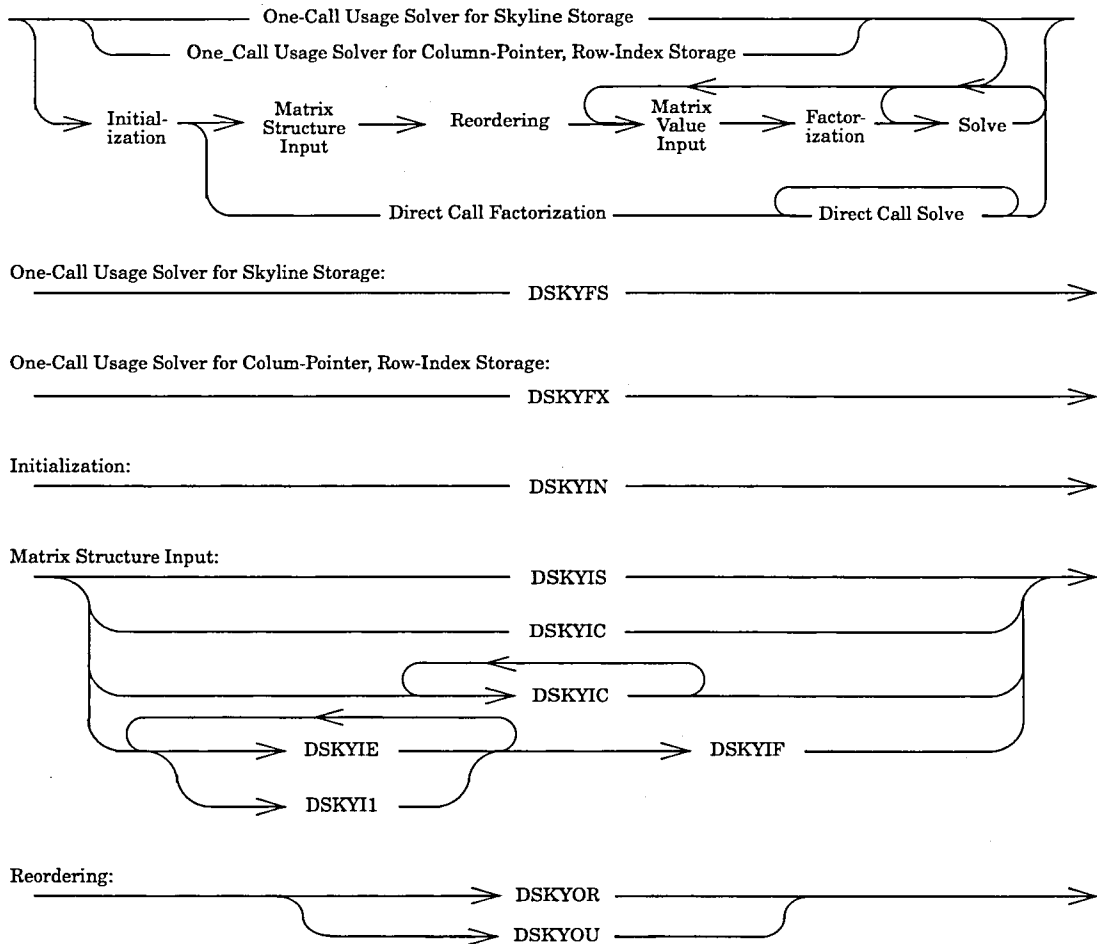
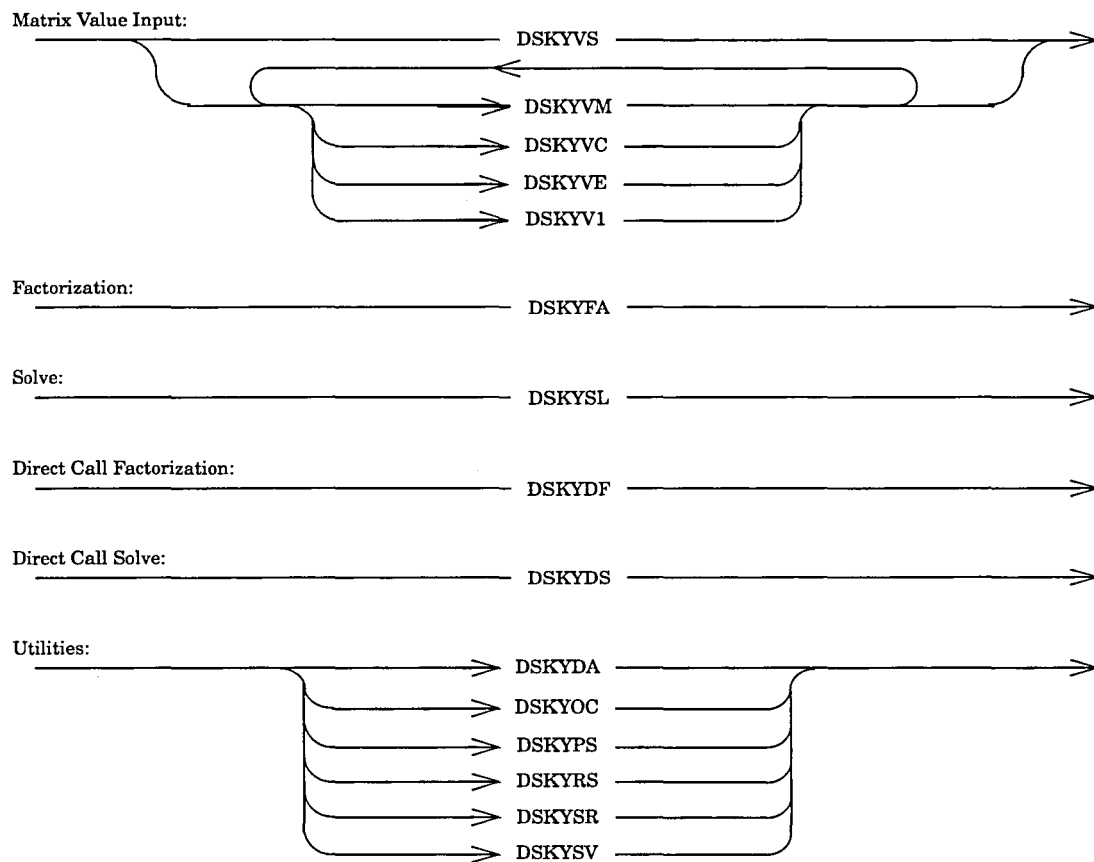


Figure 8-6 **Paths of Control (continued)**



Sample Program

As illustrated in Figure 8-5, there are many possible paths of control through this sparse matrix package. The following sample program is provided to show one possible path through the package. It is intended to demonstrate that the package is not as difficult to use as Figure 8-5 implies.

In this example, the row and column indices and the corresponding value for each nonzero entry of the matrix are stored in the three arrays IROW, JCOL, and MXVALU. The right-hand-side vector is stored in the array RHS. This example demonstrates the use of the subroutines for solving sparse systems of linear equations when subroutine DSKYFS cannot be used. This code assumes that there are NNZERO nonzero entries in the matrix, which has NEQNS rows and columns. Values of NNZERO and NEQNS are set prior to this code segment.

```

      INTEGER*4  NEQNS,IROW,JCOL,IER
      REAL*8     RHS(NEQNS),GLOBAL(150),MXVALU(NNZERO)

C
C      ... INITIALIZE THE SPARSE MATRIX PACKAGE
C

      CALL DSKYIN (NEQNS,1,6,GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000

C
C      ... INPUT THE MATRIX STRUCTURE
C

      DO 100 K = 1, NNZERO
        I = IROW(K)
        J = JCOL(K)
        CALL DSKYI1 (I,J,GLOBAL,IER)
        IF ( IER .NE. 0 ) GO TO 8000
100 CONTINUE

      CALL DSKYIF (GLOBAL, IER)
      IF ( IER .NE. 0 ) GO TO 8000

C
C      ... REORDER THE MATRIX
C

      CALL DSKYOR (GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000

C      -
C      ... INPUT MATRIX VALUES
C      -

      DO 200 K = 1, NNZERO
```

Subprograms Included in This Chapter

```

        I      = IROW(K)
        J      = JCOL(K)
        VALUE = MXVALU(K)
        CALL DSKYV1 (I,J,VALUE,GLOBAL,IER)
        IF ( IER .NE. 0 ) GO TO 8000
200 CONTINUE

C      -
C      ... FACTOR THE MATRIX
C      -

        CALL DSKYFA (GLOBAL,IER)
        IF ( IER .NE. 0 ) GO TO 8000

C      -
C      ... SOLVE FOR A GIVEN RIGHT HAND SIDE
C      -

        CALL DSKYSL (RHS,GLOBAL,IER)
        IF ( IER .NE. 0 ) GO TO 8000

C      ... USE THE SOLUTION STORED IN ARRAY RHS
C
C      .
C      .
C      .
C
C      ... ERROR TRAP
C
8000 .....
```

Subprograms Included in This Chapter

Following are the subprograms included with VECLIB for the direct solution of sparse systems of linear equations with symmetric positive definite matrices.

Name	DSKYFS One-Call Usage	
Purpose	This subprogram provides a one-call interface to the skyline linear equation solution package when you provide the matrix in the skyline or reverse skyline data structure, shown in Figure 8-2 or 8-3, respectively. If this subroutine does not fit your needs, a more flexible interface is available by using the other subroutines in the package.	
Usage	VECLIB: <pre> INTEGER*4 neqns, ier, diag(neqns+1), idata, msglvl, output REAL*8 values(nnzero), rhs(neqns), global(150) CALL DSKYFS(neqns, values, diag, rhs, idata, msglvl, output, global, ier) </pre>	
Input	neqns	Number of equations; neqns > 0.
	values	Array of matrix values in skyline or reverse skyline order.
	diag	Array of diagonal positions. diag (<i>i</i>) indicates the position of the <i>i</i> th diagonal matrix element in array values . diag (neqns +1) must be set to one more than the number of values stored in array values .
	idata	Specifies the skyline data structure format as follows: <pre> idata = 1 Array values is in skyline format. idata = 2 Array values is in reverse skyline format. </pre>
	msglvl	Message level for printable output. Options: <pre> msglvl ≤ 0 Suppress all output. msglvl = 1 Error messages and summary statistics. msglvl = 2 More complete statistics. msglvl = 3 First stage of debugging output. msglvl = 4 Complete debugging output. </pre>
	output	Fortran logical unit number to which all printable output will be written.
Updated	rhs	On input, rhs contains the right-hand side. On output, rhs has been overwritten with the computed solutions.

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

```
INTEGER*4 DIAG(7)
REAL*8    VALUES(14),RHS(6),GLOBAL(150)
DIAG(1) = 1
DIAG(2) = 2
DIAG(3) = 5
DIAG(4) = 7
DIAG(5) = 11
DIAG(6) = 14
DIAG(7) = 15

VALUES(1) = 4.0
VALUES(2) = 4.0
VALUES(3) = 1.0
VALUES(4) = 1.0
VALUES(5) = 4.0
VALUES(6) = 1.0
VALUES(7) = 4.0
VALUES(8) = 1.0
VALUES(9) = 1.0
VALUES(10) = 1.0
VALUES(11) = 4.0
VALUES(12) = 1.0
VALUES(13) = 0.0
VALUES(14) = 4.0

RHS(1) = 1.0
RHS(2) = 0.0
RHS(3) = 1.0
RHS(4) = 1.0
RHS(5) = 0.0
RHS(6) = 0.0

CALL DSKYFS (6,VALUES,DIAG,RHS,1,1,6,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name	DSKYFX One-Call Usage	
Purpose	This subprogram provides a one-call interface to the skyline linear equation solution package when you provide the coefficient matrix in the column pointer, row index representation shown in Figure 8-4. If this subroutine does not fit your needs, a more flexible interface is available by using the other subroutines in the package.	
Usage	VECLIB: <pre> INTEGER*4 neqns, nnzero, colstr(neqns+1), rowind(nnzero), ier, msglvl, output REAL*8 global(150), values(nnzero), rhs(neqns) CALL DSKYFX(neqns, values, rhs, colstr, rowind, msglvl, output, global, ier) </pre>	
Input	neqns	Number of equations; neqns > 0.
	values	Array of matrix values corresponding in positions to the indices in rowind .
	colstr	colstr(j) gives the index in rowind of the first nonzero in the lower triangular part of column <i>j</i> of the matrix. All of the nonzeros for column <i>j</i> are found, in ascending order, in rowind(colstr(j)) , rowind(colstr(j)+1) , ..., rowind(colstr(j+1)-1) . colstr(norder+1) must be set to one greater than the total number of nonzeros in the lower triangular part of the matrix (nnzero), where norder is the matrix order as specified in the call to DSKYIN.
	rowind	List of row indices for all nonzeros, in ascending order within each column, in the lower triangle part of the matrix. Input of diagonal entries is optional.
	msglvl	Message level for printable output. Options: msglvl ≤ 0 Suppress all output. msglvl = 1 Error messages and summary statistics. msglvl = 2 More complete statistics. msglvl = 3 First stage of debugging output. msglvl = 4 Complete debugging output.
	output	Fortran logical unit number to which all printable output will be written.

Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
	rhs	On input, rhs contains the right-hand side. On output, rhs has been overwritten with the computed solution.
Output	ier	Status response:
	ier = 0	Normal return.
	ier = -600	Dynamic storage error.
	ier = -601	Error from DSKYIN.
	ier = -602	Error from DSKYIM.
	ier = -603	Error from DSKYOR.
	ier = -604	Error from DSKYVM.
	ier = -605	Error from DSKYFA.
	ier = -606	Error from DSKYSL.
Notes	Calling DSKYFX is equivalent to calling DSKYIN, DSKYIM, DSKYOR, DSKYVM, DSKYFA, and DSKYSL in sequence. You may follow the call to DSKYFX with other calls to subprograms in the package to solve additional right-hand sides, to input new matrix values, to input a new matrix structure, or to perform utility functions.	
Example	Solve the small (order 6) sparse system of linear equations where the matrix <i>A</i> and right-hand side <i>b</i> are	

$$\begin{array}{rcc}
 & \begin{matrix} 4.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \end{matrix} & 1.0 \\
 & \begin{matrix} 0.0 & 4.0 & 1.0 & 0.0 & 1.0 & 0.0 \end{matrix} & 0.0 \\
 A = & \begin{matrix} 1.0 & 1.0 & 4.0 & 1.0 & 1.0 & 0.0 \end{matrix} & 2.0 \\
 & \begin{matrix} 0.0 & 0.0 & 1.0 & 4.0 & 1.0 & 1.0 \end{matrix} & 1.0 \\
 & \begin{matrix} 0.0 & 1.0 & 1.0 & 1.0 & 4.0 & 0.0 \end{matrix} & 0.0 \\
 & \begin{matrix} 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 4.0 \end{matrix} & 3.0 \\
 & & b =
 \end{array}$$

Printed output will be to Fortran logical unit 6 and only printed error messages and some runtime statistics will be printed.

```

INTEGER*4 COLSTR(7),ROWIND(13),IER
REAL*8     GLOBAL(150),VALUES(13),RHS(6)
MSGVLVL = 1
OUTPUT = 6

COLSTR(1) = 1
COLSTR(2) = 3
COLSTR(3) = 6
COLSTR(4) = 9
COLSTR(5) = 12
COLSTR(6) = 13
COLSTR(7) = 14

ROWIND(1) = 1
ROWIND(2) = 3
ROWIND(3) = 2
ROWIND(4) = 3
ROWIND(5) = 5
ROWIND(6) = 3
ROWIND(7) = 4
ROWIND(8) = 5
ROWIND(9) = 4
ROWIND(10) = 5
ROWIND(11) = 6
ROWIND(12) = 5
ROWIND(13) = 6
VALUES(1) = 4.0
VALUES(2) = 1.0
VALUES(3) = 4.0
VALUES(4) = 1.0
VALUES(5) = 1.0
VALUES(6) = 4.0
VALUES(7) = 1.0
VALUES(8) = 1.0
VALUES(9) = 4.0
VALUES(10) = 1.0
VALUES(11) = 1.0
VALUES(12) = 4.0
VALUES(13) = 4.0

RHS(1) = 1.0
RHS(2) = 0.0
RHS(3) = 2.0
RHS(4) = 1.0
RHS(5) = 0.0
RHS(6) = 3.0

CALL DSKYFX (NEQNS,VALUES,RHS,COLSTR,ROWIND,MSGVLVL,
&           OUTPUT,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF

```

Name	DSKYIN Initialize Skyline Linear Equations	
Purpose	This subprogram provides the necessary information to begin processing with the skyline linear equation solution package.	
Usage	VECLIB: INTEGER*4 neqns, msglvl, output, ier REAL*8 global(150) CALL DSKYIN(neqns, msglvl, output, global, ier)	
Input	neqns	Order of matrix; neqns > 0.
	msglvl	Message level for printable output: msglvl ≤ 0 Suppress all output. msglvl = 1 Error messages and summary statistics. msglvl = 2 More complete statistics. msglvl = 3 First stage of debugging output. msglvl = 4 Complete debugging output.
	output	Fortran logical unit number to which all printable output will be written.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: ier = 0 Normal return. ier = -101 Error in dynamic storage allocation. ier = -102 neqns ≤ 0.
Example	Prepare to solve a system of equations of order 10,000. Obtain error messages and standard minimal output on Fortran logical unit 6.	

```

      INTEGER*4 NEQNS, IER
      REAL*8    GLOBAL(150)
      NEQNS = 10000
      CALL DSKYIN (NEQNS, 1, 6, GLOBAL, IER)
      IF ( IER .NE. 0 ) THEN
         handle error condition
      ENDIF

```

Name	DSKYI1 Matrix Structure Input by Single Entry	
Purpose	This subprogram adds a single entry in the (irow , jcol) position in the lower triangle to the set of known nonzeros for the sparse matrix.	
Usage	VECLIB: <pre> INTEGER*4 irow, jcol, ier REAL*8 global(150) CALL DSKYI1(irow, jcol, global, ier) </pre>	
Input	irow	Row index of the nonzero entry; $jcol \leq irow \leq neqns$, where neqns is the number of equations specified in the call to DSKYIN. Input of diagonal entries is optional.
	jcol	Column index of the nonzero entry; $1 \leq jcol \leq neqns$, where neqns is the number of equations specified in the call to DSKYIN.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: <pre> ier = 0 Normal return. ier = -200 Incorrect processing path; DSKYIN not called or matrix input already finished. ier = -201 Error in dynamic storage allocation. ier = -202 Illegal value for jcol. ier = -203 Illegal value for irow. </pre>
Notes	Calls to DSKYI1 and DSKYIE can be intermixed. DSKYIC and DSKYIM cannot be used if DSKYI1 or DSKYIE are used.	
Example	Add the entry in row 3035, column 1024 to the list of nonzeros in the matrix.	
	<pre> INTEGER*4 IROW,JCOL,IER REAL*8 GLOBAL(150) I = 3035 J = 1024 CALL DSKYI1 (IROW,JCOL,GLOBAL,IER) IF (IER .NE. 0) THEN handle error condition ENDIF </pre>	

Name	DSKYIC Matrix Structure Input by Column	
Purpose	This subprogram adds a list of indices in a single column in the lower triangle to the set of known nonzeros for the sparse matrix.	
Usage	VECLIB: <pre> INTEGER*4 jcol, nzcol, jrowin(nzcol), ier REAL*8 global(150) CALL DSKYIC(jcol, nzcol, jrowin, global, ier) </pre>	
Input	jcol	Column index; $1 \leq \text{jcol} \leq \text{neqns}$. Columns must be presented in order from 1 to neqns , where neqns is the number of equations specified in the call to DSKYIN, except that columns with no entries can be skipped.
	nzcol	Number of nonzeros in column jcol of the matrix; nzcol ≥ 0 .
	jrowin	List of row indices for all nonzeros, in ascending order, in the lower triangular part of column jcol of the matrix; $\text{jcol} \leq \text{jrowin}(1) < \text{jrowin}(2) < \dots < \text{jrowin}(\text{nzcol}) \leq \text{neqns}$, where neqns is the number of equations specified in the call to DSKYIN. Input of diagonal entries is optional.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: <pre> ier = 0 Normal return. ier = -200 Incorrect processing path; DSKYIN not called or matrix input already finished. ier = -201 Error in dynamic storage allocation. ier = -202 Illegal value for nzcol. ier = -203 Illegal value for jcol, or out of order. </pre>

Notes The entire matrix structure must be input with DSKYIC if it is used. Its use is not compatible with DSKYE1, DSKYIE, or DSKYIM.

If the matrix entries are available by column, using DSKYIC is more efficient than using DSKYI1.

Example Column 4519 has entries in rows 1, 2735, 4519, 4520, 4521, 6000, 6002 and 6004. Add all five entries in the lower triangular part to the list of nonzeros in the matrix. Columns 1 to 4518 already have been input to the package using DSKYIC.

```
INTEGER*4 JCOL,NZCOL,JROWIN(100),IER
REAL*8    GLOBAL(150)
J = 4519
NZCOL = 5
JROWIN(1) = 4520
JROWIN(2) = 4521
JROWIN(3) = 6000
JROWIN(4) = 6002
JROWIN(5) = 6004

CALL DSKYIC (JCOL,NZCOL,JROWIN,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name	DSKYIE Matrix Structure Input by Finite Element	
Purpose	This subprogram adds indices to the set of known nonzeros in the lower triangle of the sparse matrix corresponding to a finite element or clique.	
Usage	VECLIB: <pre> INTEGER*4 nnode, nodlst(nnode), ier REAL*8 global(150) CALL DSKYIE(nnode, nodlst, global, ier) </pre>	
Input	nnode nodlst	Number of nodes in the finite element or clique. List of nodes. All pairs (nodlst(i),nodlst(j)) in the lower triangle are added to the sparsity structure of the matrix.
Updated	nodlst global	The order of the values in nodlst may be changed. Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: <pre> ier = 0 Normal return. ier = -200 Incorrect processing path; DSKYIN not called or matrix input already finished. ier = -201 Error in dynamic storage allocation. ier = -202 Illegal value for nnode. ier = -203 Illegal value for at least one entry in nodlst. </pre>
Notes	Calls to DSKYIE can be intermixed with calls to DSKYI1, DSKYIC and DSKYIM cannot be used if DSKYI1 or DSKYIE is used.	

Example Rows and columns 345, 346, 347 and 989 form a small dense submatrix of A. Add the positions consisting of all pairs of numbers from this set to the list of nonzeros in the matrix.

```
INTEGER*4 NNODE,NODLST(10),IER
REAL*8    GLOBAL(150)
NNODE = 4
NODLST(1) = 345
NODLST(2) = 346
NODLST(3) = 347
NODLST(4) = 989

CALL DSKYIE (NNODE,NODLST,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name	DSKYIM Matrix Structure Input by Matrix	
Purpose	This subprogram specifies the locations of all nonzeros in the lower triangle of the sparse matrix.	
Usage	VECLIB: INTEGER*4 colstr(neqns+1), rowind(nnzero), ier REAL*8 global(150) CALL DSKYIM(colstr, rowind, global, ier)	
Input	neqns colstr rowind	Number of equations; neqns > 0. colstr(j) gives the index in rowind of the first nonzero in the lower triangular part of column <i>j</i> of the matrix. All of the nonzeros for column <i>j</i> are found, in ascending order, in rowind(colstr(j)) , rowind(colstr(j)+1) , ..., rowind(colstr(j+1)-1) . colstr(norder+1) must be set to one greater than the total number of nonzeros in the lower triangular part of the matrix (nnzero), where norder is the matrix order as specified in the call to DSKYIN. List of row indices for all nonzeros, in ascending order within each column, in the lower triangle part of the matrix. Input of diagonal entries is optional.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: ier = 0 Normal return. ier = -200 Incorrect processing path; DSKYIN not called or matrix input already finished. ier = -201 Error in dynamic storage allocation. ier = -202 Invalid matrix structure.
Notes	This is the most efficient mechanism for specifying the nonzero structure, but the entire matrix structure must be input with DSKYIM if it is used. Its use is not compatible with DSKYI1, DSKYIE, or DSKYIC.	

Example Input the structure of this small (order 6) sparse matrix:

1.0	0.0	2.0	0.0	0.0	0.0
0.0	3.0	4.0	0.0	0.0	0.0
2.0	4.0	5.0	6.0	0.0	0.0
0.0	0.0	6.0	7.0	0.0	0.0
0.0	0.0	0.0	0.0	8.0	9.0
0.0	0.0	0.0	0.0	9.0	10.0

```
INTEGER*4 COLSTR(7), ROWIND(10), IER
REAL*8    GLOBAL(150)
```

```
COLSTR(1) = 1
COLSTR(2) = 3
COLSTR(3) = 5
COLSTR(4) = 7
COLSTR(5) = 8
COLSTR(6) = 10
COLSTR(7) = 11
```

```
ROWIND(1) = 1
ROWIND(2) = 3
ROWIND(3) = 2
ROWIND(4) = 3
ROWIND(5) = 3
ROWIND(6) = 4
ROWIND(7) = 4
ROWIND(8) = 5
ROWIND(9) = 6
ROWIND(10) = 6
```

```
CALL DSKYIM (COLSTR, ROWIND, GLOBAL, IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name	DSKYIS Matrix Structure Input by Skyline Matrix	
Purpose	This subprogram specifies the locations of all nonzeros in the upper triangle of the sparse matrix.	
Usage	VECLIB: INTEGER*4 diag(neqns+1), idata, ier REAL*8 global(150) CALL DSKYIS(diag, idata, global, ier)	
Input	diag	Array of diagonal positions. diag(i) indicates the position of the <i>i</i> th diagonal matrix element in array values . diag(neqns+1) must be set to one more than the number of values stored in array values . neqns is the number of equations specified in the call to DSKYIN.
	idata	idata = 1 Array values is in skyline format. idata = 2 Array values is in reverse skyline format.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: ier = 0 Normal return. ier = -200 Incorrect processing path; DSKYIN not called or matrix input already finished. ier = -201 Invalid data structure. ier = -202 Error in dynamic storage allocation. ier = -203 Invalid matrix structure.
Notes	This is the most efficient mechanism for specifying the nonzero structure, but the entire matrix structure must be input with DSKYIS if it is used. Its use is not compatible with DSKYI1, DSKYIE, or DSKYIC.	

Example Input the structure of this small (order 6) sparse matrix:

1.0	0.0	2.0	0.0	0.0	0.0
0.0	3.0	4.0	0.0	0.0	0.0
2.0	4.0	5.0	6.0	0.0	0.0
0.0	0.0	6.0	7.0	0.0	0.0
0.0	0.0	0.0	0.0	8.0	9.0
0.0	0.0	0.0	0.0	9.0	10.0

```
INTEGER*4 DIAG(7), IDATA, IER
REAL*8    GLOBAL(150)
```

```
DIAG(1) = 1
DIAG(2) = 3
DIAG(3) = 5
DIAG(4) = 7
DIAG(5) = 8
DIAG(6) = 10
DIAG(7) = 11
```

```
IDATA = 1
```

```
CALL DSKYIS (DIAG, IDATA, GLOBAL, IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```


Name	DSKYIF End of Matrix Structure Input	
Purpose	This subprogram specifies the end of structure input for the matrix. DSKYIF is used only if subprograms DSKYI1 and/or DSKYIE were the mechanism by which the structure was input.	
Usage	VECLIB: <pre> INTEGER*4 ier REAL*8 global(150) CALL DSKYIF(global, ier) </pre>	
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: <pre> ier = 0 Normal return. ier = -210 Incorrect processing path; DSKYIN not called or matrix input already finished. ier = -211 Error in dynamic storage allocation. </pre>
Example	The nonzero structure of a pair of finite element matrices was passed to the package using repeated calls to subroutine DSKYIE. Signal that no more nonzeros will be added to the matrix. <pre> INTEGER*4 IER REAL*8 GLOBAL(150) CALL DSKYIF (GLOBAL, IER) IF (IER .NE. 0) THEN handle error condition ENDIF </pre>	

Name	DSKYOR Automatic Reordering	
Purpose	This subprogram reorders the matrix whose pattern of nonzeros has been input to the package to obtain an efficient sparse factorization. It then constructs data structures that represent the factorization.	
Usage	VECLIB: <pre> INTEGER*4 ier REAL*8 global(150) CALL DSKYOR(global, ier) </pre>	
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: <pre> ier = 0 Normal return. ier = -300 Incorrect processing path; structure input not completed or value input subroutines already called. ier = -301 Error in dynamic storage allocation. </pre>
Example	The sparse matrix structure has been communicated to the package using DSKYIN and DSKYI1, DSKYIE, and DSKYIF, and DSKYIC or DSKYIM. The next step is obtaining good reordering for the numeric factorization phase. <pre> INTEGER*4 IER REAL*8 GLOBAL(150) CALL DSKYOR (GLOBAL, IER) IF (IER .NE. 0) THEN handle error condition ENDIF </pre>	

Name	DSKYOU User-Supplied Reordering	
Purpose	This subprogram accepts a reordering permutation provided by the user. It then constructs data structures that represent the factorization.	
Usage	VECLIB: <pre> INTEGER*4 neqns, permut(neqns), ier REAL*8 global(150) CALL DSKYOU(permut, global, ier) </pre>	
Input	permut	Permutation vector: permut (<i>i</i>) gives the index of the row and column to which the <i>i</i> th row and column are to be permuted, i.e., matrix element $A_{i,j}$ is reordered to $(PAP)_{\text{permut}(i),\text{permut}(j)}$.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: <pre> ier = 0 Normal return. ier = -300 Incorrect processing path; structure input not completed or value input subroutines already called. ier = -301 Error in dynamic storage allocation. </pre>
Example	The sparse matrix structure has been communicated to the page using DSKYIN and DSKYI1, DSKYIE, and DSKYIF, and DSKYIC or DSKYIM. The next step is obtaining good reordering for the numeric factorization phase.	

```

      INTEGER*4 IER
      REAL*8    GLOBAL(150)
      CALL DSKYOU (PERMUT, GLOBAL, IER)
      IF ( IER .NE. 0 ) THEN
         handle error condition
      ENDIF

```

Name	DSKYV1 Matrix Value Input by Single Entry	
Purpose	This subprogram adds to the value of the entry in the (irow, jcol) position in the lower triangle of the sparse matrix.	
Usage	VECLIB: <pre> INTEGER*4 irow, jcol, ier REAL*8 value, global(150) CALL DSKYV1(irow, jcol, value, global, ier) </pre>	
Input	irow	Row index of the nonzero entry, $jcol \leq irow \leq neqns$, where neqns is the number of equations specified in the call to DSKYIN.
	jcol	Column index of the nonzero entry, $1 \leq jcol \leq neqns$, where neqns is the number of equations specified in the call to DSKYIN.
	value	Numeric value that will be added to any previous values input for this location.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response:
	ier = 0	Normal return.
	ier = -400	Incorrect processing path; DSKYOR or DSKYOU not called.
	ier = -401	Error in dynamic storage allocation.
	ier = -402	Subscript pair (irow, jcol) was not specified in structure input. No room for value.
	ier = -403	Error in input total.
Notes	Calls to DSKYV1, DSKYVC, DSKYVE, and DSKYVM can be intermixed.	

Example Store the value 4.523×10^{-5} as the nonzero entry in row 3035, column 1024 of the matrix.

```
INTEGER*4 IROW,JCOL,IER
REAL*8    VALUE,GLOBAL(150)
IROW = 3035
JCOL = 1024
VALUE = 4.523D-5
CALL DSKYV1 (IROW,JCOL,VALUE,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name	DSKYVC Matrix Value Input by Column	
Purpose	This subprogram adds to the values of a list of nonzero entries in the lower triangle of a single column of the sparse matrix.	
Usage	VECLIB: <pre> INTEGER*4 jcol, nzcol, jrowin(nzcol), ier REAL*8 values(nzcol), global(150) CALL DSKYVC(jcol, nzcol, jrowin, values, global, ier) </pre>	
Input	jcol	Column index; $1 \leq \text{jcol} \leq \text{neqns}$, where neqns is the number of equations specified in the call to DSKYIN.
	nzcol	Number of nonzeros in the list of nonzeros for column jcol of the matrix; $\text{nzcol} \geq 0$.
	jrowin	List of ascending order row indices for nonzeros in the lower triangular part of column jcol of the matrix; $\text{jcol} \leq \text{jrowin}(1) < \text{jrowin}(2) < \dots < \text{jrowin}(\text{nzcol}) \leq \text{neqns}$, where neqns is the number of equations specified in the call to DSKYIN.
	values	List of values corresponding to the positions specified by jcol and jrowin .
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: <pre> ier = 0 Normal return. ier = -400 Incorrect processing path; DSKYOR or DSKYOU not called. ier = -401 Error in dynamic storage allocation. ier = -402 Error in matrix input. ier = -403 Error in input total. </pre>
Notes	Calls to DSKYV1, DSKYVC, DSKYVE and DSKYVM can be intermixed. It is not necessary that all entries in the column jcol be included in jrowin . If the matrix entries are available by column, using DSKYVC is more efficient than using DSKYV1.	

Example Column 4519 has entries in rows 1, 2735, 4519, 4520, 4521, 6000, 6002, and 6004. Add the value 1.0 to each of these positions in the matrix.

```
INTEGER*4 JCOL,NZCOL,JROWIN(100),IER
REAL*8    VALUES(100),GLOBAL(150)
J = 4519
NZCOL = 6
JROWIN(1) = 4519
JROWIN(2) = 4520
JROWIN(3) = 4521
JROWIN(4) = 6000
JROWIN(5) = 6002
JROWIN(6) = 6004

VALUES(1) = 1.0
VALUES(2) = 1.0
VALUES(3) = 1.0
VALUES(4) = 1.0
VALUES(5) = 1.0
VALUES(6) = 1.0

CALL DSKYVC (JCOL,NZCOL,JROWIN,VALUES,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name	DSKYVE Matrix Value Input by Finite Element	
Purpose	This subprogram adds to the values of a set of nonzero entries in the lower triangle of the sparse matrix corresponding to a finite element or clique.	
Usage	VECLIB: <pre> INTEGER*4 nnode, ldelmx, nodlst(nnode), ier REAL*8 elmmtx(ldelmx, nnode), global(150) CALL DSKYVE(nnode, nodlst, elmmtx, ldelmx, global, ier) </pre>	
Input	nnode	Number of nodes in the finite element or clique.
	nodlst	List of nodes in element or clique. Values for all pairs (nodlst(i) , nodlst(j)) are added to the values of the matrix.
	elmmtx	Array containing values to be added to the matrix. Only the lower triangle (including the diagonal) of elmmtx is referenced. The value in elmmtx(k, 1) is added to the value in position (nodlst(k) , nodlst(1)) in the sparse matrix.
	ldelmx	The leading dimension of array elmmtx as declared in the calling program unit, with ldelmx $\geq$ nnode .
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: ier = 0 Normal return. ier = -400 Incorrect processing path; neither DSKYOR nor DSKYOU called. ier = -401 Error in dynamic storage allocation. ier = -402 Error in matrix input.
Notes	Calls to DSKYV1, DSKYVC, DSKYVE, and DSKYVM can be intermixed. Using DSKYVE is more efficient in finite element contexts, where the matrix is created as the sum of elemental matrices, if the user has stored the elemental matrices rather than the assembled matrix.	

Example Rows and columns 345, 346, 347, and 989 form a small dense submatrix of A. Add the value 1.0 to the values in the positions consisting of all pairs of numbers from this set to the list of nonzeros in the matrix.

```

      INTEGER*4 NNODE,NODLST(10),IER
      REAL*8    ELMMTX(10,10),GLOBAL(150)
      NNODE = 4
      NODLST(1) = 345
      NODLST(2) = 346
      NODLST(3) = 347
      NODLST(4) = 989
      DO 200 K = 1, NNODE
        DO 100 L = K, NNODE
          ELMMTX(K,L) = 1.0D0
100      CONTINUE
200 CONTINUE
      CALL DSKYVE (NNODE,NODLST,ELMMTX,10,GLOBAL,IER)
      IF ( IER .NE. 0 ) THEN
        handle error condition
      ENDIF

```

Name	DSKYVM Matrix Value Input by Matrix	
Purpose	This subprogram adds to the values of many or all of the nonzero entries in the lower triangle of the sparse matrix.	
Usage	VECLIB: <pre> INTEGER*4 colstr(neqns+1), rowind(nnzero), ier REAL*8 values(nnzero), global(150) CALL DSKYVM(colstr, rowind, values, global, ier) </pre>	
Input	colstr	colstr(j) gives the index in rowind of the first nonzero in the lower triangular part of column <i>j</i> of the matrix. All of the nonzeros for column <i>j</i> are found, in ascending order, in rowind(colstr(j)), rowind(colstr(j)+1), ..., rowind(colstr(j+1)-1). colstr(norder+1) must be set to one greater than the total number of nonzeros in the lower triangular part of the matrix (nnzero), where norder is the matrix order as specified in the call to DSKYIN.
	rowind	List of row indices for all nonzeros in ascending order within each column in the lower triangle part of the matrix. Input of diagonal entries is optional.
	values	List of values corresponding in position to the indices in rowind . These values will be added to any values already present in the matrix.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: <pre> ier = 0 Normal return. ier = -400 Incorrect processing path; DSKYOR or DSKYOU not called. ier = -401 Error in dynamic storage allocation. ier = -402 Error in input total. </pre>
Notes	This is the most efficient mechanism for specifying the nonzero values. Normally, DSKYVM is used in conjunction with DSKYIM. Additional entries or modifications can be entered with DSKYV1, DSKYVC, or DSKYVE.	

Example Input the values for the following small (order 6) sparse matrix problem.

4.0	0.0	1.0	0.0	0.0	0.0
0.0	4.0	1.0	0.0	1.0	0.0
1.0	1.0	4.0	1.0	1.0	0.0
0.0	0.0	1.0	4.0	1.0	0.0
0.0	1.0	1.0	1.0	4.0	0.0
0.0	0.0	0.0	0.0	0.0	4.0

```

INTEGER*4 COLSTR(7), ROWIND(12), IER
REAL*8    VALUES(12), GLOBAL(150)

```

```

COLSTR(1) = 1
COLSTR(2) = 3
COLSTR(3) = 6
COLSTR(4) = 9
COLSTR(5) = 11
COLSTR(6) = 12
COLSTR(7) = 13

```

```

ROWIND(1) = 1
ROWIND(2) = 3
ROWIND(3) = 2
ROWIND(4) = 3
ROWIND(5) = 5
ROWIND(6) = 3
ROWIND(7) = 4
ROWIND(8) = 5
ROWIND(9) = 4
ROWIND(10) = 5
ROWIND(11) = 5
ROWIND(12) = 6

```

```

VALUES(1) = 4.0
VALUES(2) = 1.0
VALUES(3) = 4.0
VALUES(4) = 1.0
VALUES(5) = 1.0
VALUES(6) = 4.0
VALUES(7) = 1.0
VALUES(8) = 1.0
VALUES(9) = 4.0
VALUES(10) = 1.0
VALUES(11) = 4.0
VALUES(12) = 4.0

```

```

CALL DSKYVM (COLSTR,ROWIND,VALUES,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF

```

Name	DSKYVS Matrix Value Input by Skyline Matrix	
Purpose	This subprogram adds to the values of many or all of the nonzero entries in the upper triangle of the sparse matrix.	
Usage	VECLIB: <pre> INTEGER*4 diag(neqns+1), idata, ier REAL*8 values(nnzero), global(150) CALL DSKYVS(values, diag, idata, global, ier) </pre>	
Input	values	Array of matrix values in skyline or reverse skyline order.
	diag	Array of diagonal positions. diag(<i>i</i>) indicates the position of the <i>i</i> th diagonal matrix element in array values . diag(neqns+1) must be set to one more than the number of values stored in array values .
	idata	Specifies the skyline data structure format as follows: idata = 1 Array values is in skyline format. idata = 2 Array values is in reverse skyline format.
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: ier = 0 Normal return. ier = -400 Incorrect processing path; DSKYOR or DSKYOU not called. ier = -401 Error in dynamic storage allocation. ier = -402 Error in input total.
Notes	This is the most efficient mechanism for specifying the nonzero values. DSKYVS is used in conjunction with DSKYIS.	

Example Input the values for the following small (order 6) sparse matrix problem.

4.0	0.0	1.0	0.0	0.0	0.0
0.0	4.0	1.0	0.0	1.0	0.0
1.0	1.0	4.0	1.0	1.0	0.0
0.0	0.0	1.0	4.0	1.0	0.0
0.0	1.0	1.0	1.0	4.0	0.0
0.0	0.0	0.0	0.0	0.0	4.0

Use DSKYVS and DSKYV1 to input this matrix.

```

INTEGER*4 DIAG(7),IDATA, IER
REAL*8     VALUES(12), GLOBAL(150)

IDATA = 1

DIAG(1) = 1
DIAG(2) = 2
DIAG(3) = 5
DIAG(4) = 7
DIAG(5) = 11
DIAG(6) = 12
DIAG(7) = 13

VALUES(1) = 4.0
VALUES(2) = 4.0
VALUES(3) = 1.0
VALUES(4) = 1.0
VALUES(5) = 4.0
VALUES(6) = 1.0
VALUES(7) = 4.0
VALUES(8) = 1.0
VALUES(9) = 1.0
VALUES(10) = 1.0
VALUES(11) = 4.0
VALUES(12) = 4.0

CALL DSKYVS (VALUES,DIAG,IDATA,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF

```

Name	DSKYFA Numeric Factorization	
Purpose	This subprogram computes the numeric factorization of the sparse symmetric matrix input to the package. No condition number estimation is performed.	
Usage	VECLIB: INTEGER*4 ier REAL*8 global(150) CALL DSKYFA(global, ier)	
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: ier = 0 Normal return. ier = -500 Incorrect processing path; value input not performed. ier = -501 Matrix is singular. ier = -502 Dynamic allocation error.
Example	Factor the matrix input to the package. INTEGER*4 IER REAL*8 GLOBAL(150) CALL DSKYFA (GLOBAL, IER) IF (IER .NE. 0) THEN handle error condition ENDIF	

Name	DSKYSL Solve Right-Hand Side Vector	
Purpose	This subprogram computes the solution for a right-hand side vector given a numeric factorization performed by DSKYFA.	
Usage	VECLIB: INTEGER*4 ier REAL*8 rhs(neqns), global(150) CALL DSKYSL(rhs, global, ier)	
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
	rhs	On input, rhs contains the right-hand side. On output, rhs has been overwritten with the computed solution. neqns is the number of equations specified in the call to DSKYIN.
Output	ier	Status response: ier = 0 Normal return. ier = -600 Incorrect processing path. ier = -601 Dynamic storage allocation error.

Example Given the following right-hand side vector, compute the solution given the numerical factorization computed by DSKYFA.

```

                                1.0
                                0.0
                                1.0
rhs =                          1.0
                                0.0
                                0.0
```

```
INTEGER*4 IER
REAL*8    RHS(6),GLOBAL(150)
```

```
RHS(1) = 1.0
RHS(2) = 0.0
RHS(3) = 1.0
RHS(4) = 1.0
RHS(5) = 0.0
RHS(6) = 0.0
```

```
CALL DSKYSL (RHS,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```


Name	DSKYDF Numeric Factorization	
Purpose	This subprogram computes the numeric factorization of the sparse symmetric matrix without using the matrix structure input, reordering, or matrix value input facilities of the package. It is intended to be used as a replacement for a skyline solver in an existing program. No condition number estimation is performed.	
Usage	VECLIB: INTEGER*4 neqns, nnzero, diag(neqns+1), idata, ier REAL*8 global(150), values(nnzero) CALL DSKYDF(values, diag, idata, global, ier)	
Input	neqns	Number of equations; neqns > 0.
	diag	Array of diagonal positions. diag(i) indicates the position of the <i>i</i> th diagonal matrix element in array values . diag(neqns+1) must be set to one more than the number of values stored in array values . neqns is the number of equations specified in the call to DSKYIN.
	idata	idata = 1 Array values is in skyline format. idata = 2 Array values is in reverse skyline format.
Updated	values	On input array of matrix values in skyline or reverse skyline order. On output, the input has been overwritten by the factorization.
	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: ier = 0 Normal return. ier = -500 Incorrect processing path; DSKYIN not called. ier = -501 Incorrect data structure. ier = -502 Matrix is singular. ier = -503 Dynamic allocation error.

Example Solve the small (order 6) sparse system of linear equations where the matrix A and right-hand side b are

$$A = \begin{bmatrix} 4.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 4.0 & 1.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 1.0 & 4.0 & 1.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 4.0 & 1.0 & 1.0 \\ 0.0 & 1.0 & 1.0 & 1.0 & 4.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 4.0 \end{bmatrix} \quad b = \begin{bmatrix} 1.0 \\ 0.0 \\ 1.0 \\ 1.0 \\ 0.0 \\ 0.0 \end{bmatrix}$$

```

INTEGER*4  DIAG(7)
REAL*8     VALUES(14),RHS(6),GLOBAL(150)
DIAG(1) = 1
DIAG(2) = 2
DIAG(3) = 5
DIAG(4) = 7
DIAG(5) = 11
DIAG(6) = 14
DIAG(7) = 15

VALUES(1) = 4.0
VALUES(2) = 4.0
VALUES(3) = 1.0
VALUES(4) = 1.0
VALUES(5) = 4.0
VALUES(6) = 1.0
VALUES(7) = 4.0
VALUES(8) = 1.0
VALUES(9) = 1.0
VALUES(10) = 1.0
VALUES(11) = 4.0
VALUES(12) = 1.0
VALUES(13) = 0.0
VALUES(14) = 4.0

RHS(1) = 1.0
RHS(2) = 0.0
RHS(3) = 1.0
RHS(4) = 1.0
RHS(5) = 0.0
RHS(6) = 0.0

CALL DSKYDF (VALUES, DIAG, IDATA, GLOBAL, IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF

```

Name	DSKYDS Solve Right-Hand Side Vector	
Purpose	This subprogram computes the solution for a right-hand side vector given a numeric factorization performed by DSKYDF.	
Usage	VECLIB: <pre> INTEGER*4 neqns, nnzero, diag(neqns+1), idata, ier REAL*8 values(nnzero), rhs(neqns), global(150) CALL DSKYDS(values, diag, rhs, idata, global, ier) </pre>	
Input	values	Array of matrix values from DSKYDF.
	diag	Array of diagonal positions. diag (<i>i</i>) indicates the position of the <i>i</i> th diagonal matrix element in array values . diag (neqns +1) must be set to one more than the number of values stored in array values . neqns is the number of equations specified in the call to DSKYIN.
	idata	idata = 1 Array values is in skyline format. idata = 2 Array values is in reverse skyline format.
Updated	rhs	On input, rhs contains the right-hand side. On output, rhs has been overwritten with the computed solution. neqns is the number of equations specified in the call to DSKYIN.
	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: ier = 0 Normal return. ier = -600 Incorrect processing path. ier = -601 Incorrect data structure.

Example Given the following right-hand side vector, compute the solution given the numerical factorization computed by DSKYDS.

 1.0
 0.0
 1.0
 b = 1.0
 0.0
 0.0

```
INTEGER*4 DIAG(7), IER
REAL*8     VALUES(13), RHS(6), GLOBAL(150)

RHS(1) = 1.0
RHS(2) = 0.0
RHS(3) = 1.0
RHS(4) = 1.0
RHS(5) = 0.0
RHS(6) = 0.0

CALL DSKYDS (VALUES, DIAG, RHS, 1, GLOBAL, IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF
```

Name	DSKYDA Deallocate Working Storage	
Purpose	Deallocate working storage at the end of processing. If the program using the package is to continue execution after use of the package is completed, the user should deallocate the dynamically-allocated working storage with subroutine DSKYDA.	
Usage	VECLIB: <pre> INTEGER*4 ier REAL*8 global(150) CALL DSKYDA(global, ier) </pre>	
Updated	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: <pre> ier = 0 Normal return. ier = -701 Error in dynamic storage deallocation. </pre>
Example	Deallocate working storage after use of package. <pre> REAL*8 GLOBAL(150) CALL DSKYDA (GLOBAL, IER) IF (IER .NE. 0) THEN handle error condition ENDIF </pre>	

Name	DSKYOC Output Control	
Purpose	This subprogram may be called at any point after subprogram DSKYIN to alter either the output message level or the Fortran output unit number for message output.	
Usage	VECLIB: <pre> INTEGER*4 msglvl, output REAL*8 global(150) CALL DSKYOC(msglvl, output, global) </pre>	
Input	<pre> msglvl </pre>	Message level for printable output: <pre> msglvl ≤ 0 Suppress all output. msglvl = 1 Error messages and summary statistics. msglvl = 2 More complete statistics. msglvl = 3 First stage of debugging output. msglvl = 4 Complete debugging output. </pre>
	<pre> output</pre>	Fortran logical unit number to which all output will be written.
Updated	<pre> global</pre>	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Example	Increase message level from 1 to 2 while leaving the output unit at 6. <pre> REAL*8 GLOBAL(150) CALL DSKYOC (2,6,GLOBAL) </pre>	

Name	DSKYPS Print Statistics
Purpose	This subprogram prints available statistics about the original matrix, amount of work space in use, amount required for the next phase, and maximum amount used thus far. Also, this subprogram prints storage and arithmetic requirements, CPU time used, and computational rate for Cholesky factorization. The amount of information printed depends on the stage of execution. The number of lines of output ranges from 8 to 30, with the width of the lines being less than 80 characters.
Usage	VECLIB: REAL*8 global(150) CALL DSKYPS(global)
Input	global Global communications array for this problem.
Example	Print the statistics after the package has completed a numeric solution (either DSKYSL or DSKYFS). REAL*8 GLOBAL(150) CALL DSKYPS (GLOBAL)

Name	DSKYRS Restore Problem State from a Savefile	
Purpose	This subprogram restores the working problem from the state stored on the user-specified I/O file by subprogram DSKYSV.	
Usage	VECLIB: <pre> INTEGER*4 svfile, ier REAL*8 global(150) CALL DSKYRS(svfile, global, ier) </pre>	
Input	svfile	Fortran logical unit number of a file containing the saved problem state as created by DSKYSV.
Output	global	Global communications array restored from svfile.
	ier	Status response: ier = 0 Normal return. ier = -901 Error in dynamic storage allocation. ier = -902 I/O error detected by Fortran.
Example	Restart the solution process from the save file on Fortran logical unit 42 created by subroutine DSKYSV. <pre> REAL*8 GLOBAL(150) CALL DSKYRS (42,GLOBAL,IER) IF (IER .NE. 0) THEN handle error condition ENDIF </pre>	

Name	DSKYSR Retrieve Runtime Statistics	
Purpose	This subprogram retrieves certain runtime statistics from the global communications array.	
Usage	VECLIB: INTEGER*4 inuse, mxused REAL*8 global(150), time(6), opcnts(2) CALL DSKYSR(global, inuse, mxused, time, opcnts)	
Input	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	inuse	Amount of dynamically-allocated storage currently in use in units of REAL*8 words.
	mxused	Maximum amount of dynamically allocated storage allocated in units of REAL*8 words.
	time	Array which, on return, holds the CPU time spent in matrix structure input, ordering, symbolic factorization, value input, numeric factorization, and numeric solution, respectively, in seconds.
	opcnts	Array which, on return, holds the operation counts for the numeric factorization and numeric solution, respectively.
Notes	The time reported for numeric solution is the time for a single right-hand side. If a phase has been repeated, time for the last repetition is reported. Time for structure and value input includes all the time, including the user's, from the start of input until it is completed.	
Example	Retrieve the statistics after the package has completed a numeric solution (either DSKYSL or DSKYFS). <pre> INTEGER*4 INUSE, MXUSED, OPCNTS(2) REAL*8 GLOBAL(150), TIME(6) CALL DSKYSR (GLOBAL, INUSE, MXUSED, TIME, OPCNTS) </pre>	

Name	DSKYSV Save Problem State to a Savefile	
Purpose	This subprogram saves the current problem for restarting later at the current state. The global communication array and all working storage are written onto the user-specified I/O file using standard Fortran unformatted sequential write statements. The file is rewound before and after use.	
Usage	VECLIB: <pre> INTEGER*4 svfile, ier REAL*8 global(150) CALL DSKYSV(svfile, global, ier) </pre>	
Input	svfile	Fortran logical unit number of the file onto which the state is to be saved. The file will be rewound before and after use.
	global	Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.
Output	ier	Status response: <pre> ier = 0 Normal return. ier = -902 I/O error detected by Fortran. </pre>
Example	Save the current state on Fortran logical unit 42.	

```

REAL*8 GLOBAL(150)
CALL DSKYSV (42,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
ENDIF

```

9 Fast Fourier Transforms

Overview

This chapter explains how to use the VECLIB fast Fourier transform (FFT) subprograms. The operations covered are

- One-, two-, and three-dimensional complex-to-complex FFT subprograms
- One-, two-, and three-dimensional real-to-complex FFT subprograms
- One-dimensional simultaneous complex-to-complex FFT subprograms
- One-dimensional simultaneous real-to-complex FFT subprograms

The following documents provide supplemental material for this chapter:

Brigham, E.O. *The Fast Fourier Transform*. Englewood Cliffs, NJ: Prentice-Hall, Inc. 1974.

Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc. 1975.

Van Loan, C. *Computational Frameworks for the Fast Fourier Transform*. Philadelphia: Society for Industrial and Applied Mathematics, 1992.

Chapter Objectives

After reading this chapter you will

- Understand the VECLIB 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 VECLIB FFT subprograms restrict the number of points to certain forms. The number of points in each direction must be a product of powers of two, three, and five:

$$l_k = 2^{p_k} 3^{q_k} 5^{r_k}, \quad p_k, q_k, r_k \geq 0.$$

While this restriction may limit the utility of the subprograms, the gain in speed is enormous. You can frequently adapt your data set to the VECLIB FFT subprograms by augmenting it with enough zero-value data points to reach the next acceptable number of points. Doing so slightly changes the problem, which may or may not be important, 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.

Subprograms Included in This Chapter

Following are the FFT subprograms included with VECLIB.

Name C1DFFT/Z1DFFT
One-Dimensional FFT

Purpose Given an array of complex data, these subprograms compute the one-dimensional forward or inverse DFT using a radix 2-3-5 FFT algorithm. Two companion subprograms, S1DFFT and D1DFFT, perform the same operation but with the complex data presented with real and imaginary parts in separate real arrays.

The one-dimensional forward DFT of $z(n)$, for $n = 1, 2, \dots, l$, is defined by

$$Z(m) = \sum_{n=1}^l z(n) e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, \dots, l$ and $i = \sqrt{-1}$.

Alternatively, the one-dimensional scaled inverse DFT of $Z(m)$, for $m = 1, 2, \dots, l$, is defined by

$$z(n) = \frac{1}{l} \sum_{m=1}^l Z(m) e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, \dots, l$.

Finally, the one-dimensional unscaled inverse DFT of $Z(m)$, for $m = 1, 2, \dots, l$, is defined by

$$z(n) = \sum_{m=1}^l Z(m) e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, \dots, l$.

These subprograms require that l be a product of powers of 2, 3, and 5, i.e., of the form

$$l = 2^p 3^q 5^r$$

where $p, q, r \geq 0$

Usage Because it is common to use one data set length repetitively, these subprograms have a separate initialization call such that the setup can be performed only once for each different transform size. You will, therefore, always have at least two **CALL** statements to the FFT subprogram using the same working storage array. Refer to "Example."

VECLIB:

```

      INTEGER*4      l, iopt, ier
      COMPLEX*8      z(l)
      REAL*4         work(5*1/2)
      CALL C1DFFT(z, l, work, iopt, ier)

```

```

      INTEGER*4      l, iopt, ier
      COMPLEX*16     z(l)
      REAL*8         work(5*1/2)
      CALL Z1DFFT(z, l, work, iopt, ier)

```

VECLIB8:

```

      INTEGER*8      l, iopt, ier
      COMPLEX*16     z(l)
      REAL*8         work(5*1/2)
      CALL C1DFFT(z, l, work, iopt, ier)

```

Input	z	Array of data to be transformed. Not used if iopt = -3.
	l	Number of data points, of the form $l = 2^p 3^q 5^r$, with $p, q, r \geq 0$.
	iopt	Option flag:
	iopt = +1	Compute forward transform.
	iopt = -1	Compute scaled inverse transform.
Working Storage	iopt = -2	Compute unscaled inverse transform.
	iopt = -3	Initialize work for subsequent transforms of length l .
	work	If iopt = -3, work is initialized for computing transforms of length l . If iopt $\neq$ -3, work must have been initialized by a previous call with this value of l in which iopt was -3.
Output	z	If iopt $\neq$ -3, transformed data replaces the input if ier = 0 is returned. Not used as output if iopt = -3.

ier	Status response:
ier = 0	Normal return—transform or initialization successful.
ier = -1	$l < 0$.
ier = -2	l not of the required form.
ier = -3	Invalid value of iopt .
ier = -4	Insufficient dynamic memory available for workspace.

Example Compute the forward discrete Fourier transform of two COMPLEX*8 data sets of length 1024. Here, length of working storage is $5 \times 1024/2 = 2560$.

```

      INTEGER*4 L, IOPT, IER
      COMPLEX*8 Z1(1024), Z2(1024)
      REAL*4    WORK(5*1024/2)
      L = 1024
      IOPT = -3
      CALL C1DFFT (Z1, L, WORK, IOPT, IER)      ! INITIALIZE
      IF ( IER .NE. 0 ) THEN
         PRINT *, 'IER = ', IER, ' FROM C1DFFT.'
         STOP
      END IF
      CALL C1DFFT (Z1, L, WORK, IOPT, IER)      ! FIRST TRANSFORM
      IF ( IER .NE. 0 ) THEN
         PRINT *, 'IER = ', IER, ' FROM C1DFFT.'
         STOP
      END IF
      CALL C1DFFT (Z2, L, WORK, IOPT, IER)      ! SECOND TRANSFORM
      IF ( IER .NE. 0 ) THEN
         PRINT *, 'IER = ', IER, ' FROM C1DFFT.'
         STOP
      END IF

```

Name S1DFFT/D1DFFT
One-Dimensional FFT

Purpose Given a set of complex data with the real and imaginary parts in separate real arrays, these subprograms compute the one-dimensional forward or inverse DFT using a radix 2-3-5 FFT algorithm. Two companion subprograms, C1DFFT and Z1DFFT, perform the same operation but with the complex data presented in a complex array.

The one-dimensional forward discrete Fourier transform of $z(n)$, for $n = 1, 2, \dots, l$, is defined by

$$Z(m) = \sum_{n=1}^l z(n) e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, \dots, l$ and $i = \sqrt{-1}$.

Alternatively, the one-dimensional scaled inverse DFT of $Z(m)$, for $m = 1, 2, \dots, l$, is defined by

$$z(n) = \frac{1}{l} \sum_{m=1}^l Z(m) e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, \dots, l$.

Finally, the one-dimensional unscaled inverse DFT of $Z(m)$, for $m = 1, 2, \dots, l$, is defined by

$$z(n) = \sum_{m=1}^l Z(m) e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, \dots, l$.

These subprograms require that l be a product of power of 2, 3 and 5, i.e., of the form

$$l = 2^p 3^q 5^r$$

with $p, q, r \geq 0$

The complex data, z or Z , are stored with real and imaginary parts in separate real arrays, x and y , respectively.

Usage Because it is common to use one data set length repetitively, these subprograms have a separate initialization call such that the setup can be performed only once for each different transform size. You will, therefore, always have at least two CALL statements to the FFT subprogram using the same working storage array. Refer to "Example."

VECLIB:

```

      INTEGER*4      l, iopt, ier
      REAL*4         x(l), y(l), work(5*1/2)
      CALL S1DFFT(x, y, l, work, iopt, ier)

      INTEGER*4      l, iopt, ier
      REAL*8         x(l), y(l), work(5*1/2)
      CALL D1DFFT(x, y, l, work, iopt, ier)

```

VECLIB8:

```

      INTEGER*8      l, iopt, ier
      REAL*8         x(l), y(l), work(5*1/2)
      CALL S1DFFT(x, y, l, work, iopt, ier)

```

Input	x	Array of real parts of the data to be transformed. Not used if iopt = -3.							
	y	Array of imaginary parts of the data to be transformed. Not used if iopt = -3.							
	l	Number of data points, of the form $l = 2^p 3^q 5^r$, with $p, q, r \geq 0$.							
	iopt	Option flag:							
		<table><tr><td>iopt = +1</td><td>Compute forward transform.</td></tr><tr><td>iopt = -1</td><td>Compute scaled inverse transform.</td></tr><tr><td>iopt = -2</td><td>Compute unscaled inverse transform.</td></tr><tr><td>iopt = -3</td><td>Initialize work for subsequent transforms of length l.</td></tr></table>	iopt = +1	Compute forward transform.	iopt = -1	Compute scaled inverse transform.	iopt = -2	Compute unscaled inverse transform.	iopt = -3
iopt = +1	Compute forward transform.								
iopt = -1	Compute scaled inverse transform.								
iopt = -2	Compute unscaled inverse transform.								
iopt = -3	Initialize work for subsequent transforms of length l .								
Working Storage	work	If iopt = -3, work is initialized for computing transforms of length l. If iopt $\neq$ -3, work must have been initialized by a previous call with this value of l in which iopt was -3.							
Output	x and y	If iopt $\neq$ -3, the transformed data replaces the input if ier = 0 is returned. Not used as output if iopt = -3.							

ier	Status response:
ier = 0	Normal return—transform or initialization successful.
ier = -1	$l < 0$.
ier = -2	l not of the required form.
ier = -3	Invalid value of iopt .
ier = -4	Insufficient dynamic memory available for workspace.

Example Compute the forward discrete Fourier transform of two REAL*8 data sets of length 512. Here, the length of the working storage is $5 \times 512/2 = 1280$.

```

      INTEGER*4 L, IOPT, IER
      REAL*8    X1(512), Y1(512), X2(512), Y2(512), WORK(5*512/2)
      L = 512
      IOPT = -3
      CALL D1DFFT (X1,Y1,L,WORK,IOPT,IER)    ! INITIALIZE
      IF ( IER .NE. 0 ) THEN
        PRINT *, 'IER = ', IER, ' FROM D1DFFT.'
        STOP
      END IF
      IOPT = 1
      CALL D1DFFT (X1,Y1,L,WORK,IOPT,IER)    ! FIRST TRANSFORM
      IF ( IER .NE. 0 ) THEN
        PRINT *, 'IER = ', IER, ' FROM D1DFFT.'
        STOP
      END IF
      CALL D1DFFT (X2,Y2,L,WORK,IOPT,IER)    ! SECOND TRANSFORM
      IF ( IER .NE. 0 ) THEN
        PRINT *, 'IER = ', IER, ' FROM D1DFFT.'
        STOP
      END IF

```

Name C2DFFT/Z2DFFT
Two-Dimensional FFT

Purpose Given an array of complex data, these subprograms compute the two-dimensional forward or inverse DFT using a radix 2-3-5 FFT algorithm. Two companion subprograms, S2DFFT and D2DFFT, perform the same operation but with the complex data presented with real and imaginary parts in separate real arrays.

The two-dimensional forward discrete Fourier transform of $z(n_1, n_2)$, for $n_1 = 1, 2, \dots, l_1$ and $n_2 = 1, 2, \dots, l_2$, is defined by

$$Z(m_1, m_2) = \sum_{n_1=1}^{l_1} \sum_{n_2=1}^{l_2} z(n_1, n_2) e^{-2\pi i(m_1-1)(n_1-1)/l_1} e^{-2\pi i(m_2-1)(n_2-1)/l_2}$$

for $m_1 = 1, 2, \dots, l_1$, $m_2 = 1, 2, \dots, l_2$, and $i = \sqrt{-1}$.

Alternatively, the two-dimensional inverse discrete Fourier transform of $Z(m_1, m_2)$, for $m_1 = 1, 2, \dots, l_1$ and $m_2 = 1, 2, \dots, l_2$, is defined by

$$z(n_1, n_2) = \frac{1}{l_1 l_2} \sum_{m_1=1}^{l_1} \sum_{m_2=1}^{l_2} Z(m_1, m_2) e^{+2\pi i(m_1-1)(n_1-1)/l_1} e^{+2\pi i(m_2-1)(n_2-1)/l_2}$$

for $n_1 = 1, 2, \dots, l_1$ and $n_2 = 1, 2, \dots, l_2$.

These subprograms require that l_1 and l_2 be products of powers of 2, 3, and 5, i.e., of the form

$$l_k = 2^{p_k} 3^{q_k} 5^{r_k},$$

where $p_k, q_k, r_k \geq 0$, $k=1,2$.

Usage VECLIB:

```

      INTEGER*4      11, 12, ldz, iopt, ier
      COMPLEX*8      z(ldz, 12)
      CALL C2DFFT(z, 11, 12, ldz, iopt, ier)

      INTEGER*4      11, 12, ldz, iopt, ier
      COMPLEX*16     z(ldz, 12)
      CALL Z2DFFT(z, 11, 12, ldz, iopt, ier)
```

VECLIB8:

```

      INTEGER*8      l1, l2, ldz, iopt, ier
      COMPLEX*16     z(ldz, l2)
      CALL C2DFFT(z, l1, l2, ldz, iopt, ier)

```

Input	z	Array of data to be transformed.
	l1	Number of rows of data, of the form $l1 = 2^{p_1} 3^{q_1} 5^{r_1}$, with $p_1, q_1, r_1 \geq 0$.
	l2	Number of columns of data, of the form $l2 = 2^{p_2} 3^{q_2} 5^{r_2}$, with $p_2, q_2, r_2 \geq 0$.
	ldz	The leading dimension of array z, with $ldz \geq l1$.
	iopt	Option flag: iopt ≥ 0 Compute forward transform. iopt < 0 Compute inverse transform.
Output	z	The transformed data replaces the input if ier = 0 is returned.
	ier	Status response: ier = 0 Normal return—transform successful. ier = -1 l1 not of the required form. ier = -2 l2 not of the required form. ier = -3 ldz < l1. ier ≤ -4 Probable error in ldz or dimensions of z.

Name S2DFFT/D2DFFT
Two-Dimensional FFT

Purpose Given a set of complex data with the real and imaginary parts in separate real arrays, these subprograms compute the two-dimensional forward or inverse DFT using a radix 2-3-5 FFT algorithm. Two companion subprograms, C2DFFT and Z2DFFT, perform the same operation but with the complex data presented in a complex array.

The two-dimensional forward DFT of $z(n_1, n_2)$, for $n_1 = 1, 2, \dots, l_1$ and $n_2 = 1, 2, \dots, l_2$, is defined by

$$Z(m_1, m_2) = \sum_{n_1=1}^{l_1} \sum_{n_2=1}^{l_2} z(n_1, n_2) e^{-2\pi i(m_1-1)(n_1-1)/l_1} e^{-2\pi i(m_2-1)(n_2-1)/l_2}$$

for $m_1 = 1, 2, \dots, l_1$, $m_2 = 1, 2, \dots, l_2$, and $i = \sqrt{-1}$.

Alternatively, the two-dimensional inverse DFT of $Z(m_1, m_2)$, for $m_1 = 1, 2, \dots, l_1$ and $m_2 = 1, 2, \dots, l_2$, is defined by

$$z(n_1, n_2) = \frac{1}{l_1 l_2} \sum_{m_1=1}^{l_1} \sum_{m_2=1}^{l_2} Z(m_1, m_2) e^{+2\pi i(m_1-1)(n_1-1)/l_1} e^{+2\pi i(m_2-1)(n_2-1)/l_2}$$

for $n_1 = 1, 2, \dots, l_1$ and $n_2 = 1, 2, \dots, l_2$.

The complex data, z or Z , are stored with real and imaginary parts in separate real arrays, x and y , respectively.

These subprograms require that l_1 and l_2 be products of powers of 2, 3, and 5, i.e., of the form

$$l_k = 2^{p_k} 3^{q_k} 5^{r_k},$$

where $p_k, q_k, r_k \geq 0$, $k=1,2$.

Usage VECLIB:

```

INTEGER*4      l1, l2, ldxy, iopt, ier
REAL*4         x(ldxy, l2), y(ldxy, l2)
CALL S2DFFT(x, y, l1, l2, ldxy, iopt, ier)

INTEGER*4      l1, l2, ldxy, iopt, ier
REAL*8         x(ldxy, l2), y(ldxy, l2)
CALL D2DFFT(x, y, l1, l2, ldxy, iopt, ier)

```

VECLIB8:

```

      INTEGER*8      l1, l2, ldx, iopt, ier
      REAL*8         x(ldx, l2), y(ldx, l2)
      CALL S2DFFT(x, y, l1, l2, ldx, iopt, ier)

```

Input	x	Array of real parts of the data to be transformed.
	y	Array of imaginary parts of the data to be transformed.
	l1	Number of rows of data, of the form $l1 = 2^{p_1} 3^{q_1} 5^{r_1}$, with $p_1, q_1, r_1 \geq 0$.
	l2	Number of columns of data, of the form $l2 = 2^{p_2} 3^{q_2} 5^{r_2}$, with $p_2, q_2, r_2 \geq 0$.
	ldx	The leading dimension of arrays x and y , with $ldx \geq l1$.
Output	iopt	Option flag: iopt ≥ 0 Compute forward transform. iopt < 0 Compute inverse transform.
	x and y	The transformed data replaces the input if ier = 0 is returned.
	ier	Status response: ier = 0 Normal return—transform successful. ier = -1 l1 not of the required form. ier = -2 l2 not of the required form. ier = -3 ldx < l1. ier ≤ -4 Probable error in ldx or dimensions of x and y .

Name C3DFFT/Z3DFFT
Three-Dimensional FFT

Purpose Given an array of complex data, these subprograms compute the three-dimensional forward or inverse DFT using a radix 2-3-5 FFT algorithm. Two companion subprograms, S3DFFT and D3DFFT, perform the same operation but with the complex data presented with real and imaginary parts in separate real arrays.

The three-dimensional forward DFT of $z(n_1, n_2, n_3)$, for $n_1 = 1, 2, \dots, l_1$, $n_2 = 1, 2, \dots, l_2$, and $n_3 = 1, 2, \dots, l_3$, is defined by

$$Z(m_1, m_2, m_3) = \sum_{n_1=1}^{l_1} \sum_{n_2=1}^{l_2} \sum_{n_3=1}^{l_3} z(n_1, n_2, n_3) \times e^{-2\pi i(m_1-1)(n_1-1)/l_1} e^{-2\pi i(m_2-1)(n_2-1)/l_2} e^{-2\pi i(m_3-1)(n_3-1)/l_3}$$

for $m_1 = 1, 2, \dots, l_1$, $m_2 = 1, 2, \dots, l_2$, $m_3 = 1, 2, \dots, l_3$, and $i = \sqrt{-1}$.

Alternatively, the three-dimensional inverse DFT of $Z(m_1, m_2, m_3)$, for $m_1 = 1, 2, \dots, l_1$, $m_2 = 1, 2, \dots, l_2$, and $m_3 = 1, 2, \dots, l_3$, is defined by

$$z(n_1, n_2, n_3) = \frac{1}{l_1 l_2 l_3} \sum_{m_1=1}^{l_1} \sum_{m_2=1}^{l_2} \sum_{m_3=1}^{l_3} Z(m_1, m_2, m_3) \times e^{+2\pi i(m_1-1)(n_1-1)/l_1} e^{+2\pi i(m_2-1)(n_2-1)/l_2} e^{+2\pi i(m_3-1)(n_3-1)/l_3}$$

for $n_1 = 1, 2, \dots, l_1$, $n_2 = 1, 2, \dots, l_2$, and $n_3 = 1, 2, \dots, l_3$.

These subprograms require that l_1 , l_2 , and l_3 be products of powers of 2, 3, and 5, i.e., of the form

$$l_k = 2^{p_k} 3^{q_k} 5^{r_k},$$

where $p_k, q_k, r_k \geq 0$, $k=1,2,3$.

Usage VECLIB:

```

      INTEGER*4      l1, l2, l3, ldz, mdz, iopt, ier
      COMPLEX*8      z(ldz, mdz, l3)
      CALL C3DFFT(z, l1, l2, l3, ldz, mdz, iopt, ier)

      INTEGER*4      l1, l2, l3, ldz, mdz, iopt, ier
      COMPLEX*16     z(ldz, mdz, l3)
      CALL Z3DFFT(z, l1, l2, l3, ldz, mdz, iopt, ier)

```

VECLIB8:

```

      INTEGER*8      l1, l2, l3, ldz, mdz, iopt, ier
      COMPLEX*16     z(ldz, mdz, l3)
      CALL C3DFFT(z, l1, l2, l3, ldz, mdz, iopt, ier)

```

Input	z	Array of data to be transformed.
	l1	Number of rows of data, of the form $l1 = 2^{p_1} 3^{q_1} 5^{r_1}$, with $p_1, q_1, r_1 \geq 0$.
	l2	Number of columns of data, of the form $l2 = 2^{p_2} 3^{q_2} 5^{r_2}$, with $p_2, q_2, r_2 \geq 0$.
	l3	Number of planes of data, of the form $l3 = 2^{p_3} 3^{q_3} 5^{r_3}$, with $p_3, q_3, r_3 \geq 0$.
	ldz	The leading dimension of array z, with $ldz \geq l1$.
	mdz	The middle dimension of array z, with $mdz \geq l2$.
	iopt	Option flag: $iopt \geq 0$ Compute forward transform. $iopt < 0$ Compute inverse transform.
Output	z	The transformed data replaces the input if $ier = 0$ is returned.
	ier	Status response: $ier = 0$ Normal return—transform successful. $ier = -1$ l1 not of the required form. $ier = -2$ l2 not of the required form. $ier = -3$ l3 not of the required form. $ier = -4$ $ldz < l1$. $ier = -5$ $mdz < l2$. $ier = -6$ Probable error in ldz or mdz.

Name S3DFFT/D3DFFT
Three-Dimensional FFT

Purpose Given a set of complex data with real and imaginary parts in separate real arrays, these subprograms compute the three-dimensional forward or inverse DFT using a radix 2-3-5 FFT algorithm. Two companion subprograms, C3DFFT and Z3DFFT, perform the same operation but with the complex data presented in a complex array.

The three-dimensional forward DFT of $z(n_1, n_2, n_3)$, for $n_1 = 1, 2, \dots, l_1$, $n_2 = 1, 2, \dots, l_2$, and $n_3 = 1, 2, \dots, l_3$, is defined by

$$Z(m_1, m_2, m_3) = \sum_{n_1=1}^{l_1} \sum_{n_2=1}^{l_2} \sum_{n_3=1}^{l_3} z(n_1, n_2, n_3) \times e^{-2\pi i(m_1-1)(n_1-1)/l_1} e^{-2\pi i(m_2-1)(n_2-1)/l_2} e^{-2\pi i(m_3-1)(n_3-1)/l_3}$$

for $m_1 = 1, 2, \dots, l_1$, $m_2 = 1, 2, \dots, l_2$, $m_3 = 1, 2, \dots, l_3$, and $i = \sqrt{-1}$.

Alternatively, the three-dimensional inverse DFT of $Z(m_1, m_2, m_3)$, for $m_1 = 1, 2, \dots, l_1$, $m_2 = 1, 2, \dots, l_2$, and $m_3 = 1, 2, \dots, l_3$, is defined by

$$z(n_1, n_2, n_3) = \frac{1}{l_1 l_2 l_3} \sum_{m_1=1}^{l_1} \sum_{m_2=1}^{l_2} \sum_{m_3=1}^{l_3} Z(m_1, m_2, m_3) \times e^{+2\pi i(m_1-1)(n_1-1)/l_1} e^{+2\pi i(m_2-1)(n_2-1)/l_2} e^{+2\pi i(m_3-1)(n_3-1)/l_3}$$

for $n_1 = 1, 2, \dots, l_1$, $n_2 = 1, 2, \dots, l_2$, and $n_3 = 1, 2, \dots, l_3$.

The complex data, z or Z , are stored with real and imaginary parts in separate real arrays, x and y , respectively.

These subprograms require that l_1 , l_2 , and l_3 be products of powers of 2, 3, and 5, i.e., of the form

$$l_k = 2^{p_k} 3^{q_k} 5^{r_k},$$

where $p_k, q_k, r_k \geq 0$, $k=1,2,3$.

Usage VECLIB:

```
INTEGER*4      l1, l2, l3, ldx, mdxy, iopt, ier
REAL*4         x(ldx, mdxy, l3), y(ldx, mdxy, l3)
CALL S3DFFT(x, y, l1, l2, l3, ldx, mdxy, iopt, ier)
```

```

INTEGER*4      11, 12, 13, ldx, mdx, iopt, ier
REAL*8         x(ldx, mdx, 13), y(ldx, mdx, 13)
CALL D3DFFT(x, y, 11, 12, 13, ldx, mdx, iopt, ier)

```

VECLIB8:

```

INTEGER*8      11, 12, 13, ldx, mdx, iopt, ier
REAL*8         x(ldx, mdx, 13), y(ldx, mdx, 13)
CALL S3DFFT(x, y, 11, 12, 13, ldx, mdx, iopt, ier)

```

Input	x	Array of real parts of the data to be transformed.
	y	Array of imaginary parts of the data to be transformed.
	11	Number of rows of data, of the form $11 = 2^{p_1} 3^{q_1} 5^{r_1}$, with $p_1, q_1, r_1 \geq 0$.
	12	Number of columns of data, of the form $12 = 2^{p_2} 3^{q_2} 5^{r_2}$, with $p_2, q_2, r_2 \geq 0$.
	13	Number of planes of data, of the form $13 = 2^{p_3} 3^{q_3} 5^{r_3}$, with $p_3, q_3, r_3 \geq 0$.
	ldx	The leading dimension of arrays x and y , with $ldx \geq 11$.
	mdx	The middle dimension of arrays x and y , with $mdx \geq 12$.
Output	iopt	Option flag: iopt ≥ 0 Compute forward transform. iopt < 0 Compute inverse transform.
	x and y	The transformed data replaces the input if ier = 0 is returned.
	ier	Status response: ier = 0 Normal return—transform successful. ier = -1 11 not of the required form. ier = -2 12 not of the required form. ier = -3 13 not of the required form. ier = -4 $ldx < 11$. ier = -5 $mdx < 12$. ier = -6 Probable error in ldx or mdx .

Name CFFTS/ZFFTS
Simultaneous One-Dimensional FFT

Purpose Given a number of sets of one-dimensional complex data in a complex array, these subprograms simultaneously compute all of their one-dimensional forward or inverse DFT using a radix 2-3-5 FFT algorithm. Two companion subprograms, SFFTS and DFFTS, perform the same operation but with the complex data presented with real and imaginary parts in separate real arrays. Other subprograms, documented elsewhere in this chapter, are more suited for computing just one transform.

The one-dimensional forward DFT of a complex data set $z(n)$, for $n = 1, 2, \dots, l$, is defined by

$$Z(m) = \sum_{n=1}^l z(n) e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, \dots, l$ and $i = \sqrt{-1}$.

Alternatively, the one-dimensional scaled inverse DFT of a data set $Z(m)$, for $m = 1, 2, \dots, l$, is defined by

$$z(n) = \frac{1}{l} \sum_{m=1}^l Z(m) e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, \dots, l$.

These subprograms perform forward or inverse transform operations simultaneously on a number of data sets. They require that the length l of the data sets be a product of powers of 2, 3, and 5, i.e., of the form

$$l = 2^p 3^q 5^r,$$

where $p, q, r \geq 0$.

Usage VECLIB:

```

      INTEGER*4      l, incl, n, incn, iopt, ier
      COMPLEX*8      z(lenz)
      CALL CFFTS(z, l, incl, n, incn, iopt, ier)

      INTEGER*4      l, incl, n, incn, iopt, ier
      COMPLEX*16     z(lenz)
      CALL ZFFTS(z, l, incl, n, incn, iopt, ier)
```

VECLIB8:

```

INTEGER*8      l, incl, n, incn, iopt, ier
COMPLEX*16     z(lenz)
CALL CFFTS(z, l, incl, n, incn, iopt, ier)

```

Input

z Array containing **n** data sets, each consisting of **l** data points, to be transformed. Typically, **z** will be a two- or three-dimensional array with each set of data being a one-dimensional array section. Refer to "Notes" for suggested usages. Treating **z** as a one-dimensional array results in

$$\text{lenz} = (l - 1) \times \text{incl} + (n - 1) \times \text{incn} + 1.$$

The i -th data point of the j -th data set, $1 \leq i \leq l$, $1 \leq j \leq n$, is stored in

$$z((i-1) \times \text{incl} + (j-1) \times \text{incn} + 1).$$

l Number of data points in each data set, of the form $l = 2^p 3^q 5^r$, with $p, q, r \geq 0$.

incl Storage increment between successive elements of the same data set, **incl** > 0. Use **incl** = 1 if each data set is stored contiguously in array **z**.

n The number of data sets, **n** > 0.

incn Storage increment between corresponding data points of successive data sets, **incn** > 0.

iopt Option flag:

iopt ≥ 0 Compute forward transform.

iopt < 0 Compute inverse transform.

Output

z The transformed data replaces the input if **ier** = 0 is returned. Unchanged if **ier** < 0.

ier	Status response:
ier = 0	Normal return—transform successful.
ier = -1	l not of the form $2^p 3^q 5^r$ for $p, q, r \geq 0$.
ier = -2	incl ≤ 0 .
ier = -3	n ≤ 0 .
ier = -4	incn ≤ 0 .
ier = -5	l, incl , n , and incn are incompatible. Refer to "Notes."

Notes

Typically, **z** 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 **z** is a two-dimensional array of dimension **ldz** by **mdz**, and if the data sets are stored in the columns of **z**, $1 \leq \text{ldz}$, $n \leq \text{mdz}$, **incl** = 1, and **incn** = **ldz**. For example,

CALL CFFTS (z, l, 1, n, ldz, iopt, ier)

If **z** is a two-dimensional array as above and data sets are stored in rows of **z**, $1 \leq \text{mdz}$, $n \leq \text{ldz}$, **incl** = **ldz**, and **incn** = 1. For example,

CALL CFFTS (z, l, ldz, n, 1, iopt, ier)

If **z** is a three-dimensional array of dimension **ldz** by **mdz** by **ndz**, then **incl** and **incn** will usually be 1, **ldz**, or **ldz**×**mdz**, depending on which of the subscripts of the three-dimensional array varies within a data set, which subscript varies between data sets, and which remains constant. Specifically, if the subscript that varies within a data set is the

- 1st subscript, use **incl** = 1.
- 2nd subscript, use **incl** = **ldz**.
- 3rd subscript, use **incl** = **ldz**×**mdz**.

Similarly, if the subscript that varies between data sets is the

- 1st subscript, use **incn** = 1.
- 2nd subscript, use **incn** = **ldz**.
- 3rd subscript, use **incn** = **ldz**×**mdz**.

l, **incl**, **n**, and **incn** must be such that no two points of any data sets occupy the same element of **z**. These subprograms detect this situation and return **ier** = -5 if

$$\text{incl} < n \times \text{gcd}(\text{incl}, \text{incn})$$

and

$$\text{incn} < l \times \text{gcd}(\text{incl}, \text{incn})$$

where $\text{gcd}(\dots)$ is the greatest common divisor.

Example 1 Compute the forward DFT of 256 COMPLEX*8 data sets of length 1024. The data sets are stored as the columns of array Z whose dimensions are 1025 by 256.

```

INTEGER*4 L, INCL, N, INCN, IOPT, IER
COMPLEX*8 Z(1025,256)
L = 1024
INCL = 1
N = 256
INCN = 1025
IOPT = 1
CALL CFPTS (Z,L,INCL,N,INCN,IOPT,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
END IF

```

Example 2 Compute the inverse DFT of 1024 COMPLEX*8 data sets of length 256. The data sets are stored as the rows of array Z whose dimensions are 1025 by 256.

```

INTEGER*4 L, INCL, N, INCN, IOPT, IER
COMPLEX*8 Z(1025,256)
L = 256
INCL = 1025
N = 1024
INCN = 1
IOPT = -1
CALL CFPTS (Z,L,INCL,N,INCN,IOPT,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
END IF

```

Name SFFTS/DFFTS
Simultaneous One-Dimensional FFT

Purpose Given a number of sets of one-dimensional complex data with real and imaginary parts in separate real arrays, these subprograms compute all of their one-dimensional forward or inverse DFT using a radix 2-3-5 FFT algorithm. Two companion subprograms, CFFTS and ZFFTS, perform the same operation but with the complex data presented in a complex array. Other subprograms, documented elsewhere in this chapter, are more suited for computing just one transform.

The one-dimensional forward discrete Fourier transform of a complex set of data $z(n)$, for $n = 1, 2, \dots, l$, is defined by

$$Z(m) = \sum_{n=1}^l z(n) e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, \dots, l$ and $i = \sqrt{-1}$.

Alternatively, the one-dimensional scaled inverse DFT of $Z(m)$, for $m = 1, 2, \dots, l$, is defined by

$$z(n) = \frac{1}{l} \sum_{m=1}^l Z(m) e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, \dots, l$.

These subprograms perform forward or inverse transform operations simultaneously on a number of data sets. They require that the length l of the data sets be a product of powers of 2, 3, and 5, i.e., of the form

$$l = 2^p 3^q 5^r,$$

where $p, q, r \geq 0$.

The complex data, z or Z , are stored with real and imaginary parts in separate real arrays, x and y , respectively.

Usage VECLIB:

```

      INTEGER*4      l, incl, n, incn, iopt, ier
      REAL*4         x(lenxy), y(lenxy)
      CALL SFFTS(x, y, l, incl, n, incn, iopt, ier)

      INTEGER*4      l, incl, n, incn, iopt, ier
      REAL*8         x(lenxy), y(lenxy)
      CALL DFFTS(x, y, l, incl, n, incn, iopt, ier)
```

VECLIB8:

```

INTEGER*8      l, incl, n, incn, iopt, ier
REAL*8        x(lenxy), y(lenxy)
CALL SFFTS(x, y, l, incl, n, incn, iopt, ier)

```

Input	x and y	Arrays containing n data sets, each consisting of l data points, to be transformed. Typically, x and y will be two- or three-dimensional arrays with each data set being a one-dimensional array section. Refer to "Notes" for suggested usages. Treating x and y as one-dimensional arrays results in $\text{lenxy} = (l - 1) \times \text{incl} + (n - 1) \times \text{incn} + 1.$ The real and imaginary parts of the <i>i</i>-th data point of the <i>j</i>-th data set, $1 \leq i \leq l$, $1 \leq j \leq n$, are stored in $\mathbf{x}((i - 1) \times \text{incl} + (j - 1) \times \text{incn} + 1)$ and $\mathbf{y}((i - 1) \times \text{incl} + (j - 1) \times \text{incn} + 1),$ respectively.
	l	Number of data points in each data set, of the form $l = 2^p 3^q 5^r$, with $p, q, r \geq 0$.
	incl	Storage increment between successive elements of the same data set, incl > 0. Use incl = 1 if each data set is stored contiguously in arrays x and y .
	n	The number of data sets, n > 0.
	incn	Storage increment between corresponding data points of successive data sets, incn > 0.
	iopt	Option flag: iopt 0 Compute forward transform. iopt < 0 Compute inverse transform.
Output	x and y	The transformed data replaces the input if ier = 0 is returned. Unchanged if ier < 0.

ier	Status response:
ier = 0	Normal return—transform successful.
ier = -1	l not of the form $2^p 3^q 5^r$ for $p, q, r \geq 0$.
ier = -2	incl ≤ 0 .
ier = -3	n ≤ 0 .
ier = -4	incn ≤ 0 .
ier = -5	l , incl , n , and incn are incompatible. Refer to "Notes."

Notes Typically, **x** and **y** will be two- or three-dimensional arrays with each data set being a one-dimensional section of the arrays, 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**, $1 \leq \text{ldxy}$, $n \leq \text{mdxy}$, **incl** = 1, and **incn** = **ldxy**. For example:

CALL SFFTS (x, y, l, 1, n, ldxy, iopt, ier)

If **x** and **y** are two-dimensional arrays as above and data sets are stored in rows of **x** and **y**, $1 \leq \text{mdxy}$, $n \leq \text{ldxy}$, **incl** = **ldxy**, and **incn** = 1. For example:

CALL SFFTS (x, y, l, ldxy, n, 1, iopt, ier)

If **x** and **y** are three-dimensional arrays of dimension **ldxy** by **mdxy** by **ndxy**, then **incl** and **incn** will usually be 1, **ldxy**, or **ldxy**×**mdxy**, depending on which of the subscripts of the three-dimensional array varies within a data set, which subscript varies between data sets, and which remains constant. Specifically, if the subscript that varies within a data set is the

1st	subscript, use	incl = 1.
2nd	subscript, use	incl = ldxy .
3rd	subscript, use	incl = ldxy × mdxy .

Similarly, if the subscript that varies between data sets is the

1st	subscript, use	incn = 1.
2nd	subscript, use	incn = ldxy .
3rd	subscript, use	incn = ldxy × mdxy .

l, **incl**, **n**, and **incn** must be such that no two points of any data sets occupy the same elements of **x** and **y**. These subprograms detect this situation and return **ier** = -5 if

$$\text{incl} < n \times \text{gcd}(\text{incl}, \text{incn})$$

and

$$\text{incn} < l \times \text{gcd}(\text{incl}, \text{incn})$$

where $\text{gcd}(\dots)$ is the greatest common divisor.

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

Example 1 Compute the forward discrete Fourier transform of 256 complex data sets of length 1024. Real and imaginary parts of data sets are stored as columns of arrays X and Y whose dimensions are 1025 by 256.

```

INTEGER*4 L, INCL, N, INCN, IOPT, IER
REAL*4    X(1025,256), Y(1025,256)
L = 1024
INCL = 1
N = 256
INCN = 1025
IOPT = 1
CALL SFFTS (X,Y,L,INCL,N,INCN,IOPT,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
END IF

```

Example 2 Compute the inverse discrete Fourier transform of 1024 complex data sets of length 256. Real and imaginary parts of data sets are stored as rows of arrays X and Y whose dimensions are 1025 by 256.

```

INTEGER*4 L, INCL, N, INCN, IOPT, IER
REAL*4    X(1025,256), Y(1025,256)
L = 256
INCL = 1025
N = 1024
INCN = 1
IOPT = -1
CALL SFFTS (X,Y,L,INCL,N,INCN,IOPT,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
END IF

```

Name CRC1FT/ZRC1FT
Real-to-Complex One-Dimensional FFT

Purpose These subprograms compute either the forward real-to-complex or the inverse complex-to-real, one-dimensional, DFT using a radix 2-3-5 FFT algorithm optimized for real input or output. Two companion subprograms, SRC1FT and DRC1FT, perform a similar operation but in a space-conserving manner.

The one-dimensional forward DFT of a real data set, $z(n)$, for $n = 1, 2, \dots, l$, is defined by

$$Z(m) = \sum_{n=1}^l z(n) e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, \dots, l$ and $i = \sqrt{-1}$.

The sequence $Z(m)$ is conjugate-symmetric about $Z(l/2+1)$, i.e.,

$$\text{Im}(Z(1)) = \text{Im}(Z(l/2 + 1)) = 0$$

and

$$Z(l/2 + 1 + m) = \bar{Z}(l/2 + 1 - m), m = 1, 2, \dots, l/2 - 1,$$

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

Alternatively, if $Z(m)$, for $m = 1, 2, \dots, l$, is a conjugate-symmetric complex data set, the one-dimensional real scaled inverse discrete Fourier transform of $Z(m)$ is defined by

$$z(n) = \frac{1}{l} \sum_{m=1}^l Z(m) e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, \dots, l$.

Finally, the one-dimensional, real, unscaled, inverse DFT of the conjugate-symmetric complex data set $Z(m)$, for $m = 1, 2, \dots, l$, is defined by

$$z(n) = \sum_{m=1}^l Z(m) e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, \dots, l$.

These subprograms require that l 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.

Usage

Because it is common to use one data set length repetitively, these subprograms have a separate initialization call such that the setup can be performed only once for each different transform size. Therefore, you will always have at least two **CALL** statements to the FFT subprogram using the same working storage array.

VECLIB:

```

      INTEGER*4      l, iopt, ier
      COMPLEX*8      z(l)
      REAL*4         work(3*1/2)
      CALL CRC1FT(z, l, work, iopt, ier)

```

```

      INTEGER*4      l, iopt, ier
      COMPLEX*16     z(l)
      REAL*8         work(3*1/2)
      CALL ZRC1FT(z, l, work, iopt, ier)

```

VECLIB8:

```

      INTEGER*8      l, iopt, ier
      COMPLEX*16     z(l)
      REAL*8         work(3*1/2)
      CALL CRC1FT(z, l, work, iopt, ier)

```

Input

z	Array of data to be transformed. For a forward real-to-complex transform, only the real parts of z are used. For an inverse complex-to-real transform, only the first $l/2+1$ elements are used. Not used at all if iopt = -3.								
l	Number of data points, of the form $l = 2^p$, with $p \geq 0$.								
iopt	Option flag: <table border="0" style="margin-left: 20px;"> <tr> <td style="padding-right: 20px;">iopt = +1</td> <td>Compute forward real-to-complex transform.</td> </tr> <tr> <td style="padding-right: 20px;">iopt = -1</td> <td>Compute scaled inverse complex-to-real transform.</td> </tr> <tr> <td style="padding-right: 20px;">iopt = -2</td> <td>Compute unscaled inverse complex-to-real transform.</td> </tr> <tr> <td style="padding-right: 20px;">iopt = -3</td> <td>Initialize work for subsequent transforms of length l.</td> </tr> </table>	iopt = +1	Compute forward real-to-complex transform.	iopt = -1	Compute scaled inverse complex-to-real transform.	iopt = -2	Compute unscaled inverse complex-to-real transform.	iopt = -3	Initialize work for subsequent transforms of length l .
iopt = +1	Compute forward real-to-complex transform.								
iopt = -1	Compute scaled inverse complex-to-real transform.								
iopt = -2	Compute unscaled inverse complex-to-real transform.								
iopt = -3	Initialize work for subsequent transforms of length l .								

**Working
Storage****work**

If **iopt** = -3, **work** is initialized for computing transforms of length **l**.

If **iopt** ≠ -3, **work** must have been initialized by a previous call with this value of **l** in which **iopt** was -3.

Output**z**

If **iopt** ≠ -3, the transformed data replaces the input if **ier** = 0 is returned.

For a forward real-to-complex transform, all **l** elements of the transform are returned.

For an inverse complex-to-real transform, the real result is stored in the real parts of **z** and the imaginary parts of **z** are set to zero.

Not used as output if **iopt** = -3.

ier

Status response:

ier = 0 Normal return—transform or initialization successful.

ier = -1 **l** < 0.

ier = -2 **l** not a power of 2.

ier = -3 Invalid value of **iopt**.

ier = -4 Insufficient dynamic memory available for workspace.

Name SRC1FT/DRC1FT
Real-to-Complex One-Dimensional FFT

Purpose Given a set of real data in a real array, these subprograms compute the nonredundant part of the complex one-dimensional forward DFT using a radix 2 FFT algorithm optimized for real input. Alternatively, given the nonredundant part of a conjugate-symmetric complex one-dimensional data set, these subprograms compute the real inverse discrete Fourier transform. A pair of companion subprograms, CRC1FT and ZRC1FT, performs a similar operation, but with the real and complex data presented in a complex array.

The one-dimensional forward DFT of a real data set, $z(n)$, for $n = 1, 2, \dots, l$, is defined by

$$Z(m) = \sum_{n=1}^l z(n) e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, \dots, l$ and $i = \sqrt{-1}$.

The sequence $Z(m)$ is conjugate-symmetric about $Z(l/2+1)$, i.e.,

$$\text{Im}(Z(1)) = \text{Im}(Z(l/2 + 1)) = 0$$

and

$$Z(l/2 + 1 + m) = \bar{Z}(l/2 + 1 - m), m = 1, 2, \dots, l/2 - 1,$$

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

Alternatively, if $Z(m)$, for $m = 1, 2, \dots, l$, is a conjugate-symmetric complex data set, the one-dimensional real scaled inverse discrete Fourier transform of $Z(m)$ is defined by

$$z(n) = \frac{1}{l} \sum_{m=1}^l Z(m) e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, \dots, l$.

Finally, the one-dimensional real unscaled inverse DFT of the conjugate-symmetric complex data set $Z(m)$, for $m = 1, 2, \dots, l$, is defined by

$$z(n) = \sum_{m=1}^l Z(m) e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, \dots, l$.

These subprograms require that l 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.

Usage

Since it is common to use one data set length repetitively, these subprograms have a separate initialization call such that the setup can be performed only once for each different transform size. You will, therefore, always have at least two call statements to the FFT subprogram, using the same working storage array.

VECLIB:

```
INTEGER*4      l, iopt, ier
REAL*4         x(l+2), work(3*l/2)
CALL SRC1FT(x, l, work, iopt, ier)
```

```
INTEGER*4      l, iopt, ier
REAL*8         x(l+2), work(3*l/2)
CALL DRC1FT(x, l, work, iopt, ier)
```

VECLIB8:

```
INTEGER*8      l, iopt, ier
REAL*8         x(l+2), work(3*l/2)
CALL SRC1FT(x, l, work, iopt, ier)
```

Input

x Array containing l real data points or the first $l/2+1$ complex data points of a conjugate-symmetric complex data set of length l , to be transformed.

For a forward real-to-complex transform, the i -th real data point is stored in $x(i)$, $i = 1, 2, \dots, l$.

For an inverse complex-to-real transform, the real part of the i -th complex data point is stored in $x(2i-1)$ and the imaginary part of the i -th data point is stored in $x(2i)$, $i = 1, 2, \dots, l/2+1$.

Not used if $iopt = -3$.

l Number of data points, of the form $l = 2^p$, with $p \geq 0$.

	iopt	Option flag: iopt = +1 Compute forward transform. iopt = -1 Compute scaled inverse transform. iopt = -2 Compute unscaled inverse transform. iopt = -3 Initialize work for subsequent transforms of length l.
Working Storage	work	If iopt = -3, work is initialized for computing transforms of length l. If iopt $\neq$ -3, work must have been initialized by a previous call with this value of l in which iopt was -3.
Output	x	If iopt $\neq$ -3, the transformed data replaces the input if ier = 0 is returned. For a forward real-to-complex transform, the real part of the i-th complex output point is stored in $x(2 \times i - 1)$ and the imaginary part of the i-th output point is stored in $x(2 \times i)$, $i = 1, 2, \dots, l/2 + 1$. If needed, the remaining $l/2 - 1$ complex output values may be formed by using the conjugate symmetry condition. For an inverse complex-to-real transform, the i-th real output point is stored in $x(i)$, $i = 1, 2, \dots, l$. Not used as output if iopt = -3.
	ier	Status response: ier = 0 Normal return—transform or initialization successful. ier = -1 $l < 0$. ier = -2 l not a power of 2. ier = -3 Invalid value of iopt. ier = -4 Insufficient dynamic memory available for workspace.

Name CRC2FT/ZRC2FT
Real-to-Complex Two-Dimensional FFT

Purpose These subprograms compute either the forward real-to-complex or the inverse complex-to-real, two-dimensional, discrete Fourier transform using a radix 2-3-5 fast Fourier transform (FFT) algorithm optimized for real input or output. A pair of companion subprograms, SRC2FT and DRC2FT, performs a similar operation, but in a space-conserving manner.

The two-dimensional, complex, forward discrete Fourier transform of a real data set, $z(n_1, n_2)$, for $n_1 = 1, 2, \dots, l_1$ and $n_2 = 1, 2, \dots, l_2$, is defined by

$$Z(m_1, m_2) = \sum_{n_1=1}^{l_1} \sum_{n_2=1}^{l_2} z(n_1, n_2) e^{-2\pi i(m_1-1)(n_1-1)/l_1} e^{-2\pi i(m_2-1)(n_2-1)/l_2}$$

for $m_1 = 1, 2, \dots, l_1$, $m_2 = 1, 2, \dots, l_2$, and $i = \sqrt{-1}$.

The $Z(m_1, m_2)$ satisfy the two-dimensional, conjugate-symmetry conditions:

$$\text{Im}(Z(1, 1)) = 0,$$

$$Z(m_1, 1) = \bar{Z}(l_1 + 2 - m_1, 1), m_1 = 2, 3, \dots, l_1,$$

$$Z(1, m_2) = \bar{Z}(1, l_2 + 2 - m_2), m_2 = 2, 3, \dots, l_2,$$

and

$$Z(m_1, m_2) = \bar{Z}(l_1 + 2 - m_1, l_2 + 2 - m_2), m_k = 2, 3, \dots, l_k, k = 1, 2,$$

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

Alternatively, if $Z(m_1, m_2)$, for $m_1 = 1, 2, \dots, l_1$ and $m_2 = 1, 2, \dots, l_2$, is a conjugate-symmetric complex data set, the two-dimensional, real, scaled, inverse discrete Fourier transform of $Z(m_1, m_2)$ is defined by

$$z(n_1, n_2) = \frac{l}{l_1 l_2} \sum_{m_1=1}^{l_1} \sum_{m_2=1}^{l_2} Z(m_1, m_2) e^{+2\pi i(m_1-1)(n_1-1)/l_1} e^{+2\pi i(m_2-1)(n_2-1)/l_2}$$

for $n_1 = 1, 2, \dots, l_1$ and $n_2 = 1, 2, \dots, l_2$.

These subprograms require that l_1 and l_2 be products of powers of 2, 3, and 5, i.e., of the form

$$l_k = 2^{p_k} 3^{q_k} 5^{r_k},$$

where $p_k, q_k, r_k \geq 0$, $k=1,2$, and where either $l_1 = 1$ or l_1 is even.

Usage

VECLIB:

```

      INTEGER*4      l1, l2, ldz, iopt, ier
      COMPLEX*8      z(ldz, l2)
      CALL CRC2FT(z, l1, l2, ldz, iopt, ier)

      INTEGER*4      l1, l2, ldz, iopt, ier
      COMPLEX*16     z(ldz, l2)
      CALL ZRC2FT(z, l1, l2, ldz, iopt, ier)

```

VECLIB8:

```

      INTEGER*8      l1, l2, ldz, iopt, ier
      COMPLEX*16     z(ldz, l2)
      CALL CRC2FT(z, l1, l2, ldz, iopt, ier)

```

Input

z	Array of data to be transformed. For a forward real-to-complex transform, only the real parts of z are used as input. For an inverse complex-to-real transform, only the first $l1/2+1$ rows of z are used as input.
l1	Number of rows of data, of the form $l1 = 2^{p_1} 3^{q_1} 5^{r_1}$, with $q_1, r_1 \geq 0$ and either $l1 = 1$ or $p_1 \geq 1$.
l2	Number of columns of data, of the form $l2 = 2^{p_2} 3^{q_2} 5^{r_2}$, with $p_2, q_2, r_2 \geq 0$.
ldz	The leading dimension of array z , with $ldz \geq l1$.
iopt	Option flag: iopt ≥ 0 Compute forward transform. iopt < 0 Compute inverse transform.

Output

z	The $l1$ -by- $l2$ array of transformed data replaces the input if ier = 0 is returned. For an inverse complex-to-real transform, the real result is stored in the real parts of z and the imaginary parts of z are set to zero.

ier

Status response:

ier = 0	Normal return—transform successful.
ier = -1	l1 not of the required form.
ier = -2	l2 not of the required form.
ier = -3	ldz < l1 .
ier = -4	Probable error in ldz or dimensions of z .

Name SRC2FT/DRC2FT
Real-to-Complex Two-Dimensional FFT

Purpose Given a set of two-dimensional real data, these subprograms compute the nonredundant portion of the complex, two-dimensional, forward DFT using a radix 2-3-5 FFT algorithm optimized for real input. Alternatively, given the nonredundant part of a conjugate-symmetric two-dimensional complex data set, these subprograms compute the real inverse discrete Fourier transform using a radix 2-3-5 FFT algorithm optimized for real output. A pair of companion subprograms, CRC2FT and ZRC2FT, performs similar operations, but with the real or complex data presented in a complex array. The companion subprograms require more storage than the ones described here.

The two-dimensional complex forward DFT of a real data set, $z(n_1, n_2)$, for $n_1 = 1, 2, \dots, l_1$ and $n_2 = 1, 2, \dots, l_2$, is defined by

$$Z(m_1, m_2) = \sum_{n_1=1}^{l_1} \sum_{n_2=1}^{l_2} z(n_1, n_2) e^{-2\pi i(m_1-1)(n_1-1)/l_1} e^{-2\pi i(m_2-1)(n_2-1)/l_2}$$

for $m_1 = 1, 2, \dots, l_1$, $m_2 = 1, 2, \dots, l_2$, and $i = \sqrt{-1}$.

The $Z(m_1, m_2)$ satisfy the two-dimensional conjugate-symmetry conditions:

$$\text{Im}(Z(1, 1)) = 0,$$

$$Z(m_1, 1) = \bar{Z}(l_1 + 2 - m_1, 1), m_1 = 2, 3, \dots, l_1,$$

$$Z(1, m_2) = \bar{Z}(1, l_2 + 2 - m_2), m_2 = 2, 3, \dots, l_2,$$

and

$$Z(m_1, m_2) = \bar{Z}(l_1 + 2 - m_1, l_2 + 2 - m_2), m_k = 2, 3, \dots, l_k, k = 1, 2,$$

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

Alternatively, the two-dimensional, real, inverse DFT of $Z(m_1, m_2)$, for $m_1 = 1, 2, \dots, l_1$ and $m_2 = 1, 2, \dots, l_2$, is defined by

$$z(n_1, n_2) = \frac{1}{l_1 l_2} \sum_{m_1=1}^{l_1} \sum_{m_2=1}^{l_2} Z(m_1, m_2) e^{+2\pi i(m_1-1)(n_1-1)/l_1} e^{+2\pi i(m_2-1)(n_2-1)/l_2}$$

for $n_1 = 1, 2, \dots, l_1$ and $n_2 = 1, 2, \dots, l_2$.

These subprograms require that l_1 and l_2 be products of powers of 2, 3, and 5, i.e., of the form

$$l_k = 2^{p_k} 3^{q_k} 5^{r_k},$$

where $p_k, q_k, r_k \geq 0$, $k=1,2$, and where either $l_1=1$ or l_1 is even. Refer to "Notes" for a partial list of permissible values of l_1 and l_2 .

Usage

VECLIB:

```

      INTEGER*4      l1, l2, ldx, iopt, ier
      REAL*4         x(ldx, l2)
      CALL SRC2FT(x, l1, l2, ldx, iopt, ier)

      INTEGER*4      l1, l2, ldx, iopt, ier
      REAL*8         x(ldx, l2)
      CALL DRC2FT(x, l1, l2, ldx, iopt, ier)

```

VECLIB8:

```

      INTEGER*8      l1, l2, ldx, iopt, ier
      REAL*8         x(ldx, l2)
      CALL SRC2FT(x, l1, l2, ldx, iopt, ier)

```

Input

x	Array of data to be transformed. For a forward real-to-complex transform, the real data point $z(n_1, n_2)$ is stored in $x(n_1, n_2)$, $n_1 = 1, 2, \dots, l_1$, $n_2 = 1, 2, \dots, l_2$. For an inverse complex-to-real transform, the real part of $Z(m_1, m_2)$ is stored in $x(2 \times m_1 - 1, m_2)$ and the imaginary part is stored in $x(2 \times m_1, m_2)$, $m_1 = 1, 2, \dots, l_1/2 + 1$, $m_2 = 1, 2, \dots, l_2$.
l1	Number of rows of data, of the form $l_1 = 2^{p_1} 3^{q_1} 5^{r_1}$, with $q_1, r_1 \geq 0$ and either $l_1 = 1$ or $p_1 \geq 1$.
l2	Number of columns of data, of the form $l_2 = 2^{p_2} 3^{q_2} 5^{r_2}$, with $p_2, q_2, r_2 \geq 0$.
ldx	The leading dimension of array x , with $ldx \geq l_1 + 2$.

	iopt	Option flag: iopt ≥ 0 Compute forward transform. iopt < 0 Compute inverse transform.
Output	x	The transformed data replaces the input if ier = 0 is returned. For a forward real-to-complex transform, the real part of $Z(m_1, m_2)$ is stored in $\mathbf{x}(2 \times m_1 - 1, m_2)$ and the imaginary part is stored in $\mathbf{x}(2 \times m_1, m_2)$, $m_1 = 1, 2, \dots, 11/2 + 1$, $m_2 = 1, 2, \dots, 12$. If needed, the remaining $(11/2 - 1) \times 12$ complex output values may be formed by using the conjugate-symmetry condition. For an inverse complex-to-real transform, the real output point $z(n_1, n_2)$ is stored in $\mathbf{x}(n_1, n_2)$, $n_1 = 1, 2, \dots, 11$, $n_2 = 1, 2, \dots, 12$.
	ier	Status response: ier = 0 Normal return—transform successful. ier = -1 11 not of the required form. ier = -2 12 not of the required form. ier = -3 $ldx < 11 + 2$. ier = -4 Probable error in ldx or dimensions of x .

Name CRC3FT/ZRC3FT
Real-to-Complex Three-Dimensional FFT

Purpose These subprograms compute either the forward real-to-complex or the inverse complex-to-real, three-dimensional DFT using a radix 2-3-5 FFT algorithm optimized for real input or output. A pair of companion subprograms, SRC3FT and DRC3FT, performs a similar operation but in a space-conserving manner.

The three-dimensional complex forward DFT of a real data set $z(n_1, n_2, n_3)$, for $n_1 = 1, 2, \dots, l_1$, $n_2 = 1, 2, \dots, l_2$, and $n_3 = 1, 2, \dots, l_3$, is defined by

$$Z(m_1, m_2, m_3) = \sum_{n_1=1}^{l_1} \sum_{n_2=1}^{l_2} \sum_{n_3=1}^{l_3} z(n_1, n_2, n_3) \times e^{-2\pi i(m_1-1)(n_1-1)/l_1} e^{-2\pi i(m_2-1)(n_2-1)/l_2} e^{-2\pi i(m_3-1)(n_3-1)/l_3}$$

for $m_1 = 1, 2, \dots, l_1$, $m_2 = 1, 2, \dots, l_2$, $m_3 = 1, 2, \dots, l_3$, and $i = \sqrt{-1}$.

The $Z(m_1, m_2, m_3)$ satisfy the three-dimensional conjugate-symmetry conditions:

$$\text{Im}(Z(1, 1, 1)) = 0,$$

$$Z(m_1, 1, 1) = \bar{Z}(l_1 + 2 - m_1, 1, 1), m_1 = 2, 3, \dots, l_1,$$

$$Z(1, m_2, 1) = \bar{Z}(1, l_2 + 2 - m_2, 1), m_2 = 2, 3, \dots, l_2,$$

$$Z(1, 1, m_3) = \bar{Z}(1, 1, l_3 + 2 - m_3), m_3 = 2, 3, \dots, l_3,$$

$$Z(m_1, m_2, 1) = \bar{Z}(l_1 + 2 - m_1, l_2 + 2 - m_2, 1), m_k = 2, 3, \dots, l_k, k = 1, 2,$$

$$Z(m_1, 1, m_3) = \bar{Z}(l_1 + 2 - m_1, l_3 + 2 - m_3), m_k = 2, 3, \dots, l_k, k = 1, 3,$$

$$Z(1, m_2, m_3) = \bar{Z}(1, l_2 + 2 - m_2, l_3 + 2 - m_3), m_k = 2, 3, \dots, l_k, k = 2, 3,$$

and

$$Z(m_1, m_2, m_3) = \bar{Z}(l_1 + 2 - m_1, l_2 + 2 - m_2, l_3 + 2 - m_3), m_k = 2, 3, \dots, l_k, k = 1, 2, 3,$$

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

Alternatively, if $Z(m_1, m_2, m_3)$, for $m_1 = 1, 2, \dots, l_1$, $m_2 = 1, 2, \dots, l_2$, and $m_3 = 1, 2, \dots, l_3$, is a conjugate-symmetric complex data set, the

three-dimensional, real, scaled, inverse discrete Fourier transform of $Z(m_1, m_2, m_3)$ is defined by

$$z(n_1, n_2, n_3) = \frac{1}{l_1 l_2 l_3} \sum_{m_1=1}^{l_1} \sum_{m_2=1}^{l_2} \sum_{m_3=1}^{l_3} Z(m_1, m_2, m_3) \times e^{+2\pi i(m_1-1)(n_1-1)/l_1} e^{+2\pi i(m_2-1)(n_2-1)/l_2} e^{+2\pi i(m_3-1)(n_3-1)/l_3}$$

for $n_1 = 1, 2, \dots, l_1$, $n_2 = 1, 2, \dots, l_2$, and $n_3 = 1, 2, \dots, l_3$.

These subprograms require that l_1, l_2 , and l_3 be products of powers of 2, 3, and 5, i.e., of the form

$$l_k = 2^{p_k} 3^{q_k} 5^{r_k},$$

where $p_k, q_k, r_k \geq 0$, $k=1,2,3$, and where either $l_1 = 1$ or l_1 is even.

Usage

VECLIB:

```

INTEGER*4      l1, l2, l3, ldz, mdz, iopt, ier
COMPLEX*8      z(ldz, mdz, l3)
CALL CRC3FT(z, l1, l2, l3, ldz, mdz, iopt, ier)

INTEGER*4      l1, l2, l3, ldz, mdz, iopt, ier
COMPLEX*16     z(ldz, mdz, l3)
CALL ZRC3FT(z, l1, l2, l3, ldz, mdz, iopt, ier)

```

VECLIB8:

```

INTEGER*8      l1, l2, l3, ldz, mdz, iopt, ier
COMPLEX*16     z(ldz, mdz, l3)
CALL CRC3FT(z, l1, l2, l3, ldz, mdz, iopt, ier)

```

Input

z	Array of data to be transformed. For a forward real-to-complex transform, only the real parts of z are used as input. For an inverse complex-to-real transform, only the first $l_1/2+1$ rows of z are used as input.
l1	Number of rows of data, of the form $l_1 = 2^{p_1} 3^{q_1} 5^{r_1}$, with $q_1, r_1 \geq 0$ and either $l_1 = 1$ or $p_1 \geq 1$.
l2	Number of columns of data, of the form $l_2 = 2^{p_2} 3^{q_2} 5^{r_2}$, with $p_2, q_2, r_2 \geq 0$.

	l3	Number of planes of data, of the form $l3 = 2^{p_3} 3^{q_3} 5^{r_3}$, with $p_3, q_3, r_3 \geq 0$.
	ldz	The leading dimension of array z , with $ldz \geq 11$.
	mdz	The middle dimension of array z , with $mdz \geq 12$.
	iopt	Option flag: iopt ≥ 0 Compute forward transform. iopt < 0 Compute inverse transform.
Output	z	The 11-by-12-by-13 array of transformed data replaces the input if ier = 0 is returned. For an inverse complex-to-real transform, the real result is stored in the real parts of z and the imaginary parts of z are set to zero.
	ier	Status response: ier = 0 Normal return—transform successful. ier = -1 11 not of the required form. ier = -2 12 not of the required form. ier = -3 13 not of the required form. ier = -4 $ldz < 11$. ier = -5 $mdz < 12$. ier ≤ -6 Probable error in ldx or mdx .

Name SRC3FT/DRC3FT
Real-to-Complex Three-Dimensional FFT

Purpose Given a set of three-dimensional real data, these subprograms compute the nonredundant portion of the complex three-dimensional forward DFT using a radix 2-3-5 FFT algorithm optimized for real input. Alternatively, given the nonredundant part of a conjugate-symmetric, three-dimensional, complex data set, these subprograms compute the real inverse DFT using a radix 2-3-5 FFT algorithm optimized for real output. A pair of companion subprograms, CRC3FT and ZRC3FT, performs similar operations, but with the real or complex data presented in a complex array. These companion subprograms require more storage than the ones described here.

The three-dimensional, complex, forward a DFT of a real data set $z(n_1, n_2, n_3)$, for $n_1 = 1, 2, \dots, l_1$, $n_2 = 1, 2, \dots, l_2$, and $n_3 = 1, 2, \dots, l_3$, is defined by

$$Z(m_1, m_2, m_3) = \sum_{n_1=1}^{l_1} \sum_{n_2=1}^{l_2} \sum_{n_3=1}^{l_3} z(n_1, n_2, n_3) \times e^{-2\pi i(m_1-1)(n_1-1)/l_1} e^{-2\pi i(m_2-1)(n_2-1)/l_2} e^{-2\pi i(m_3-1)(n_3-1)/l_3}$$

for $m_1 = 1, 2, \dots, l_1$, $m_2 = 1, 2, \dots, l_2$, $m_3 = 1, 2, \dots, l_3$, and $i = \sqrt{-1}$.

The $Z(m_1, m_2, m_3)$ satisfy the three-dimensional conjugate-symmetry conditions:

$$\text{Im}(Z(1, 1, 1)) = 0,$$

$$Z(m_1, 1, 1) = \bar{Z}(l_1 + 2 - m_1, 1, 1), m_1 = 2, 3, \dots, l_1,$$

$$Z(1, m_2, 1) = \bar{Z}(1, l_2 + 2 - m_2, 1), m_2 = 2, 3, \dots, l_2,$$

$$Z(1, 1, m_3) = \bar{Z}(1, 1, l_3 + 2 - m_3), m_3 = 2, 3, \dots, l_3,$$

$$Z(m_1, m_2, 1) = \bar{Z}(l_1 + 2 - m_1, l_2 + 2 - m_2, 1), m_k = 2, 3, \dots, l_k, k = 1, 2,$$

$$Z(m_1, 1, m_3) = \bar{Z}(l_1 + 2 - m_1, 1, l_3 + 2 - m_3), m_k = 2, 3, \dots, l_k, k = 1, 3,$$

$$Z(1, m_2, m_3) = \bar{Z}(1, l_2 + 2 - m_2, l_3 + 2 - m_3), m_k = 2, 3, \dots, l_k, k = 2, 3,$$

and

$$Z(m_1, m_2, m_3) = \bar{Z}(l_1 + 2 - m_1, l_2 + 2 - m_2, l_3 + 2 - m_3), m_k = 2, 3, \dots, l_k, k = 1, 2, 3,$$

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

Alternatively, the three-dimensional, real, inverse discrete Fourier transform of $Z(m_1, m_2, m_3)$, for $m_1 = 1, 2, \dots, l_1$, $m_2 = 1, 2, \dots, l_2$, and $m_3 = 1, 2, \dots, l_3$, is defined by

$$z(n_1, n_2, n_3) = \frac{1}{l_1 l_2 l_3} \sum_{m_1=1}^{l_1} \sum_{m_2=1}^{l_2} \sum_{m_3=1}^{l_3} Z(m_1, m_2, m_3) \times e^{+2\pi i(m_1-1)(n_1-1)/l_1} e^{+2\pi i(m_2-1)(n_2-1)/l_2} e^{+2\pi i(m_3-1)(n_3-1)/l_3}$$

for $n_1 = 1, 2, \dots, l_1$, $n_2 = 1, 2, \dots, l_2$, and $n_3 = 1, 2, \dots, l_3$.

These subprograms require that l_1 , l_2 , and l_3 be products of powers of 2, 3, and 5, i.e., of the form

$$l_k = 2^{p_k} 3^{q_k} 5^{r_k},$$

where $p_k, q_k, r_k \geq 0$, $k=1,2,3$, and where either $l_1 = 1$ or l_1 is even.

Usage

VECLIB:

```

INTEGER*4      l1, l2, l3, ldx, mdx, iopt, ier
REAL*4         x(ldx, mdx, l3)
CALL SRC3FT(x, l1, l2, l3, ldx, mdx, iopt, ier)

INTEGER*4      l1, l2, l3, ldx, mdx, iopt, ier
REAL*8         x(ldx, mdx, l3)
CALL DRC3FT(x, l1, l2, l3, ldx, mdx, iopt, ier)

```

VECLIB8:

```

INTEGER*8      l1, l2, l3, ldx, mdx, iopt, ier
REAL*8         x(ldx, mdx, l3)
CALL SRC3FT(x, l1, l2, l3, ldx, mdx, iopt, ier)

```

Input

x Array of data to be transformed.

For a forward real-to-complex transform, the real data point $z(n_1, n_2, n_3)$ is stored in $x(n_1, n_2, n_3)$,
 $n_1 = 1, 2, \dots, l_1$, $n_2 = 1, 2, \dots, l_2$,
 $n_3 = 1, 2, \dots, l_3$.

If needed, the remaining $(l_2 - 1) \times l_2 \times l_3$ complex output values may be formed by using the conjugate-symmetry condition.

For an inverse complex-to-real transform, the real part of $Z(m_1, m_2, m_3)$ is stored in $\mathbf{x}(2 \times m_1 - 1, m_2, m_3)$ and the imaginary part is stored in $\mathbf{x}(2 \times m_1, m_2, m_3)$,

$m_1 = 1, 2, \dots, 11/2+1, m_2 = 1, 2, \dots, 12,$

$m_3 = 1, 2, \dots, 13.$

- l1** Number of rows of data, of the form $l1 = 2^{p_1} 3^{q_1} 5^{r_1}$, with $q_1, r_1 \geq 0$ and either $l1 = 1$ or $p_1 \geq 1$.
- l2** Number of columns of data, of the form $l2 = 2^{p_2} 3^{q_2} 5^{r_2}$, with $p_2, q_2, r_2 \geq 0$.
- l3** Number of planes of data, of the form $l3 = 2^{p_3} 3^{q_3} 5^{r_3}$, with $p_3, q_3, r_3 \geq 0$.
- ldx** The leading dimension of array $\mathbf{x}$, with $ldx \geq l1+2$.
- mdx** The middle dimension of array $\mathbf{x}$, with $mdx \geq l2$.
- iopt** Option flag:
 $iopt \geq 0$ Compute forward transform.
 $iopt < 0$ Compute inverse transform.

Output

- x** The transformed data replaces the input if $ier = 0$ is returned.
- For a forward real-to-complex transform, the real part of $Z(m_1, m_2, m_3)$ is stored in $\mathbf{x}(2 \times m_1 - 1, m_2, m_3)$ and the imaginary part is stored in $\mathbf{x}(2 \times m_1, m_2, m_3)$,
 $m_1 = 1, 2, \dots, 11/2+1, m_2 = 1, 2, \dots, 12,$
 $m_3 = 1, 2, \dots, 13.$
- For an inverse complex-to-real transform, the real output point $z(n_1, n_2, n_3)$ is stored in $\mathbf{x}(n_1, n_2, n_3)$,
 $n_1 = 1, 2, \dots, 11, n_2 = 1, 2, \dots, 12, n_3 = 1, 2, \dots, 13.$

ier

Status response:

ier = 0	Normal return—transform successful.
ier = -1	l1 not of the required form.
ier = -2	l2 not of the required form.
ier = -3	l3 not of the required form.
ier = -4	ldx < l1 +2.
ier = -5	mdx < l2 .
ier ≤ -6	Probable error in ldx or mdx .

Name CRCFTS/ZRCFTS
Simultaneous Real-to-Complex 1-D FFT

Purpose Given a number of one-dimensional data sets, these subprograms simultaneously compute all of their one-dimensional forward real-to-complex or inverse complex-to-real discrete Fourier transforms using a radix 2-3-5 FFT algorithm optimized for real input or output. A pair of companion subprograms, SRCFTS and DRCFTS, performs the same operation, but in a space-conserving manner. Other subprograms, documented elsewhere in this chapter, are more suited for computing just one real-to-complex or complex-to-real transform.

The one-dimensional complex forward DFT of a real data set $z(n)$, for $n = 1, 2, \dots, l$, is defined by

$$Z(m) = \sum_{n=1}^l z(n) e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, \dots, l$ and $i = \sqrt{-1}$.

The sequence $Z(m)$ is conjugate-symmetric about $Z(l/2+1)$, i.e.,

$$\text{Im}(Z(1)) = \text{Im}(Z(l/2+1)) = 0$$

and

$$Z(l/2+1+m) = \bar{Z}(l/2+1-m), m = 1, 2, \dots, l/2-1.$$

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

Alternatively, if $Z(m)$, for $m = 1, 2, \dots, l$, is a conjugate-symmetric complex data set, the one-dimensional, real, scaled, inverse DFT of $Z(m)$ is defined by

$$z(n) = \frac{1}{l} \sum_{m=1}^l Z(m) e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, \dots, l$.

These subprograms perform forward real-to-complex or inverse complex-to-real transform operations simultaneously on a number of data sets. They require that the length l of the data sets be a product of powers of 2, 3, and 5, 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.

Usage

VECLIB:

```

    INTEGER*4      l, incl, n, incn, iopt, ier
    COMPLEX*8      z(lenz)
    CALL CRCFTS(z, l, incl, n, incn, iopt, ier)

    INTEGER*4      l, incl, n, incn, iopt, ier
    COMPLEX*16     z(lenz)
    CALL ZRCFTS(z, l, incl, n, incn, iopt, ier)

```

VECLIB8:

```

    INTEGER*8      l, incl, n, incn, iopt, ier
    COMPLEX*16     z(lenz)
    CALL CRCFTS(z, l, incl, n, incn, iopt, ier)

```

Input

z Array containing **n** data sets, each consisting of **l** data points, to be transformed. Typically, **z** is a two- or three-dimensional array with each set of data being a one-dimensional array section. Refer to "Notes" for suggested usages.

Treating **z** as a one-dimensional array, results in

$$\text{lenz} = (l - 1) \times \text{incl} + (n - 1) \times \text{incn} + 1.$$

The i -th data point of the j -th data set, $1 \leq i \leq l$, $1 \leq j \leq n$, is stored in

$$z((i - 1) \times \text{incl} + (j - 1) \times \text{incn} + 1).$$

For a forward real-to-complex transform, only real parts of **z** are used as input.

For an inverse complex-to-real transform, only the first $l/2+1$ complex data points of each data set are used as input.

l Number of data points in each data set, of the form $l = 2^p 3^q 5^r$, with $q, r \geq 0$ and either $l = 1$ or $p \geq 1$.

incl Storage increment between successive elements of the same data set, **incl** > 0. Use **incl** = 1 if each data set is stored contiguously in array **z**.

n The number of data sets, **n** > 0.

incn Storage increment between corresponding data points of successive data sets, **incn** > 0.

	iopt	Option flag: iopt ≥ 0 Compute forward transform. iopt < 0 Compute inverse transform.
Output	z	The transformed data replaces the input if ier = 0 is returned. Unchanged if ier < 0 . For a forward real-to-complex transform, each transformed data set satisfies the conjugate-symmetry condition. For an inverse complex-to-real transform, the real result is stored in the real parts of z ; the imaginary parts of z are set to zero.
	ier	Status response: ier = 0 Normal return—transform successful. ier = -1 1 not of the required form. ier = -2 incl ≤ 0 . ier = -3 n ≤ 0 . ier = -4 incn ≤ 0 . ier = -5 l , incl , n , and incn are incompatible. Refer to "Notes."

Notes Typically, **z** is a two- or three-dimensional array with each set of data being a one-dimensional section of the array, that is, all but one subscript is constant within a data set.

If **z** is a two-dimensional array of dimension **ldz** by **mdz**, and if the data sets are stored in the columns of **z**, then $1 \leq \text{ldz}$, $n \leq \text{mdz}$, **incl** = 1, and **incn** = **ldz**. For example, use

CALL CRCFTS (z, 1, 1, n, ldz, iopt, ier)

If **z** is a two-dimensional array as above and the data sets are stored in the rows of **z**, then $1 \leq \text{mdz}$, $n \leq \text{ldz}$, **incl** = **ldz**, and **incn** = 1. For example, use

CALL CRCFTS (z, 1, ldz, n, 1, iopt, ier)

If **z** is a three-dimensional array of dimension **ldz** by **mdz** by **ndz**, then **incl** and **incn** will usually be 1, **ldz**, or **ldz** × **mdz**, depending on which of the subscripts of the three-dimensional array varies within a data set, which subscript varies between data sets, and which remains constant. Specifically, if the subscript that varies within a data set is the


```

1st    subscript, use  incl = 1.
2nd    subscript, use  incl = ldz.
3rd    subscript, use  incl = ldz×mdz.

```

Similarly, if the subscript that varies between data sets is the

```

1st    subscript, use  incn = 1.
2nd    subscript, use  incn = ldz.
3rd    subscript, use  incn = ldz×mdz.

```

l, **incl**, **n**, and **incn** must be such that no two points of any data sets occupy the same element of **z**. These subprograms detect this situation and return **ier** = -5 if

$$\mathbf{incl} < \mathbf{n} \times \mathbf{gcd}(\mathbf{incl}, \mathbf{incn})$$

and

$$\mathbf{incn} < \mathbf{l} \times \mathbf{gcd}(\mathbf{incl}, \mathbf{incn}),$$

where $\mathbf{gcd}(\dots)$ is the greatest common divisor.

Example 1 Compute the forward discrete Fourier transform of 256 data sets data of length 1024, stored as columns of the COMPLEX*8 array **Z** whose dimensions are 1025 by 256.

```

INTEGER*4 L, INCL, N, INCN, IOPT, IER
COMPLEX*8 Z(1025, 256)
L = 1024
INCL = 1
N = 256
INCN = 1025
IOPT = 1
CALL CRCFTS (Z, L, INCL, N, INCN, IOPT, IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
END IF

```

Example 2 Compute the inverse discrete Fourier transform of 1024 sets of conjugate-symmetric complex data of length 256, stored as the rows of COMPLEX*8 array Z whose dimensions are 1025 by 256.

```
INTEGER*4 L,INCL,N,INCN,IOPT,IER
COMPLEX*8 Z(1025,256)
L = 256
INCL = 1025
N = 1024
INCN = 1
IOPT = -1
CALL CRCFTS (Z,L,INCL,N,INCN,IOPT,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
END IF
```

Name SRCFTS/DRCFTS
Simultaneous Real-to-Complex 1-D FFT

Purpose Given a number of one-dimensional real data sets, these subprograms compute nonredundant portions of all of their one-dimensional forward real-to-complex discrete Fourier transforms using a radix 2-3-5 fast Fourier transform (FFT) algorithm optimized for real input. Alternatively, given the nonredundant parts of a number of conjugate-symmetric one-dimensional complex data sets, these subprograms compute the inverse complex-to-real discrete Fourier transform using a radix 2-3-5 FFT algorithm optimized for real output. A pair of companion subprograms, CRCFTS and ZRCFTS, performs similar operations, but with the real or complex data presented in a complex array. These companion subprograms require more storage than the ones described here. Other subprograms, documented elsewhere in this chapter, are more suited for computing just one real-to-complex or complex-to-real transform.

The one-dimensional forward discrete Fourier transform of a real data set $z(n)$, for $n = 1, 2, \dots, l$, is defined by

$$Z(m) = \sum_{n=1}^l z(n) e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, \dots, l$ and $i = \sqrt{-1}$.

The sequence $Z(m)$ is conjugate-symmetric about $Z(l/2+1)$, i.e.,

$$\text{Im}(Z(1)) = \text{Im}(Z(l/2+1)) = 0$$

and

$$Z(l/2+1+m) = \bar{Z}(l/2+1-m), m = 1, 2, \dots, l/2-1,$$

where $\bar{Z}$ is the complex conjugate of Z . Therefore, the nonredundant part consists of the first $l/2+1$ elements of Z . This is all of Z that is computed or stored.

Alternatively, if $Z(m)$, for $m = 1, 2, \dots, l$, is a conjugate-symmetric complex data set, the one-dimensional, real, scaled inverse discrete Fourier transform of $Z(m)$ is defined by

$$z(n) = \frac{1}{l} \sum_{m=1}^l Z(m) e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, \dots, l$. Only the nonredundant part of Z is used.

These subprograms perform forward real-to-complex or inverse complex-to-real transform operations simultaneously on a number of data sets. They require

that the length l of the data sets be a product of powers of 2, 3, and 5, 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.

Usage

VECLIB:

```

      INTEGER*4      l, incl, n, incn, iopt, ier
      REAL*4         x(lenx)
      CALL SRCFTS(x, l, incl, n, incn, iopt, ier)

      INTEGER*4      l, incl, n, incn, iopt, ier
      REAL*8         x(lenx)
      CALL DRCFTS(x, l, incl, n, incn, iopt, ier)

```

VECLIB8:

```

      INTEGER*8      l, incl, n, incn, iopt, ier
      REAL*8         x(lenx)
      CALL SRCFTS(x, l, incl, n, incn, iopt, ier)

```

Input

x Array containing **n** one-dimensional data sets, each consisting of **l** real data points or the first $1/2+1$ 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

$$\text{lenx} = (l + 1) \times \text{incl} + (n - 1) \times \text{incn} + 1.$$

For a forward real-to-complex transform, the i -th real data point of the j -th data set, $1 \leq i \leq l$, $1 \leq j \leq n$, is stored in

$$\mathbf{x}((i - 1) \times \text{incl} + (j - 1) \times \text{incn} + 1).$$

For an inverse complex-to-real transform, the real part of the i -th data point of the j -th data set, $1 \leq i \leq 1/2+1$, $1 \leq j \leq n$, is stored in

$$\mathbf{x}((2 \times i - 2) \times \text{incl} + (j - 1) \times \text{incn} + 1)$$

and the imaginary part is stored in

$$\mathbf{x}((2 \times i - 1) \times \mathbf{incl} + (j - 1) \times \mathbf{incn} + 1)$$

respectively.

l	Number of data points in each complete data set, of the form $l = 2^p 3^q 5^r$, with $q, r \geq 0$ and either $l = 1$ or $p \geq 1$.
incl	Storage increment between successive elements of the same data set, incl > 0. incl = 1 means that the data points of a real data set are stored contiguously in array x , or that the real and imaginary parts of the data points of a complex data set are stored alternately in contiguous elements of x .
n	The number of data sets, n > 0.
incn	Storage increment between corresponding data points of successive data sets, incn > 0.
iopt	Option flag: iopt ≥ 0 Compute forward transform. iopt < 0 Compute inverse transform.

Output

x	The transformed data replaces the input if ier = 0 is returned. Unchanged if ier < 0. For a forward real-to-complex transform, the real part of the <i>i</i> -th output point of the <i>j</i> -th data set, $1 \leq i \leq l/2 + 1$, $1 \leq j \leq n$, is stored in

$$\mathbf{x}((2 \times i - 2) \times \mathbf{incl} + (j - 1) \times \mathbf{incn} + 1)$$

and the imaginary part is stored in

$$\mathbf{x}((2 \times i - 1) \times \mathbf{incl} + (j - 1) \times \mathbf{incn} + 1),$$

respectively. If needed, the remaining $(l/2 - 1) \times n$ complex output values may be formed by using the conjugate-symmetry condition.

For an inverse complex-to-real transform, the *i*-th real output point of the *j*-th data set, $1 \leq i \leq l$, $1 \leq j \leq n$, is stored in

$$\mathbf{x}((i - 1) \times \mathbf{incl} + (j - 1) \times \mathbf{incn} + 1).$$

ier	Status response: ier = 0 Normal return—transform successful. ier = -1 1 not of the required form.

ier = -2	incl ≤ 0.
ier = -3	n ≤ 0.
ier = -4	incn ≤ 0.
ier = -5	l , incl , n , and incn are incompatible. Refer to "Notes."

Notes

Typically, **x** will be a two- or three-dimensional array with each set of data being a one-dimensional section of the array, 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 $l+2 \leq ldx$, $n \leq mdx$, **incl** = 1, and **incn** = **ldx**. For example,

CALL SRCFTS (x, l, 1, n, ldxy, iopt, ier)

If **x** is a two-dimensional array as above and the data sets are stored in the rows of **x**, then $l+2 \leq mdx$, $n \leq ldx$, **incl** = **ldx**, and **incn** = 1. For example,

CALL SRCFTS (x, l, ldxy, n, 1, iopt, ier)

If **x** is a three-dimensional array of dimension **ldx** by **mdx** by **ndx**, then **incl** and **incn** will usually be 1, **ldx**, or **ldx**×**mdx**, depending on which of the subscripts of the three-dimensional array varies within a data set, which subscript varies between data sets, and which remains constant. Specifically, if the subscript that varies within a data set is the

1st	subscript, use	incl = 1.
2nd	subscript, use	incl = ldx .
3rd	subscript, use	incl = ldx × mdx .

Similarly, if the subscript that varies between data sets is the

1st	subscript, use	incn = 1.
2nd	subscript, use	incn = ldx .
3rd	subscript, use	incn = ldx × mdx .

l, **incl**, **n**, and **incn** must be such that no two points of any data sets occupy the same elements of **x**. These subprograms detect this situation and return **ier** = -5 if

$$\text{incl} < n \times \text{gcd}(\text{incl}, \text{incn})$$

and

$$\text{incn} < (l+2) \times \text{gcd}(\text{incl}, \text{incn}),$$

where **gcd**(...) is the greatest common divisor.

Example 1 Compute the forward real-to-complex discrete Fourier transform of 256 real data sets of length 1024. The real input data sets are stored as columns of REAL*4 array X whose dimensions are 1027 by 256. The complex output data sets are stored as columns of array X, with the real parts in rows 1, 3, 5, ..., 1025, and the imaginary parts in rows 2, 4, 6, ..., 1026.

```

INTEGER*4 L, INCL, N, INCN, IOPT, IER
REAL*4    X(1027,256)
L = 1024
INCL = 1
N = 256
INCN = 1027
IOPT = 1
CALL SRCFTS (X,L,INCL,N,INCN,IOPT,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
END IF

```

Example 2 Compute the inverse complex-to-real discrete Fourier transform of 1024 sets of conjugate-symmetric complex data length 256. The real and imaginary parts of the first 129 complex data points of the input data sets are stored as the rows of array X whose dimensions are 1025 by 258, with the real parts in columns 1, 3, 5, ..., 257, and the imaginary parts in columns 2, 4, 6, ..., 258. The real output data sets will be stored by row in the first 256 columns of X.

```

INTEGER*4 L, INCL, N, INCN, IOPT, IER
REAL*4    X(1025,258)
L = 256
INCL = 1025
N = 1024
INCN = 1
IOPT = -1
CALL SRCFTS (X,L,INCL,N,INCN,IOPT,IER)
IF ( IER .NE. 0 ) THEN
    handle error condition
END IF

```


10 Correlation and Convolution Subprograms

Overview

This chapter explains how to use the VECLIB subprograms available for correlations, convolutions, and related operations such as filtering by means of convolutions.

The following document provides supplemental material for this chapter:

Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc. 1975.

Chapter Objectives

After reading this chapter you will

- Understand the relationship between the described subprograms and subprograms in the FPS AP120B APLIB mathematical software library
- Know how use the described subprograms to compute correlation and convolution

What You Need to Know to Use These Subprograms

The subprograms in this section are patterned after the corresponding subprograms in the FPS AP120B APLIB mathematical software library.

Subprograms Included in This Chapter

Following are the subprograms included with VECLIB for correlations, convolutions, and related operations.

Name SCONV/DCONV
Correlation and Convolution

Purpose These subprograms compute the fully engaged portion of the discrete correlation or discrete convolution of two real data sequences. They can be used to compute the complete discrete correlation or convolution (the fully engaged portion plus the *tails*) by appending zeros to the ends of the longer of the operand vectors. Refer to "Example 2."

If $x_i, i = 1, 2, \dots, l$ and $w_j, j = 1, 2, \dots, n$ are two data sequences, their discrete correlation y_k is defined by

$$y_k = \sum_i w_{i-k} x_i,$$

for $k = -n+1, -n+2, \dots, l-1$, where the sum is taken over all indices i for which both w_{i-k} and x_i are defined.

If $x_i, i = 1, 2, \dots, l$ and $w_j, j = 1, 2, \dots, n$ are two data sequences, their discrete convolution z_k is defined by

$$z_k = \sum_j w_{k-i+1} x_i,$$

for $k = 1, 2, \dots, l+n-1$, where the sum is taken over all indices i for which both w_{k-i+1} and x_i are defined.

These subprograms compute only the fully engaged portion of the correlation or convolution, i.e., the part where the sums have exactly $\min(l, n)$ terms. Hence, if $l \geq n$, they compute

$$\tilde{y}_k = \sum_{i=1}^n w_{i-k} x_i,$$

for $k = 1, 2, \dots, m$, where $m=l-n+1$. This is the correlation operation if w is stored or indexed in the same direction as x and is the convolution operation if x and w are stored or indexed in opposite directions.

Usage VECLIB:

```
INTEGER*4      incx, incw, incy, m, n
REAL*4         x(lenx), w(lenw), y(leny)
CALL SCONV(x, incx, w, incw, y, incy, m, n)
```

```
INTEGER*4      incx, incw, incy, m, n
REAL*8         x(lenx), w(lenw), y(leny)
CALL DCONV(x, incx, w, incw, y, incy, m, n)
```

VECLIB8:

```

      INTEGER*8      incx, incw, incy, m, n
      REAL*8         x(lenx), w(lenw), y(leny)
      CALL SCONV(x, incx, w, incw, y, incy, m, n)

```

Input	x Array containing the operand (or trace) vector x of length $m+n-1$. $\text{lenx} = (m+n-2) \times \text{incx} + 1$.
incx	Increment for the array x , $\text{incx} \neq 0$. x_i is stored in $x((i-1) \times \text{incx} + 1)$. incx is normally positive whether computing the correlation or the convolution. Use $\text{incx} = 1$ if the vector x is stored contiguously in array x , i.e., if x_i is stored in $x(i)$. Refer to "Notes."
w	Array containing the operator (or kernel) vector w of length n . $\text{lenw} = (n-1) \times \text{incw} + 1$. Refer to the description of incw for alternate usage of w .
incw	Increment for the array w , $\text{incw} \neq 0$. w_j is stored in $w((j-1) \times \text{incw} + 1)$. incw is normally positive to compute the correlation and negative to compute the convolution. Use $\text{incw} = 1$ if computing the correlation and vector w is stored contiguously in forward order in array w , i.e., if w_i is stored in $w(i)$. Refer to "Example 1." Also, use $\text{incw} = 1$ if computing the convolution and vector w is stored contiguously in backward order in array w , i.e., if w_i is stored in $w(n+1-i)$. Use $\text{incw} = -1$ if computing the convolution and vector w is stored contiguously in forward order in array w , i.e., if w_i is stored in $w(i)$. In this case, in order to index backward through the w array, the w argument is usually coded as $w(n)$. Refer to "Notes" and "Example 2."
incy	Increment for the array y , $\text{incy} \neq 0$. $\tilde{y}_k$ is stored in $y((k-1) \times \text{incy} + 1)$. incy will normally be positive whether computing the correlation or the convolution. Use $\text{incy} = 1$ if the vector y is stored contiguously in array y , i.e., if $\tilde{y}_i$ is stored in $y(i)$. Refer to "Notes."
m	The length of the y vector. $m > 0$.

	n	The length of the w vector. $n > 0$.
Output	y	The convolution or correlation vector y of length m . $\text{leny} = (m-1) \times \text{incy} + 1$.

Notes Except for the difference in names, allowing for single- and double-precision, these subprograms are compatible with the CONV subprogram in the FPS AP120B APLIB mathematical software library.

These subprograms do not use the BLAS indexing scheme described in "BLAS Indexing Conventions" in the introduction to Chapter 2. Indexing works the same way as the BLAS for a positive increment but, for a negative increment, these subprograms access data backward from the beginning of the array passed to it rather than from the end. To index backward through an array, as desired for a convolution, the ending location of the array should be passed to the subprogram. Refer to "Example 2."

To compute the complete correlation or convolution vector, including both tails as well as the fully engaged portion, append $n-1$ zeros to each end of the x vector. The fully engaged portion of the correlation or convolution of the resulting vector is the complete correlation or convolution corresponding to the original x vector. Refer to "Example 2."

Example 1 Compute the fully engaged portion of the discrete correlation of the REAL*4 vectors $x = (4, 1, 3, 5, 2)$ and $w = (2, 1)$ stored in arrays X and W, respectively. In this instance, $n = 2$ and $m+n-1 = 5$, so $m = 4$.

```

INTEGER*4 INCX, INCW, INCY, M, N
REAL*4     X(5), W(2), Y(4)
DATA       X / 4.0 , 1.0 , 3.0 , 5.0, 2.0 /
DATA       W / 2.0 , 1.0 /
M = 4
N = 2
INCX = 1
INCW = 1
INCY = 1
CALL SCONV (X, INCX, W, INCW, Y, INCY, M, N)

```

The result is $y = (9, 5, 11, 12)$.

Example 2 Compute the complete discrete convolution of the REAL*4 vectors $x = (1, 3, 5)$ and $w = (2, 1)$ stored in arrays X and W, respectively. To get the complete convolution, append $n-1 = 1$ zeros to each end of x , getting $\tilde{x} = (0, 1, 3, 5, 0)$. Then, you have $n = 2$ and $m+n-1 = 5$, so $m = 4$.

```
INTEGER*4 INCX, INCW, INCY, M, N
REAL*4    X(5), W(2), Y(4)
DATA      X / 0.0 , 1.0 , 3.0 , 5.0, 0.0 /
DATA      W / 2.0 , 1.0 /
M = 4
N = 2
INCX = 1
INCW = -1
INCY = 1
CALL SCONV (X, INCX, W(N), INCW, Y, INCY, M, N)
```

The result is $y = (2, 7, 13, 5)$.

Name STCONV/DTCONV
Tapered Correlation and Convolution

Purpose These subprograms compute the discrete tapered correlation or discrete tapered convolution of two real data sequences.

If x_i and w_i , $i = 1, 2, \dots, n$ are two data sequences, their discrete correlation y_k is defined by

$$y_k = \sum_i x_{k+i-1} w_i,$$

for $k = -n+2, -n+3, \dots, n$, where the sum is taken over all indices i for which both x_{k+i-1} and w_i are defined.

Again, if x_i and w_i , $i = 1, 2, \dots, n$ are two data sequences, their discrete convolution z_k is defined by

$$z_k = \sum_i x_{k-i+1} w_i,$$

for $k = 1, 2, \dots, 2n-1$, where the sum is taken over all indices i for which both x_{k-i+1} and w_i are defined.

These subprograms compute only m elements on one side of the correlation or convolution, specifically the part where $l \leq k \leq m$. Hence, they compute

$$\tilde{y}_k = \sum_{i=1}^{n-k+1} x_{k+i-1} w_i,$$

for $k = 1, 2, \dots, m$. If $m > n$, $\tilde{y}_k = 0$ for $k = n+1, n+2, \dots, m$. This is the tapered correlation operation if w is stored or indexed in the same direction as x , and is the tapered convolution operation if x and w are stored or indexed in opposite directions.

Usage VECLIB:

```

      INTEGER*4      incx, incw, incy, m, n
      REAL*4         x(lenx), w(lenw), y(leny)
      CALL STCONV(x, incx, w, incw, y, incy, m, n)

      INTEGER*4      incx, incw, incy, m, n
      REAL*8         x(lenx), w(lenw), y(leny)
      CALL DTCONV(x, incx, w, incw, y, incy, m, n)

```

VECLIB8:

```

      INTEGER*8      incx, incw, incy, m, n
      REAL*8         x(lenx), w(lenw), y(leny)
      CALL STCONV(x, incx, w, incw, y, incy, m, n)

```

Input	x Array containing the operand (or trace) vector x of length n . $\text{lenx} = (n-1) \times \text{incx} + 1$.
	incx Increment for the array x , $\text{incx} \neq 0$. x_i is stored in $x((i-1) \times \text{incx} + 1)$. incx is normally positive whether computing the correlation or the convolution. Use $\text{incx} = 1$ if the vector x is stored contiguously in array x , i.e., if x_i is stored in $x(i)$. Refer to "Notes."
	w Array containing the operator (or kernel) vector w of length n . $\text{lenw} = (n-1) \times \text{incw} + 1$. Refer to the description of incw for alternate usage of w .
	incw Increment for the array w , $\text{incw} \neq 0$. w_j is stored in $w((j-1) \times \text{incw} + 1)$. incw is normally positive to compute the correlation and negative to compute the convolution. Use $\text{incw} = 1$ if computing the correlation and vector w is stored contiguously in forward order in array w ; i.e., if w_i is stored in $w(i)$. Refer to "Example 1." Also, use $\text{incw} = 1$ if computing the convolution and vector w is stored contiguously in backward order in array w , i.e., if w_i is stored in $w(n+1-i)$. Use $\text{incw} = -1$ if computing the convolution and vector w is stored contiguously in forward order in array w , i.e., if w_i is stored in $w(i)$. In this case, in order to index backward through the w array, the w argument is usually coded as $w(n)$. Refer to "Notes" and "Example 2."
	incy Increment for the array y , $\text{incy} \neq 0$. y_k is stored in $y((k-1) \times \text{incy} + 1)$. incy will normally be positive whether computing the correlation or the convolution. Use $\text{incy} = 1$ if the vector y is stored contiguously in array y , i.e., if y_i is stored in $y(i)$. Refer to "Notes."
	m The length of the y vector. $m > 0$.

	n	The length of the x and w vector. $n > 0$.
Output	y	The tapered convolution or correlation vector y of length m . $\text{leny} = (m-1) \times \text{incy} + 1$.
Notes	These subprograms do not use the BLAS indexing scheme described in "BLAS Indexing Conventions" in the introduction to Chapter 2. Indexing works the same way as the BLAS for a positive increment, but for a negative increment, these subprograms access data backward from the beginning of the array passed to it rather than from the end. To index backward through an array, as desired for a convolution, the ending location of the array should be passed to the subprogram. Refer to "Example 2."	

Example 1 Compute the discrete tapered correlation of the REAL*4 vectors $x = (4, 1)$ and $w = (2, 1)$ stored in arrays X and W, respectively. In this instance, $n = 2$ and $m = 4$.

```

INTEGER*4 INCX, INCW, INCY, M, N
REAL*4     X(2), W(2), Y(4)
DATA      X / 4.0 , 1.0 /
DATA      W / 2.0 , 1.0 /
M = 4
N = 2
INCX = 1
INCW = 1
INCY = 1
CALL STCONV (X, INCX, W, INCW, Y, INCY, M, N)

```

The result is $\tilde{y} = (9, 2, 0, 0)$.

Example 2 Compute the discrete tapered convolution of the REAL*4 vectors $x = (1, 3)$ and $w = (2, 1)$ stored in arrays X and W, respectively.

```

INTEGER*4 INCX, INCW, INCY, M, N
REAL*4     X(2), W(2), Y(2)
DATA      X / 1.0 , 3.0 /
DATA      W / 2.0 , 1.0 /
M = 2
N = 2
INCX = 1
INCW = -1
INCY = 1
CALL STCONV (X, INCX, W(N), INCW, Y, INCY, M, N)

```

The result is $\tilde{y} = (7, 3)$.

11 Linear Recurrences

Overview

This chapter explains how to use VECLIB 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

Recurrence subprograms were written for optimizing tridiagonal and pentadiagonal linear equation solvers. Consider the LINPACK tridiagonal or band solvers described in Chapter 4 or the LAPACK solvers described in the *HP MLIB LAPACK User's Guide*.

Subprograms Included in This Chapter

Following are the subprograms included with VECLIB for linear recurrence operations.

Name SFLR1C/DFLR1C
First Order Linear Recurrence

Purpose Given a real coefficient α and a real vector x of length n , these subprograms solve the first order linear recurrence

$$y_1 = x_1 \quad y_i = x_i + \alpha y_{i-1}, i = 2, 3, \dots, n,$$

overwriting the input x vector with the resulting y vector.

The vector may be stored in a one-dimensional array or in either a row or a column of a two-dimensional array, and the indexing through the array may be either forward or backward.

Usage VECLIB:

```
INTEGER*4      n, incx
REAL*4         alpha, x(lenx)
CALL SFLR1C(n, alpha, x, incx)
```

```
INTEGER*4      n, incx
REAL*8         alpha, x(lenx)
CALL DFLR1C(n, alpha, x, incx)
```

VECLIB8:

```
INTEGER*8      n, incx
REAL*8         alpha, x(lenx)
CALL SFLR1C(n, alpha, x, incx)
```

Input

n	Number of elements of vectors x to be used in the recurrence, $n \geq 0$. If $n = 0$, the subprogram does not reference x .				
alpha	The scalar α .				
x	Array of length $\text{lenx} = (n-1) \times \text{incx} + 1$ containing the n -vector x .				
incx	Increment for the array x , $\text{incx} \neq 0$: <table> <tbody> <tr> <td>$\text{incx} > 0$</td> <td>x is stored forward in array x; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$.</td> </tr> <tr> <td>$\text{incx} < 0$</td> <td>$x$ is stored backward in array x, i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$.</td> </tr> </tbody> </table>	$\text{incx} > 0$	x is stored forward in array x ; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$.	$\text{incx} < 0$	x is stored backward in array x , i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$.
$\text{incx} > 0$	x is stored forward in array x ; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$.				
$\text{incx} < 0$	x is stored backward in array x , i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$.				

Use $\text{incx} = 1$ 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 If an error in the arguments is detected, the subprograms call error handler XERVEC, which writes an error message onto the standard error file and terminates execution. The standard version of XERVEC (refer to Chapter 12, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are

$n < 0$ and
 $\text{incx} = 0$.

Fortran Equivalent

```
SUBROUTINE SFLR1C (N, ALPHA, X, INCX)
  INTEGER*4 N, INCX
  REAL*4 ALPHA, X(*)
  IF ( N .LT. 0 ) THEN
    CALL XERVEC (...)
    RETURN
  ELSE IF ( INCX .EQ. 0 ) THEN
    CALL XERVEC (...)
    RETURN
  END IF
  IX = 1 + INCX
  IF ( INCX .LT. 0 ) IX = 1 - (N-2) * INCX
  DO 10 I = 2, N
    X(IX) = X(IX) + ALPHA * X(IX-INCX)
    IX = IX + INCX
  10 CONTINUE
  RETURN
END
```

Example Solve the REAL*8 first order linear recurrence

$$x_1 = a_1 \quad x_i = a_i + 4x_{i-1}, i = 2, 3, \dots, n,$$

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

```
INTEGER*4 N, INCX
REAL*8 ALPHA, X(20)
N = 10
ALPHA = 4.0
INCX = 1
CALL DFLR1C (N, ALPHA, X, INCX)
```

Name SFLR1M/SFLR1P/DFLR1M/DFLR1P
First Order Linear Recurrence

Purpose Given real vectors a and x of length n , these subprograms solve the first order linear recurrence

$$y_1 = x_1 \quad y_i = x_i \pm a_i y_{i-1}, i = 2, 3, \dots, n,$$

overwriting the input x vector with the resulting y vector.

The operation indicated by “ $\pm$ ” above is specified by the subprogram name used:

SFLR1M	or	DFLR1M	“ $\pm$ ” represents “ $-$ ”:	$y_i = x_i - a_i y_{i-1}$
SFLR1P	or	DFLR1P	“ $\pm$ ” 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 VECLIB:

```
INTEGER*4      n, inca, incx
REAL*4         a(lena), x(lenx)
CALL SFLR1M(n, a, inca, x, incx)
```

```
INTEGER*4      n, inca, incx
REAL*4         a(lena), x(lenx)
CALL SFLR1P(n, a, inca, x, incx)
```

```
INTEGER*4      n, inca, incx
REAL*8         a(lena), x(lenx)
CALL DFLR1M(n, a, inca, x, incx)
```

```
INTEGER*4      n, inca, incx
REAL*8         a(lena), x(lenx)
CALL DFLR1P(n, a, inca, x, incx)
```

VECLIB8:

```
INTEGER*8      n, inca, incx
REAL*8         a(lena), x(lenx)
CALL SFLR1M(n, a, inca, x, incx)
```

```
INTEGER*8      n, inca, incx
REAL*8         a(lena), x(lenx)
CALL SFLR1P(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 $\text{lena} = (n-1) \times \text{inca} + 1$ containing the n -vector a .
	inca	Increment for the array a , $\text{inca} \neq 0$: $\text{inca} > 0$ a is stored forward in array a; i.e., a_i is stored in $a((i-1) \times \text{inca} + 1)$. $\text{inca} < 0$ a is stored backward in array a; i.e., a_i is stored in $a((i-n) \times \text{inca} + 1)$. Use $\text{inca} = 1$ 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 $\text{lenx} = (n-1) \times \text{incx} + 1$ containing the n -vector x .
	incx	Increment for the array x , $\text{incx} \neq 0$: $\text{incx} > 0$ x is stored forward in array x; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$. $\text{incx} < 0$ x is stored backward in array x; i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$. Use $\text{incx} = 1$ 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 XERVEC, which writes an error message onto the standard error file and terminates execution. The standard version of XERVEC (refer to Chapter 12, 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 SFLR1M (N, A, INCA, X, INCX)
  INTEGER*4 N, INCA, INCX
  REAL*4 A(*), X(*)
  IF ( N .LT. 0 ) THEN
    CALL XERVEC (...)
    RETURN
  ELSE IF ( INCA .EQ. 0 ) THEN
    CALL XERVEC (...)
    RETURN
  ELSE IF ( INCX .EQ. 0 ) THEN
    CALL XERVEC (...)
    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

$$y_1 = x_1 \quad y_i = x_i - a_i y_{i-1}, \quad i = 2, 3, \dots, n,$$

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*4 N, INCA, INCX
REAL*8 A(20), X(20)
N = 10
INCA = 1
INCX = 1
CALL DFLR1M (N, A, INCA, X, INCX)

```


Name SFLR2C/DFLR2C
First Order Linear Recurrence

Purpose Given a real coefficient α and a real vector x of length n , these subprograms solve the first order linear recurrence

$$y_1 = x_1 \quad y_i = x_i + \alpha y_{i-1}, i = 2, 3, \dots, n.$$

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 VECLIB:

```

      INTEGER*4      n, incx, incy
      REAL*4         alpha, x(lenx), y(leny)
      CALL SFLR2C(n, alpha, x, incx, y, incy)

      INTEGER*4      n, incx, incy
      REAL*8         alpha, x(lenx), y(leny)
      CALL DFLR2C(n, alpha, x, incx, y, incy)

```

VECLIB8:

```

      INTEGER*8      n, incx, incy
      REAL*8         alpha, x(lenx), y(leny)
      CALL SFLR2C(n, alpha, x, incx, y, incy)

```

Input

n Number of elements of vectors x and y to be used in the recurrence, $n \geq 0$. If $n = 0$, the subprogram does not reference x or y .

alpha The scalar α .

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

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

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

incx < 0 x is stored backward in array x ; i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$.

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

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** = $(n-1) \times |\text{incy}| + 1$ containing the n -vector **y**. If **n** = 0, then **y** is unchanged. Otherwise, the recurrence's solution vector is returned.

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

If an error in the arguments is detected, the subprograms call error handler XERVEC, which writes an error message onto the standard error file and terminates execution. The standard version of XERVEC (refer to Chapter 12, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are

n < 0,
incx = 0, and
incy = 0.

**Fortran
Equivalent**

```

SUBROUTINE SFLR2C (N, ALPHA, X, INCX, Y, INCY)
  INTEGER*4 N, INCX, INCY
  REAL*4 ALPHA, X(*), Y(*)
  IF ( N .LT. 0 ) THEN
    CALL XERVEC (...)
    RETURN
  ELSE IF ( INCX .EQ. 0 ) THEN
    CALL XERVEC (...)
    RETURN
  ELSE IF ( INCY .EQ. 0 ) THEN
    CALL XERVEC (...)
    RETURN
  END IF
  IX = 1 + INCX
  IY = 1 + INCY
  IF ( INCX .LT. 0 ) IX = 1 - (N-2) * INCX
  IF ( INCY .LT. 0 ) IY = 1 - (N-2) * INCY
  X(IX-INCX) = Y(IY-INCY)
  DO 10 I = 2, N
    X(IX) = Y(IY) + ALPHA * X(IX-INCX)
    IX = IX + INCX
    IY = IY + INCY
  10 CONTINUE
  RETURN
END

```

Example Solve the REAL*8 first order linear recurrence

$$x_1 = a_1 \quad x_i = a_i + 4x_{i-1}, i = 2, 3, \dots, n,$$

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

```

INTEGER*4 N, INCX, INCY
REAL*8    ALPHA, X(20), Y(20)
N = 10
ALPHA = 4.0
INCX = 1
INCY = 1
CALL DFLR2C (N, ALPHA, X, INCX, Y, INCY)

```

Name SFLR2M/SFLR2P/DFLR2M/DFLR2P
First Order Linear Recurrence

Purpose Given real vectors a and x of length n , these subprograms solve the first order linear recurrence

$$y_1 = x_1 \quad y_i = x_i \pm a_i y_{i-1}, i = 2, 3, \dots, n,$$

for the y vector.

The operation indicated by “ $\pm$ ” above is specified by the subprogram name used:

SFLR2M	or	DFLR2M	“ $\pm$ ” represents “ $-$ ”:	$y_i = x_i - a_i y_{i-1}$
SFLR2P	or	DFLR2P	“ $\pm$ ” 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 VECLIB:

```
INTEGER*4      n, inca, incx, incy
REAL*4         a(lena), x(lenx), y(leny)
CALL SFLR2M(n, a, inca, x, incx, y, incy)
```

```
INTEGER*4      n, inca, incx, incy
REAL*4         a(lena), x(lenx), y(leny)
CALL SFLR2P(n, a, inca, x, incx, y, incy)
```

```
INTEGER*4      n, inca, incx, incy
REAL*8         a(lena), x(lenx), y(leny)
CALL DFLR2M(n, a, inca, x, incx, y, incy)
```

```
INTEGER*4      n, inca, incx, incy
REAL*8         a(lena), x(lenx), y(leny)
CALL DFLR2P(n, a, inca, x, incx, y, incy)
```

VECLIB8:

```
INTEGER*8      n, inca, incx, incy
REAL*8         a(lena), x(lenx), y(leny)
CALL SFLR2M(n, a, inca, x, incx, y, incy)
```

```
INTEGER*8      n, inca, incx, incy
REAL*8         a(lena), x(lenx), y(leny)
CALL SFLR2P(n, a, inca, x, incx, y, incy)
```

Input	n	Number of elements of vectors a , x , and y to be used in the recurrence, $n \geq 0$. If $n = 0$, the subprograms do not reference a , x , or y .
	a	Array of length $\text{lena} = (n-1) \times \text{inca} + 1$ containing the n -vector a .
	inca	Increment for the array a , $\text{inca} \neq 0$: $\text{inca} > 0$ a is stored forward in array a ; i.e., a_i is stored in $a((i-1) \times \text{inca} + 1)$. $\text{inca} < 0$ a is stored backward in array a ; i.e., a_i is stored in $a((i-n) \times \text{inca} + 1)$. Use $\text{inca} = 1$ 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 $\text{lenx} = (n-1) \times \text{incx} + 1$ containing the n -vector x .
	incx	Increment for the array x , $\text{incx} \neq 0$: $\text{incx} > 0$ x is stored forward in array x ; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$. $\text{incx} < 0$ x is stored backward in array x ; i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$. Use $\text{incx} = 1$ 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 , $\text{incy} \neq 0$: $\text{incy} > 0$ y is stored forward in array y ; i.e., y_i is stored in $y((i-1) \times \text{incy} + 1)$. $\text{incy} < 0$ y is stored backward in array y ; i.e., y_i is stored in $y((i-n) \times \text{incy} + 1)$. Use $\text{incy} = 1$ 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** = $(n-1) \times |\text{incy}| + 1$ 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 XERVEC, which writes an error message onto the standard error file and terminates execution. The standard version of XERVEC (refer to Chapter 12, 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.

Fortran Equivalent

```

SUBROUTINE SFLR2M (N, A, INCA, X, INCX, Y, INCY)
  INTEGER*4 N, INCA, INCX, INCY
  REAL*4 A(*), X(*)
  IF ( N .LT. 0 ) THEN
    CALL XERVEC (...)
    RETURN
  ELSE IF ( INCA .EQ. 0 ) THEN
    CALL XERVEC (...)
    RETURN
  ELSE IF ( INCX .EQ. 0 ) THEN
    CALL XERVEC (...)
    RETURN
  ELSE IF ( INCY .EQ. 0 ) THEN
    CALL XERVEC (...)
    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

$$y_1 = x_1 \quad y_i = x_i - a_i y_{i-1}, i = 2, 3, \dots, n,$$

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

```
INTEGER*4 N, INCA, INCX, INCY
REAL*8    A(20), X(20), Y(20)
N = 10
INCA = 1
INCX = 1
INCY = 1
CALL DFLR2M (N, A, INCA, X, INCX, Y, INCY)
```

Name SFLRLM/.../DFLRLP
Last Term of First Order Linear Recurrence

Purpose Given real vectors a and x of length n , these subprograms solve for the last term of the first order linear recurrence

$$y_1 = x_1 \quad y_i = x_i \pm a_i y_{i-1}, i = 2, 3, \dots, n,$$

i.e., returning y_n .

The operation indicated by " $\pm$ " above is specified by the subprogram name used:

SFLRLM or DFLRLM " $\pm$ " represents " $-$ ": $y_i = x_i - a_i y_{i-1}$

SFLRLP or DFLRLP " $\pm$ " 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 VECLIB:

```
INTEGER*4      n, inca, incx
REAL*4         yn, SFLRLM, a(lena), x(lenx)
yn = SFLRLM(n, a, inca, x, incx)
```

```
INTEGER*4      n, inca, incx
REAL*4         yn, SFLRLP, a(lena), x(lenx)
yn = SFLRLP(n, a, inca, x, incx)
```

```
INTEGER*4      n, inca, incx
REAL*8         yn, DFLRLM, a(lena), x(lenx)
yn = DFLRLM(n, a, inca, x, incx)
```

```
INTEGER*4      n, inca, incx
REAL*8         yn, DFLRLP, a(lena), x(lenx)
yn = DFLRLP(n, a, inca, x, incx)
```

VECLIB8:

```
INTEGER*8      n, inca, incx
REAL*8         yn, SFLRLM, a(lena), x(lenx)
yn = SFLRLM(n, a, inca, x, incx)
```

```
INTEGER*8      n, inca, incx
REAL*8         yn, SFLRLP, a(lena), x(lenx)
yn = SFLRLP(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 $\text{lena} = (n-1) \times \text{inca} + 1$ containing the n -vector a .
	inca	Increment for the array a , $\text{inca} \neq 0$: $\text{inca} > 0$ a is stored forward in array a; i.e., a_i is stored in $a((i-1) \times \text{inca} + 1)$. $\text{inca} < 0$ a is stored backward in array a; i.e., a_i is stored in $a((i-n) \times \text{inca} + 1)$. Use $\text{inca} = 1$ 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 $\text{lenx} = (n-1) \times \text{incx} + 1$ containing the n -vector x .
	incx	Increment for the array x : $\text{incx} \geq 0$ x is stored forward in array x; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$. $\text{incx} < 0$ x is stored backward in array x; i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$. If $\text{incx} = 0$, then $x_i = x(1)$ for all i. Refer to "Notes" to see how to use SFLRLP with $\text{incx} = 0$ to evaluate a polynomial. Use $\text{incx} = 1$ 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.
	yn	If $n = 0$, then $\text{yn} = 0$. Otherwise, the last term of the recurrence's solution is returned.
Output		

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**.

SFLRLP 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

$$y_0 = a_n \quad y_i = a_{n-i} + y_{i-1}x, i = 1, 2, \dots, n.$$

Refer to "Example 2".

If an error in the arguments is detected, the subprograms call error handler XERVEC, which writes an error message onto the standard error file and terminates execution. The standard version of XERVEC (refer to Chapter 12, 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**

```

REAL*4 FUNCTION SFLRLM (N, A, INCA, X, INCX)
INTEGER*4 N, INCA, INCX
REAL*4 A(*), X(*)
FOLRN = 0.0
IF ( N .LT. 0 ) THEN
    CALL XERVEC (...)
    RETURN
ELSE IF ( INCA .EQ. 0 ) THEN
    CALL XERVEC (...)
    RETURN
ELSE IF ( INCX .EQ. 0 ) THEN
    CALL XERVEC (...)
    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

$$y_1 = x_1 \quad y_i = x_i - a_i y_{i-1}, i = 2, 3, \dots, n,$$

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

```

INTEGER*4 N, INCA, INCX
REAL*8    YN, DFLRLM, A(20), X(20)
N = 10
INCA = 1
INCX = 1
YN = DFLRLM(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*4 N, INCA, INCX
REAL*8    P, DFLRLP, A(0:20), X
N = 11
INCA = -1
INCX = 0
P = DFLRLP(N, A, INCA, X, INCX)

```

Name SPPROD/DPPROD
Compute Vector of Partial Products

Purpose Given real vector x of length n , these subprograms compute the n -vector y of partial products

$$x_i = \prod_{j=1}^i a_j, i = 1, 2, \dots, n.$$

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 VECLIB:

```

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

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

```

VECLIB8:

```

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

```

Input

n Number of elements of vectors x and y , $n \geq 0$. If $n = 0$, the subprograms do not reference x or y .

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

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

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

incx < 0 x is stored backward in array x ; i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$.

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

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 $\text{leny} = (n-1) \times |\text{incy}| + 1$ containing the n -vector **y**. If **n** = 0, then **y** is unchanged. Otherwise, the vector of partial products replaces the input.

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

If an error in the arguments is detected, the subprograms call error handler XERVEC, which writes an error message onto the standard error file and terminates execution. The standard version of XERVEC (refer to Chapter 12, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are

n < 0,
incx = 0, and
incy = 0.

**Fortran
Equivalent**

```

SUBROUTINE SPPROD (N, X, INCX, Y, INCY)
  INTEGER*4 N, INCX, INCY
  REAL*4 X(*), Y(*)
  IF ( N .LT. 0 ) THEN
    CALL XERVEC (...)
    RETURN
  ELSE IF ( INCX .EQ. 0 ) THEN
    CALL XERVEC (...)
    RETURN
  ELSE IF ( INCY .EQ. 0 ) THEN
    CALL XERVEC (...)
    RETURN
  END IF
  IX = 1 + INCX
  IY = 1 + INCY
  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) = Y(IY-INCY) * X(IX)
    IX = IX + INCX
    IY = IY + INCY
  10 CONTINUE
  RETURN
END

```

Example Compute the REAL*8 vector of partial products of the vector x , where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20. The result is stored in array Y , also of dimension 20.

```

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

```

Name SPSUM/DPSUM/IPSUM
Compute Vector of Partial Sums

Purpose Given real vector x of length n , these subprograms computes the n -vector y of partial sums

$$y_i = \sum_{j=1}^i x_j, i = 1, 2, \dots, n.$$

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 VECLIB:

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

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

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

VECLIB8:

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

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

Input

n	Number of elements of vectors x and y , $n \geq 0$. If $n = 0$, the subprograms do not reference x or y .
x	Array of length $\text{lenx} = (n-1) \times \text{incx} + 1$ containing the n -vector x .
incx	Increment for the array x , $\text{incx} \neq 0$: $\text{incx} > 0$ x is stored forward in array x; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$. $\text{incx} < 0$ x is stored backward in array x; i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$.

Use $\text{incx} = 1$ 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 , $\text{incy} \neq 0$:

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

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

Use $\text{incy} = 1$ 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 $\text{leny} = (n-1) \times |\text{incy}| + 1$ containing the n -vector y . If $n = 0$, then y is unchanged. Otherwise, the vector of partial sums replaces the input.

Notes

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

If an error in the arguments is detected, the subprograms call error handler XERVEC, which writes an error message onto the standard error file and terminates execution. The standard version of XERVEC (refer to Chapter 12, Miscellaneous Routines) may be replaced with a user-supplied version to change the error procedure. Error conditions are

$n < 0$,
 $\text{incx} = 0$, and
 $\text{incy} = 0$.

**Fortran
Equivalent**

```

SUBROUTINE SPSUM (N, X, INCX, Y, INCY)
  INTEGER*4 N, INCX, INCY
  REAL*4 X(*), Y(*)
  IF ( N .LT. 0 ) THEN
    CALL XERVEC (...)
    RETURN
  ELSE IF ( INCX .EQ. 0 ) THEN
    CALL XERVEC (...)
    RETURN
  ELSE IF ( INCY .EQ. 0 ) THEN
    CALL XERVEC (...)
    RETURN
  END IF
  IX = 1 + INCX
  IY = 1 + INCY
  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) = Y(IY-INCY) + X(IX)
    IX = IX + INCX
    IY = IY + INCY
  10 CONTINUE
  RETURN
END

```

Example

Compute the REAL*8 vector of partial sums of the vector x , where x is a vector 10 elements long stored in a one-dimensional array X of dimension 20. The result is stored in array Y, also of dimension 20.

```

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

```

Name SSLR2/DSLR2
Second Order Linear Recurrence

Purpose Given real vectors a and b of length n and the first two elements of n -vector x , these subprograms solve the second order linear recurrence

$$x_i = a_{i-2}x_{i-1} + b_{i-2}x_{i-2}, i = 3, 4, \dots, m.$$

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 VECLIB:

```
INTEGER*4      n, inca, incb, incx
REAL*4         a(lena), b(lenb), x(lenx)
CALL SSLR2(n, a, inca, b, incb, x, incx)
```

```
INTEGER*4      n, inca, incb, incx
REAL*8         a(lena), b(lenb), x(lenx)
CALL DSLR2(n, a, inca, b, incb, x, incx)
```

VECLIB8:

```
INTEGER*8      n, inca, incb, incx
REAL*8         a(lena), b(lenb), x(lenx)
CALL SSLR2(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 $\text{lena} = (n-1) \times |\text{inca}| + 1$ containing the n -vector a .

inca Increment for the array a , $\text{inca} \neq 0$:

inca > 0 a is stored forward in array a ; i.e., a_i is stored in $a((i-1) \times \text{inca} + 1)$.

inca < 0 a is stored backward in array a ; i.e., a_i is stored in $a((i-n) \times \text{inca} + 1)$.

Use $\text{inca} = 1$ 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 $\text{lenb} = (n-1) \times |\text{incb}| + 1$ containing the n -vector b .

incb	Increment for the array b , incb $\neq$ 0: incb > 0 b is stored forward in array b ; i.e., b_i is stored in $b((i-1) \times \text{incb} + 1)$. incb < 0 b is stored backward in array b ; i.e., b_i is stored in $b((i-n) \times \text{incb} + 1)$. 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 $\text{lenx} = (n-1) \times \text{incx} + 1$ 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 $x((i-1) \times \text{incx} + 1)$. incx < 0 x is stored backward in array x ; i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$. 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 XERVEC, which writes an error message onto the standard error file and terminates execution. The standard version of XERVEC (refer to Chapter 12, 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 SSLR2 (N, A, INCA, B, INCB, X, INCX)
  INTEGER*4 N, INCA, INCB, INCX
  REAL*4 A(*), B(*), X(*)
  IF ( N .LT. 0 ) THEN
    CALL XERVEC (...)
    RETURN
  ELSE IF ( INCA .EQ. 0 ) THEN
    CALL XERVEC (...)
    RETURN
  ELSE IF ( INCB .EQ. 0 ) THEN
    CALL XERVEC (...)
    RETURN
  ELSE IF ( INCX .EQ. 0 ) THEN
    CALL XERVEC (...)
    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

```

Example Solve the REAL*8 second order linear recurrence

$$x_i = a_{i-2}x_{i-1} + b_{i-2}x_{i-2}, i = 3, 4, \dots, n,$$

where a , b , and x are vectors 10 elements long stored in one-dimensional arrays A , B , and X of dimension 20.

```

INTEGER*4 N, INCA, INCB, INCX
REAL*8 A(20), B(20), X(20)
N = 10
INCA = 1
INCB = 1
INCX = 1
X(1) = ...
X(2) = ...
CALL DSLR2 (N, A, INCA, B, INCB, X, INCX)

```

Name SSLR3/DSLR3
 Second Order Linear Recurrence

Purpose Given real vectors a , b , and x of length n , these subprograms solves the second order linear recurrence

$$\begin{aligned} y_1 &= x_1 \\ y_2 &= x_2 \\ y_i &= x_i + a_{i-2}y_{i-1} + b_{i-2}y_{i-2}, i = 3, 4, \dots, n, \end{aligned}$$

overwriting the input x vector with the resulting y vector.

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 VECLIB:

```
INTEGER*4      n, inca, incb, incx
REAL*4         a(lena), b(lenb), x(lenx)
CALL SSLR3(n, a, inca, b, incb, x, incx)
```

```
INTEGER*4      n, inca, incb, incx
REAL*8         a(lena), b(lenb), x(lenx)
CALL DSLR3(n, a, inca, b, incb, x, incx)
```

VECLIB8:

```
INTEGER*8      n, inca, incb, incx
REAL*8         a(lena), b(lenb), x(lenx)
CALL SSLR3(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 $\text{lena} = (n-1) \times |\text{inca}| + 1$ containing the n -vector a .

inca Increment for the array a , $\text{inca} \neq 0$:

$\text{inca} > 0$ a is stored forward in array a ; i.e., a_i is stored in $a((i-1) \times \text{inca} + 1)$.

$\text{inca} < 0$ a is stored backward in array a ; i.e., a_i is stored in $a((i-n) \times \text{inca} + 1)$.

		Use inca = 1 if the vector <i>a</i> is stored contiguously in array a , i.e., if a_i is stored in a (<i>i</i>). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.
	b	Array of length $\text{lenb} = (n-1) \times \text{incb} + 1$ containing the <i>n</i> -vector <i>b</i> .
	incb	Increment for the array b , incb $\neq$ 0: incb > 0 <i>b</i> is stored forward in array b ; i.e., b_i is stored in b ((<i>i</i> -1) $\times$ incb +1). incb < 0 <i>b</i> is stored backward in array b ; i.e., b_i is stored in b ((<i>i</i> - n) $\times$ incb +1). Use incb = 1 if the vector <i>b</i> is stored contiguously in array b , i.e., if b_i is stored in b (<i>i</i>). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.
	x	Array of length $\text{lenx} = (n-1) \times \text{incx} + 1$ containing the <i>n</i> -vector <i>x</i> .
	incx	Increment for the array x , incx $\neq$ 0: incx > 0 <i>x</i> is stored forward in array x ; i.e., x_i is stored in x ((<i>i</i> -1) $\times$ incx +1). incx < 0 <i>x</i> is stored backward in array x ; i.e., x_i is stored in x ((<i>i</i> - n) $\times$ incx +1). Use incx = 1 if the vector <i>x</i> is stored contiguously in array x , i.e., if x_i is stored in x (<i>i</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 <i>a</i> , <i>b</i> , or <i>x</i> shares a memory location with any other element of <i>a</i> , <i>b</i> , or <i>x</i> . If an error in the arguments is detected, the subprograms call error handler XERVEC, which writes an error message onto the standard error file and terminates execution. The standard version of XERVEC (refer to Chapter 12, 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 SSLR3 (N, A, INCA, B, INCB, X, INCX)
  INTEGER*4 N, INCA, INCB, INCX
  REAL*4 A(*), B(*), X(*)
  IF ( N .LT. 0 ) THEN
    CALL XERVEC (...)
    RETURN
  ELSE IF ( INCA .EQ. 0 ) THEN
    CALL XERVEC (...)
    RETURN
  ELSE IF ( INCB .EQ. 0 ) THEN
    CALL XERVEC (...)
    RETURN
  ELSE IF ( INCX .EQ. 0 ) THEN
    CALL XERVEC (...)
    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

$$\begin{aligned}
 y_1 &= x_1 \\
 y_2 &= x_2 \\
 y_i &= x_i + a_{i-2}y_{i-1} + b_{i-2}y_{i-2}, \quad i = 3, 4, \dots, n
 \end{aligned}$$

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*4 N, INCA, INCB, INCX
REAL*8 A(20), B(20), X(20)
N = 10
INCA = 1
INCB = 1
INCX = 1
CALL DSLR3 (N, A, INCA, B, INCB, X, INCX)

```

Name SSLRL/DSLRL
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 , these subprograms solve for the last term of the second order linear recurrence

$$x_i = a_{i-2}x_{i-1} + b_{i-2}x_{i-2}, i = 3, 4, \dots, n,$$

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 VECLIB:

```
INTEGER*4      n, inca, incb, incx
REAL*4         xn, SSLRL, a(lena), b(lenb), x(lenx)
xn = SSLRL(n, a, inca, b, incb, x, incx)
```

```
INTEGER*4      n, inca, incb, incx
REAL*8         xn, DSLRL, a(lena), b(lenb), x(lenx)
xn = DSLRL(n, a, inca, b, incb, x, incx)
```

VECLIB8:

```
INTEGER*8      n, inca, incb, incx
REAL*8         xn, SSLRL, a(lena), b(lenb), x(lenx)
xn = SSLRL(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 $\text{lena} = (n-1) \times |\text{inca}| + 1$ containing the n -vector a .

inca Increment for the array a , $\text{inca} \neq 0$:

inca > 0 a is stored forward in array a ; i.e., a_i is stored in $a((i-1) \times \text{inca} + 1)$.

inca < 0 a is stored backward in array a ; i.e., a_i is stored in $a((i-n) \times \text{inca} + 1)$.

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 $\text{lenb} = (n-1) \times \text{incb} + 1$ containing the n -vector b .
	incb	Increment for the array b , $\text{incb} \neq 0$:
	incb > 0	b is stored forward in array b ; i.e., b_i is stored in $b((i-1) \times \text{incb} + 1)$.
	incb < 0	b is stored backward in array b ; i.e., b_i is stored in $b((i-n) \times \text{incb} + 1)$.
Use $\text{incb} = 1$ 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 $\text{lenx} = (n-1) \times \text{incx} + 1$ containing the first two elements, x_1 and x_2 of the n -vector x .
	incx	Increment for the array x , $\text{incx} \neq 0$:
	incx > 0	x is stored forward in array x ; i.e., x_i is stored in $x((i-1) \times \text{incx} + 1)$.
	incx < 0	x is stored backward in array x ; i.e., x_i is stored in $x((i-n) \times \text{incx} + 1)$.
Use $\text{incx} = 1$ 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 $\text{xn} = 0$. 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 XERVEC, which writes an error message onto the standard error file and terminates execution. The standard version of XERVEC (refer to Chapter 12, 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

```

REAL*4 FUNCTION SSLRL (N, A, INCA, B, INCB, X, INCX)
INTEGER*4 N, INCA, INCB, INCX
REAL*4 A(*), B(*), X(*)
SSLRL = 0.0
IF ( N .LT. 0 ) THEN
  CALL XERVEC (...)
  RETURN
ELSE IF ( INCA .EQ. 0 ) THEN
  CALL XERVEC (...)
  RETURN
ELSE IF ( INCB .EQ. 0 ) THEN
  CALL XERVEC (...)
  RETURN
ELSE IF ( INCX .EQ. 0 ) THEN
  CALL XERVEC (...)
  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
  SSLRL = X(IX-INCX)
ELSE
  SSLRL = X(IX)
END IF
RETURN
END
  
```

Example Solve for the last term of the REAL*8 second order linear recurrence

$$x_i = a_{i-2}x_{i-1} + b_{i-2}x_{i-2}, i = 3, 4, \dots, n,$$

where a , b , and x are vectors 10 elements long stored in one-dimensional arrays A, B, and X of dimension 20.

```

INTEGER*4 N, INCA, INCB, INCX
REAL*8    XN, DSLRL, A(20), B(20), X(20)
N = 10
INCA = 1
INCB = 1
INCX = 1
X(1) = ...
X(2) = ...
XN = DSLRL (N, A, INCA, B, INCB, X, INCX)

```


12 Miscellaneous Routines

Overview

This chapter explains how to use VECLIB subprograms for a variety of operations that are not included in other chapters. It includes a description of subprograms that:

- Control shared-memory parallelism on systems with multiple processors
- Measure CPU or wall-clock time
- Allocate nonvolatile dynamic memory
- Generate random numbers
- Sort the elements of a vector into ascending or descending order
- Report errors detected in the usage of VECLIB subprograms

The following documents provide supplemental material for this chapter:

Knuth, D.E. *The Art of Computer Programming*, Vol. 2: Seminumerical Algorithms. Menlo Park, California: Addison-Wesley. 1973.

Knuth, D.E. *The Art of Computer Programming*, Vol. 3: Sorting and Searching. Menlo Park, California: Addison-Wesley. 1973.

Chapter Objective

After reading this chapter you will know how to use the described subprograms.

What You Need to Know to Use These Subprograms

This chapter describes several dynamic memory subprograms:

- DALLOC
- NALLOC
- RALLOC

NALLOC, DALLOC, and RALLOC are a family of VECLIB dynamic memory allocators for FORTRAN 77 programs. NALLOC allocates arrays at run-time that will exist for the duration of a process, unless explicitly deallocated.

DALLOC deallocates a dynamic array previously established via NALLOC.

RALLOC changes the size of a nonvolatile dynamic array while preserving its contents.

Subprograms Included in This Chapter

Following are miscellaneous subprograms included with VECLIB.

Name	CPUTIME Measure CPU Time Used	
Purpose	CPUTIME returns the total amount of CPU time used so far by a process. Both user time and system time are included. The time unit is seconds and the resolution of the timer is 0.01 seconds. CPUTIME also makes it easy to time a segment of code in a program. In a multithreaded execution, CPUTIME returns the sum of the CPU time from all the threads of the process, not the individual CPU time of a thread.	
Usage	VECLIB: REAL*4 CPUTIME, time, tzero time = CPUTIME(tzero)	
Input	tzero	Starting time of a time interval you wish to measure. See Notes for details.
	time	Elapsed CPU time in seconds minus tzero.
Notes	To measure the total CPU time used by a program, use tzero = 0.0. To measure the amount of CPU time used by a segment of code in a program, use CPUTIME before and after the segment as follows: REAL*4 CPUTIME, time, tzero tzero = CPUTIME (0.0) <i>segment of code to be timed</i> time = CPUTIME(tzero)	

Name	DALLOC Deallocate Nonvolatile Dynamic Memory	
Purpose	DALLOC is a dynamic memory deallocator for Fortran programs. It uses the general-purpose memory allocation package to provide a facility that is not included in standard FORTRAN 77. DALLOC is a member of a family of VECLIB dynamic memory allocators. The other subprograms in the family are NALLOC and RALLOC. NALLOC allocates arrays at run-time that will exist for the duration of a process unless explicitly deallocated. RALLOC is used to change the size of a nonvolatile dynamic array while preserving its contents. DALLOC is used to deallocate a dynamic array previously established via NALLOC. NALLOC and RALLOC are documented elsewhere in this chapter.	
Usage	VECLIB: INTEGER*4 iptr, ier CALL DALLOC (iptr, ier) IF (ier .LT. 0) THEN <i>handle error</i> END IF VECLIB8: INTEGER*4 iptr INTEGER*8 ier CALL DALLOC (iptr, ier) IF (ier .LT. 0) THEN <i>handle error</i> END IF	
Input	iptr	The memory address of a nonvolatile array, previously allocated via NALLOC or reallocated by RALLOC.
Output	iptr	Invalid memory address if ier = 0 is returned. Do not use the output value of iptr as a memory address, or problems are certain to result.
	ier	Status response:
		ier = 0 Normal return. ier = -1 iptr is not a valid array address.

Notes DALLOC uses the general-purpose memory allocation package; refer to *malloc(3)* for details and error conditions.

Example Deallocate the arrays allocated in the “Example” in the documentation for subprogram RALLOC. It is assumed in this code segment that the variables IPTR, JPTR, MOLD and NOLD are the same quantities as in the RALLOC “Example.”

```
INTEGER*4 IPTR,JPTR,IER1,IER2,MOLD,NOLD
IF ( MOLD .NE. 0 .AND. NOLD .NE. 0 ) THEN
  CALL DALLOC (IPTR,IER1)
  CALL DALLOC (JPTR,IER2)
  IF ( IER1 .LT. 0 .OR. IER2 .LT. 0 ) THEN
    WRITE (6,*) 'DALLOC ERROR:',IER1,IER2
    STOP
  END IF
  MOLD = 0
  NOLD = 0
END IF
```

Name	MLIB_GETNUMTHREADS Determine Permitted Extent of Parallelism	
Purpose	This subprogram is used by all parallelized MLIB subprograms to determine the extent to which they may use shared-memory parallelism. If companion subroutine MLIB_SETNUMTHREADS has not been called, or if it was last called with a zero argument, MLIB_GETNUMTHREADS uses the environment variable MLIB_NUMBER_OF_THREADS to determine the maximum number of threads that are allowed to execute parallelized MLIB subprograms called by the program. If MLIB_NUMBER_OF_THREADS is not set, only one thread is allowed; i.e., MLIB subprograms are not run with parallelism. If MLIB_SETNMTHREADS has been called and the argument for its last call was nonzero, MLIB_GETNUMTHREADS ignores MLIB_NUMBER_OF_THREADS and bases its return value on the last MLIB_SETNUMTHREADS call.	
Usage	VECLIB: <pre> INTEGER*4 MLIB_GETNUMTHREADS, nthread nthread = MLIB_GETNUMTHREADS() </pre> VECLIB8: <pre> INTEGER*4 MLIB_GETNUMTHREADS, nthread nthread = MLIB_GETNUMTHREADS() </pre>	
Output	nthread	The maximum number of threads that can be used at this time. nthread = 1 if the machines has only one CPU, the operating system or version of MLIB does not support thread-based parallelism, the program is running on a subcomplex consisting of one CPU, the executable file is not marked for parallel execution, the default maximum number of threads is one, or the program currently is multi-threaded.

Name	MLIB_SETNUMTHREADS Control Extent of Parallelism	
Purpose	This subprogram may be used to control the extent to which parallelized MLIB subprograms may use shared-memory parallelism.	
Usage	VECLIB: INTEGER*4 nthread CALL MLIB_SETNUMTHREADS (nthread) VECLIB8: INTEGER*4 nthread CALL MLIB_SETNUMTHREADS (nthread)	
Input	nthread	If nthread > 0, it specifies the maximum number of parallel threads that can be spawned in subsequent calls to parallelized MLIB subprograms. If nthread = 0, MLIB_GETNUMTHREADS will, on its next call, restore the default maximum number of threads from the MLIB_NUMBER_OF_THREADS environment variable, if defined. If MLIB_NUMBER_OF_THREADS is not defined, the default maximum number of threads is set to one.
Notes	nthread becomes internal state for MLIB_GETNUMTHREADS. Only one value of nthread is saved within a UNIX process; subsequent calls to MLIB_SETNUMTHREADS overwrite the previous value.	

Name	NALLOC Allocate Nonvolatile Dynamic Memory
Purpose	NALLOC is a dynamic memory allocator for Fortran programs. It uses the general-purpose memory allocation package to provide a facility that is not included in standard FORTRAN 77. NALLOC provides arrays that exist for the duration of a process unless explicitly deallocated. When NALLOC is called, it obtains a block of memory of the specified size and returns the memory address of that space. If Fortran 77 had pointers, using the memory address would be easy, but as it is, there is no simple way to use a variable containing a memory address except to pass it to a lower-level subprogram using the nonstandard %VAL intrinsic. %VAL causes the Fortran compiler to pass the value of its argument to a subroutine rather than the address; this turns the pointer in the calling subprogram into an array in the lower-level subprogram. NALLOC is a member of a family of VECLIB dynamic memory allocators. The other subprograms in the family are DALLOC and RALLOC. DALLOC is used to deallocate a dynamic array previously established via NALLOC. RALLOC is used to reallocate a dynamic array, i.e., to change its size while preserving its contents. DALLOC and RALLOC are documented elsewhere in this chapter. NALLOC is included in VECLIB because extra arrays are occasionally required to optimize code or use a VECLIB subprogram. It is unnecessary when using NALLOC to build fixed-size temporary arrays into a program and take the risk that they will not always be big enough or to modify the program's storage allocation scheme to provide the extra space.
Usage	VECLIB: <pre>INTEGER*4 l, iptr, ier CALL NALLOC (l, iptr, ier) IF (ier .LT. 0) THEN <i>handle error</i> END IF CALL sub (... , %VAL(iptr), ...)</pre>

VECLIB8:

```

      INTEGER*4 iptr
      INTEGER*8 l, ier
      CALL NALLOC (l, iptr, ier)
      IF ( ier .LT. 0 ) THEN
         handle error
      END IF
      CALL sub (... , %VAL(iptr), ...)

```

Input	l	The length, in bytes, of the nonvolatile array to allocate, $l > 0$. l should be the product of the byte length for the data type and the range of elements in each dimension. Thus, for a REAL*4 array of dimension 5 by 6, use $l = 4 \times 5 \times 6 = 120$.								
Output	iptr	The memory address of the allocated array if ier = 0 is returned. Retain iptr as long as the array is needed; it is the only identification of the array and the only way to make use of it.								
	ier	Status response: <table><tr><td>ier = 0</td><td>Normal return.</td></tr><tr><td>ier = -1</td><td>Illegal array size; $l \leq 0$.</td></tr><tr><td>ier = -2</td><td>Too many nonvolatile arrays allocated.</td></tr><tr><td>ier = -3</td><td>Error response from function <code>_malloc</code>.</td></tr></table>	ier = 0	Normal return.	ier = -1	Illegal array size; $l \leq 0$.	ier = -2	Too many nonvolatile arrays allocated.	ier = -3	Error response from function <code>_malloc</code> .
ier = 0	Normal return.									
ier = -1	Illegal array size; $l \leq 0$.									
ier = -2	Too many nonvolatile arrays allocated.									
ier = -3	Error response from function <code>_malloc</code> .									
Notes	At most, 100 arrays may be allocated at a time via NALLOC. The number of allocated arrays is increased by one for each call to NALLOC that results in a normal return and is decreased by one for each call to DALLOC that results in a normal return. NALLOC first tries to allocate the array aligned on a longword boundary. If the array does not fit entirely into a single virtual memory page, it is realigned to a page boundary. NALLOC uses the general-purpose memory allocation package; refer to <i>malloc(3)</i> for details and error conditions. Allocations are cumulative. Thus, calling NALLOC in a loop may use up an inordinate amount of memory if the calling subprogram does not call subprogram DALLOC occasionally to deallocate the previously-allocated space.									

Example Allocate a nonvolatile m -by- n REAL*8 array and a nonvolatile INTEGER*4 array of length $n+1$ and pass them as arguments to subroutines SUB1 and SUB2. (Compare with "Example" in the description of RALLOC.)

```
INTEGER*4 M,N,IPTR,JPTR,IER1,IER2
SAVE      IPTR,JPTR
CALL NALLOC (8*M*N,IPTR,IER1)
CALL NALLOC (4*N+4,JPTR,IER2)
IF ( IER1 .LT. 0 .OR. IER2 .LT. 0 ) THEN
    WRITE (6,*) 'NALLOC ERROR:',M,N,IER1,IER2
    STOP
END IF
CALL SUB1 (...,%VAL(IPTR),...,%VAL(JPTR),...)
CALL SUB2 (...,%VAL(IPTR),...,%VAL(JPTR),...)
```

Name RALLOC
Reallocate Nonvolatile Dynamic Memory

Purpose RALLOC is a dynamic memory reallocator for Fortran programs. It uses the general-purpose memory allocation package to provide a facility that is not included in standard FORTRAN 77.

One of the arguments in the **CALL RALLOC** statement is the memory address of an array of nonvolatile memory previously allocated with NALLOC or previously reallocated with RALLOC. Another argument is the desired new size for the array. The size of the given array is changed to satisfy the new size request, if possible. Otherwise, RALLOC obtains a new block of memory of the specified size, copies the contents of the current array to the new one, and returns the memory address of the new array. The new memory address can be used in the same way as the original one, via the nonstandard %VAL intrinsic.

RALLOC is a member of a family of VECLIB dynamic memory allocators. The other subprograms in the family are NALLOC and DALLOC. NALLOC allocates arrays at run-time that exist for the duration of a process unless explicitly deallocated. DALLOC is used to deallocate a dynamic array previously established via NALLOC. RALLOC is used to change the size of a nonvolatile dynamic array while preserving its contents. NALLOC and DALLOC are documented elsewhere in this chapter.

Usage VECLIB:

```
INTEGER*4 l, iptr, ier
CALL RALLOC (l, iptr, ier)
IF ( ier .LT. 0 ) THEN
    handle error
END IF
CALL sub (... , %VAL(iptr), ...)
```

VECLIB8:

```
INTEGER*4 iptr
INTEGER*8 l, ier
CALL RALLOC (l, iptr, ier)
IF ( ier .LT. 0 ) THEN
    handle error
END IF
CALL sub (... , %VAL(iptr), ...)
```

Input	l	The new length, in bytes, of the nonvolatile array, $l > 0$. l should be the product of the byte length for the data type and the range of elements in each dimension. Thus for a REAL*4 array of dimension 5 by 6, use $l = 4 \times 5 \times 6 = 120$.
	iptr	The memory address of a nonvolatile array previously allocated via NALLOC or reallocated by RALLOC.
Output	iptr	The memory address of the allocated array if ier = 0 is returned. iptr may be different from or the same as its input value, depending on whether or not the array had to be moved. Retain the output value of iptr as long as the reallocated array is needed, since it is the only identification of the array and the only way to make use of it.
	ier	Status response: ier = 0 Normal return. ier = -1 Illegal array size; $l \leq 0$. ier = -2 iptr is not a valid array address. ier = -3 Error response from function <code>_realloc</code>.
Notes		RALLOC allocates the array aligned on a longword boundary. If the array does not fit entirely into a single virtual memory page, it is realigned to a page boundary.
		RALLOC uses the general-purpose memory allocation package; refer to <code>malloc(3)</code> for details and error conditions.
Example		Allocate a nonvolatile m -by- n REAL*8 array and a nonvolatile INTEGER*4 array of length $n+1$. Retain the arrays between executions of this code, and reallocate them when m or n have changed. Pass the arrays as arguments to subroutine SUB1 whenever they are allocated or reallocated and to SUB2 during every execution.


```

INTEGER*4 M,N,IPTR,JPTR,IER1,IER2,MOLD,NOLD
SAVE      IPTR,JPTR,MOLD,NOLD
DATA      MOLD,NOLD /0,0/ ! VALUES TO INDICATE FIRST EXECUTION
IF ( M .NE. MOLD .OR. N .NE. NOLD ) THEN
  IF ( MOLD .EQ. 0 .AND. NOLD .EQ. 0 ) THEN
    CALL NALLOC (8*M*N,IPTR,IER1)
    CALL NALLOC (4*N+4,JPTR,IER2)
    IF ( IER1 .LT. 0 .OR. IER2 .LT. 0 ) THEN
      WRITE (6,*) 'NALLOC ERROR:',M,N,IER1,IER2
      STOP
    END IF
  ELSE
    CALL RALLOC (8*M*N,IPTR,IER1)
    CALL RALLOC (4*N+4,JPTR,IER2)
    IF ( IER1 .LT. 0 .OR. IER2 .LT. 0 ) THEN
      WRITE (6,*) 'RALLOC ERROR:',IER1,IER2
      STOP
    END IF
  END IF
  MOLD = M
  NOLD = N
  CALL SUB1 (...,%VAL(IPTR),...,%VAL(JPTR),...)
END IF
CALL SUB2 (...,%VAL(IPTR),...,%VAL(JPTR),...)

```

Name RAN
VAX-Compatible Random Numbers

Purpose This subprogram produces uniform [0,1) pseudorandom numbers identical to those generated by the VAX random-number generator, RAN. The function RAN is identical in usage to its VAX counterpart. Function RAN returns one random number per call, while companion subroutine RANV, also documented in this chapter, returns an array of random numbers per call.

These generators are based on the linear congruential method introduced by D. H. Lehmer in 1949; see Knuth. Given a starting seed, S_0 , with $0 \leq S_0 < 2^{32}$, they obtain a sequence of seeds $\{S_n\}$ by setting

$$S_n = (69069S_{n-1}) \bmod 2^{32}, n > 0.$$

This is generator 25 in Table 1 on page 102 of the reference mentioned above. Uniformly distributed numbers X_n between 0 (inclusive) and 1 (exclusive) are produced by the scaling

$$X_n = 2^{-32}S_n.$$

The period of these generators is 2^{32} , i.e., they repeat the same sequence of pseudorandom numbers after about 4.3 billion numbers. Subprograms SRAN and DRAN, documented elsewhere in this chapter, are random number generators with a much longer period.

Usage VECLIB:
INTEGER*4 **iseed**
REAL*4 **RAN, x**
x = RAN(iseed)

VECLIB8:
INTEGER*8 **iseed**
REAL*8 **RAN, x**
x = RAN(iseed)

Input **iseed** An initial seed to start a pseudorandom sequence, or the **iseed** returned by the previous call to RAN to continue a sequence.

Output **iseed** The seed that produces the next pseudorandom number in the sequence replaces the input seed.
 x The next pseudorandom number in the sequence.

Notes

Starting a sequence twice with the same value of **iseed** will produce the same pseudorandom sequence.

You may have as many independent sequences going at a time as you desire by having a different **iseed** for each one.

The subprograms treat **iseed** as an unsigned 32-bit quantity. To write it out for later continuation of the same sequence, either use unformatted I/O statements or write it out with an octal, unsigned integer, or hexadecimal format descriptor of the form **O11**, **SU,I10**, or **Z8**, respectively.

Example

Fill an array **X** that is 10 elements long with a sequence of pseudorandom numbers using the scalar subprogram **RAN**.

```
INTEGER*4 ISEED,N
REAL*4    RAN,X(10)
ISEED = 1234                ! STARTING SEED
N = 10
DO I = 1, N
    X(I) = RAN(ISEED)
END DO
```

This fills array **X** with the same sequence of pseudorandom numbers and also returns the same final value of **ISEED**, as the “Example” for **RANV**.

Name RANV
VAX-Compatible Random Numbers

Purpose This subprogram produces uniform [0,1) pseudorandom numbers identical to those generated by the VAX random-number generator, RAN. Function RAN returns one random number per call, while companion subroutine RANV, also documented in this chapter, returns an array of random numbers per call.

These generators are based on the linear congruential method introduced by D. H. Lehmer in 1949; see Knuth. Given a starting seed, S_0 , with $0 \leq S_0 < 2^{32}$, they obtain a sequence of seeds $\{ S_n \}$ by setting

$$S_n = (69069S_{n-1} + 1) \bmod 2^{32}, n > 0.$$

This is generator 25 in Table 1 on page 102 of the reference mentioned above. Uniformly distributed numbers X_n between 0 (inclusive) and 1 (exclusive) are produced by the scaling

$$X_n = 2^{-32} S_n.$$

The period of these generators is 2^{32} , i.e., they repeat the same sequence of pseudorandom numbers after about 4.3 billion numbers. Subprograms SRANV and DRANV, documented elsewhere in this chapter, are vectorized random number generators with a much longer period.

Usage VECLIB:

```

      INTEGER*4      iseed, n
      REAL*4         x(n)
      CALL RANV(iseed, n, x)

```

VECLIB8:

```

      INTEGER*8      iseed, n
      REAL*8         x(n)
      CALL RANV(iseed, n, x)

```

Input

iseed	An initial seed to start a pseudorandom sequence, or the iseed returned by the previous call to RANV to continue a sequence.
n	The number of pseudorandom numbers to place in array x .

Output	iseed	The seed that produces the next pseudorandom number in the sequence replaces the input seed.
	x	The next n pseudorandom numbers in the sequence are stored in the first n elements of x .

Notes **CALL RANV (iseed, n, x)** produces the same **n** pseudorandom numbers as returned by **n** successive references to **RAN(iseed)**.

CALL RANV (iseed, n, x) produces the same value of **iseed** that would have resulted from the last of **n** successive references to **RAN(iseed)**.

Starting a sequence twice with the same value of **iseed** will produce the same pseudorandom sequence.

You may have as many independent sequences going at a time as you desire by having a different **iseed** for each one.

The subprograms treat **iseed** as an unsigned 32-bit quantity. To write it out for later continuation of the same sequence, either use unformatted I/O statements or write it out with an octal, unsigned integer, or hexadecimal format descriptor of the form **O11**, **SU,I10**, or **Z8**, respectively.

Example Fill an array **X** that is 10 elements long with a sequence of pseudorandom numbers using the vector subprogram **RANV**.

```

INTEGER*4 ISEED,N
REAL*4    X(10)
ISEED = 1234                      ! STARTING SEED
N = 10
CALL RANV (ISEED,N,X)

```

This fills array **X** with the same sequence of pseudorandom numbers and also returns the same final value of **ISEED**, as the "Example" for **RAN**.

Name SRAN/DRAN
Scalar Long Period Random Number Generator

Purpose These subprograms produce a sequence of uniformly-distributed [0,1) pseudorandom numbers with a period of 2^{48} . Functions SRAN and DRAN return one random number per call, while companion subprograms SRANV and DRANV, also documented in this chapter, return an array of random numbers per call.

These generators are based on the linear congruential method introduced by D. H. Lehmer in 1949; see Knuth. Given a starting seed, S_0 , with $0 \leq S_0 < 2^{48}$, they obtain a sequence of seeds $\{ S_n \}$ by setting

$$S_n = (31167285S_{n-1} + 1) \bmod 2^{48}, n > 0.$$

This is generator 29 in Table 1 on page 102 of the reference mentioned above. Uniformly distributed numbers X_n between 0 (inclusive) and 1 (exclusive) are produced by the scaling

$$X_n = 2^{-48} S_n.$$

The period of these generators is 2^{48} , i.e., they repeat the same sequence of pseudorandom numbers after about 281 trillion numbers.

Usage VECLIB:

```

      INTEGER*8      iseed
      REAL*4         SRAN, x
      x = SRAN(iseed)

      INTEGER*8      iseed
      REAL*4         DRAN, x
      x = DRAN(iseed)

```

Alternatively, in the absence of INTEGER*8, use:

```

      INTEGER*4      iseed(2)
      REAL*4         SRAN, x
      x = SRAN(iseed)

      INTEGER*4      iseed(2)
      REAL*8         DRAN, x
      x = DRAN(iseed)

```

VECLIB8:

```

      INTEGER*8      iseed
      REAL*8         SRAN, x
      x = SRAN(iseed)

```

Input	iseed	An initial seed to start a pseudorandom sequence, or the iseed returned by the previous call to the subprogram to continue a sequence.
Output	iseed	The seed that produces the next pseudorandom number in the sequence replaces the input seed.
	x	The next pseudorandom number in the sequence.
Notes	Starting a sequence twice with the same value of iseed will produce the same pseudorandom sequence. You may have as many independent sequences going at a time as you desire by having a different iseed for each one. The subprograms treat iseed as an unsigned 48-bit quantity. To write it out for later continuation of the same sequence, either use unformatted I/O statements or write it out with an octal, unsigned integer, or hexadecimal format descriptor of the form O16, SU, I15, or Z12, respectively. Alternatively, in the absence of INTEGER*8, use 2O12, SU, 2I/O or 2Z8, respectively.	

Example Fill an array X that is 10 elements long with a sequence of pseudorandom numbers using the scalar subprogram SRAN.

```

      INTEGER*4 N
      INTEGER*8 ISEED
      REAL*4     SRAN, X(10)
      ISEED = 1234                                ! STARTING SEED
      N = 10
      DO I = 1, N
         X(I) = SRAN(ISEED)
      END DO

```

This fills array X with the same sequence of pseudorandom numbers and also returns the same final value of ISEED, as the "Example" for SRANV.

Name SRANV/DRANV
Vector Long Period Random Number Generator

Purpose These subprograms produce a sequence of uniformly-distributed $[0,1)$ pseudorandom numbers with a period of 2^{48} . Subroutines SRANV and DRANV return an array of random numbers per call at less time per number than the functions SRAN and DRAN, which return one random number per call.

These generators are based on the linear congruential method introduced by D. H. Lehmer in 1949; see Knuth. Given a starting seed, S_0 , with $0 \leq S_0 < 2^{48}$, they obtain a sequence of seeds $\{S_n\}$ by setting

$$S_n = (31167285S_{n-1} + 1) \bmod 2^{48}, n > 0.$$

This is generator 29 in Table 1 on page 102 of the reference mentioned above. Uniformly distributed numbers X_n between 0 (inclusive) and 1 (exclusive) are produced by the scaling

$$X_n = 2^{-48} S_n.$$

The period of these generators is 2^{48} , i.e., they repeat the same sequence of pseudorandom numbers after about 281 trillion numbers.

Usage VECLIB:

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

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

Alternatively, in the absence of INTEGER*8, use:

```
INTEGER*4      iseed(2)
INTEGER*4      n, incx
REAL*4         x(lenx)
CALL SRANV(iseed, n, x, incx)
```

```
INTEGER*4      iseed(2)
INTEGER*4      n, incx
REAL*8         x(lenx)
CALL DRANV(iseed, n, x, incx)
```


VECLIB8:

```

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

```

Input	iseed	An initial seed to start a pseudorandom sequence, or the iseed returned by the previous call to the subprogram to continue a sequence.
	n	The number of pseudorandom numbers to place in array x .
	incx	Increment for the array x :
	incx ≥ 0	<i>x</i> is stored forward in array x ; i.e., <i>x_i</i> is stored in x ((<i>i</i> −1)×incx+1).
	incx < 0	<i>x</i> is stored backward in array x ; i.e., <i>x_i</i> is stored in x ((<i>i</i> −n)×incx+1).
		Use incx = 1 if the vector <i>x</i> is stored contiguously in array x , i.e., if <i>x_i</i> is stored in x (<i>i</i>). Refer to “BLAS Indexing Conventions” in the introduction to Chapter 2.
Output	iseed	The seed that produces the next pseudorandom number in the sequence replaces the input seed.
	x	Array of length lenx = (n−1)× incx +1 containing the <i>n</i> -vector <i>X</i> . If n ≤ 0, then x is not referenced. Otherwise, the next n pseudorandom numbers in the sequence replaces the input.
Notes	CALL SRANV (iseed, n, x, incx) produces the same n pseudorandom numbers as returned by n successive references to SRAN(iseed) .	
	CALL SRANV (iseed, n, x, incx) produces the same value of iseed that would have resulted from the last of n successive references to SRAN(iseed) .	
	Starting a sequence twice with the same value of iseed will produce the same pseudorandom sequence.	
	You may have as many independent sequences going at a time as you desire by having a different iseed for each one.	
The subprograms treat iseed as an unsigned 48-bit quantity. To write it out for later continuation of the same sequence, either use unformatted I/O statements or write it out with an octal, unsigned integer, or hexadecimal format descriptor		

of the form **O16**, **SU,I15**, or **Z12**, respectively. Alternatively, in the absence of **INTEGER*8**, use **2O12**, **SU**, **2I/O** or **2Z8**, respectively

Example Fill an array **X** that is 10 elements long with a sequence of pseudorandom numbers using the vector subprogram **SRANV**.

```
INTEGER*8 ISEED
INTEGER*4 N, INCX
REAL*4    X(10)
ISEED = 1234                ! STARTING SEED
N = 10
INCX = 1
CALL SRANV (ISEED,N,X,INCX)
```

This fills array **X** with the same sequence of pseudorandom numbers and also returns the same final value of **ISEED**, as the “Example” for **SRAN**.

Name SSORT/DSORT/ISORT
Sort Array

Purpose These subprograms use a Quicksort algorithm to sort elements of a vector into ascending or descending algebraic order. The vector may be stored in a one-dimensional array or in either a row or a column of a two-dimensional array.

Usage VECLIB:

```

      CHARACTER*(*) order
      INTEGER*4      n, incx
      REAL*4         x(lenx)
      CALL SSORT(order, n, x, incx)

      CHARACTER*(*) order
      INTEGER*4      n, incx
      REAL*8         x(lenx)
      CALL DSORT(order, n, x, incx)

      CHARACTER*(*) order
      INTEGER*4      n, incx, x(lenx)
      CALL ISORT(order, n, x, incx)

```

VECLIB8:

```

      CHARACTER*(*) order
      INTEGER*8      n, incx
      REAL*8         x(lenx)
      CALL SSORT(order, n, x, incx)

      CHARACTER*(*) order
      INTEGER*8      n, incx, x(lenx)
      CALL ISORT(order, n, x, incx)

```

Input	order	Sort order option:
		'A' or 'a' Sort the elements of x into ascending algebraic order.
		'D' or 'd' Sort the elements of x into descending algebraic order.
	n	Number of elements of vector x to be sorted. If $n \leq 1$, the subprograms do not reference x .
	x	Array of length $\text{lenx} = (n-1) \times \text{incx} + 1$ containing the data to be sorted.

incx Increment for the array **x**. **x** is stored forward in array **x** with increment $|\text{incx}|$; i.e., x_i is stored in $x((i-1) \times |\text{incx}| + 1)$.
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 Chapter 2.

Output **x** If **n** ≤ 0 or if **order** is not 'A', 'a', 'D', or 'd', then **x** is unchanged. Otherwise, the sorted result replaces the input.

Notes Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **order** argument as 'ascending' or 'descending'. Refer to "Example 2."

Example 1 Sort the elements of a REAL*4 vector **x** into ascending order, where **x** is a vector 100 elements long stored in a one-dimensional array **X** of dimension 200.

```
CHARACTER*(*) ORDER
INTEGER*4      N, INCX
REAL*4         X(200)
ORDER = 'A'
N = 100
INCX = 1
CALL SSORT (ORDER, N, X, INCX)
```

Example 2 Sort the elements of a REAL*8 vector **x** into descending order, where **x** is a vector 100 elements long stored in a one-dimensional array **X** of dimension 200.

```
CHARACTER*(*) ORDER
INTEGER*4      N, INCX
REAL*8         X(200)
CALL DSORT ('DESCENDING', 100, X, 1)
```

Name WALLTIME
Measure Wallclock Time Used

Purpose WALLTIME returns the current wallclock time. A base time is set during the first execution of WALLTIME, and the return value is the elapsed time from this base. The time unit is seconds and the resolution of the timer is microseconds. WALLTIME also makes it easy to time a segment of code in a program.

Usage VECLIB:

REAL*4 WALLTIME, time, tzero

Input **tzero** Starting time of a time interval you wish to measure.
See Notes for details.

Output **time** Elapsed CPU time in seconds minus **tzero**.

Notes To measure the wallclock time used by a segment of code in a program, use WALLTIME before and after the segment as follows:

REAL*4 WALLTIME, time, tzero

tzero = WALLTIME (0.0)

segment of code to be timed

time = WALLTIME(tzero)

Name XERVEC
VECLIB Error Handler

Purpose XERVEC is the error handler for many of the subprograms in the VECLIB library, as indicated in the "Notes" section in the subprogram descriptions. As supplied in VECLIB, XERVEC writes one of the following error messages onto the standard error file:

```
*****
* XERVEC: subprogram name called with invalid value of argument number iarg *
*****

*****
* XERVEC: error detected by subprogram name: text of error message *
*****

*****
* XERVEC: error iarg detected by subprogram name: text of error message *
*****
```

where *name* is the name of the subprogram in which the error was detected, *iarg* is the argument number of the offending argument, and *text of error message* is a character string. If the main program is in Fortran, a call traceback is also written onto the standard error file. XERVEC then terminates execution with a nonzero exit status.

You may supply a version of XERVEC that alters this action. All VECLIB subprograms that call XERVEC have a **RETURN** statement after the **CALL** statement, so your version could set a flag in a common block and **RETURN**. The flag could be tested in the program unit that calls the VECLIB subprogram.

Usage VECLIB:

```
CHARACTER*(*) name, messag
INTEGER*4      iarg
CALL XERVEC(name, iarg, messag)
```

VECLIB8:

```
CHARACTER*(*) name, messag
INTEGER*8      iarg
CALL XERVEC(name, iarg, messag)
```

Input **name** The name of the subprogram in which the error was detected.

iarg	If $iarg > 0$, the error message is printed in the first form given above and $iarg$ is the number of the argument that was found to be in error. If $iarg = 0$, the error message is printed in the second form shown and $iarg$ is not part of the error message. If $iarg < 0$, the error message is printed in the third form shown and $iarg$ is the error number.
messag	The text of the error message to be printed if $iarg \leq 0$. Not used as input if $iarg > 0$.

A Calling VECLIB from C

Introduction

Subprogram calls in Fortran use the same calling convention as used by other Hewlett-Packard language processors. This Hewlett-Packard standard permits VECLIB subprograms, which are written in Fortran or assembly language compatible with Fortran, to be called from programs written in the C language. This appendix describes the calling conventions necessary to call VECLIB subprograms from C programs. The same conventions apply to calling SCILIB and LAPACK subprograms from C programs.

General Interlanguage Programming Rules for VECLIB

The following are rules for calling a VECLIB subprogram from within a C language program.

- Spell VECLIB subprogram names in lower case and append an underscore character to them. For example, use `ddot_` for `DDOT`.
- All arguments to VECLIB subprograms are passed by address. This rule means that the only arguments you should pass to VECLIB subprograms are pointers. If `x` is type `int`, `long`, `long int`, `double`, or `float`, then `&x` is the address of `x`. Arrays and character strings already are passed as pointers and do not need the address-of operator, `&`. Because constants do not have addresses, use, for example, the declaration `int two = 2` and specify `&two` to pass the constant value 2.
- By default, the first element of a Fortran array is `X(1)`, but the first element of a C array is `X[0]`. As a consequence, you must adjust any input or output arguments or function values that are indices into arrays by decrementing the quantity by 1. For example, since `idamax` returns the one-based index of the element of an array that has maximum magnitude, you might code

```
i = idamax_(&n, x, &incx, &one) - 1;
```

- As explained in Chapter 2, VECLIB expects arrays to be stored in column-major order; however, C stores them in row-major order. Hence, a VECLIB subprogram will see the transpose of the C program's matrix. Three ways to handle this incompatibility are:
 - In your C program, store the transpose of the matrix you are using so it would appear to be in column-major order to the VECLIB subprogram.
 - Some VECLIB subprograms have a "transpose" option that uses the transpose of the matrix to do its calculations. In this case, store the matrix in normal order (for C) and use the transpose option to solve the intended problem. For example, to solve a system of linear equations $Ax = b$, factor the matrix you call A but which VECLIB sees as A^T with DGEFA, and use $job = 1$ in DGESL to solve for x .
 - Recast the computation so that it operates on C's row-major order. For example, the matrix-matrix multiplication $C = AB$ can be recast as $C^T = B^T A^T$. Thus, for example, VECLIB subroutine DGEMM can be used with $transa = transb = 'NonTransposed'$ by reversing the order of the matrices A and B and their transposition options, sizes, and leading dimensions in the CALL statement. See Figure A-2 for an example.
- Another consequence in the difference in storage order of arrays is that "leading dimension" arguments, such as lda in subroutine DGEMV, must be defined based on the last dimension of the C array declaration. For example, the C declaration


```
double a[2][4];
```

 is equivalent to the Fortran declaration


```
REAL*8 a(4,2)
```

 and corresponds to $lda = 4$.
- VECLIB subprograms that expect character arguments may be called from C with a string constant, a char array, or a pointer to a char variable for each character argument. In addition, an int must be passed at the end of the argument list for each character argument. These extra arguments are in the same order as the character arguments and indicate the length of the character string, not counting any "\0" terminator. For subroutine XERBLA, the quantity passed for argument `name` must be at least six characters long, and only six characters are significant. For subroutine XERSCI, the quantity passed for argument `name` must be at least eight characters long, and only eight characters are significant.
- Because C does not support complex arithmetic in a pre-defined way, storage layouts for Fortran types COMPLEX*8 and COMPLEX*16 usually are defined by typedef statements. Use the following typedef statements:

```
typedef struct {float re, im;} complex8_t;
/* COMPLEX*8 type */
typedef struct {double re, im;} complex16_t;
/* COMPLEX*16 type */
```

- Using the above typedef statements, Fortran and C declarations are related as shown in Table A-1.

Table A-1 Relationship Between Fortran and C Declarations

Fortran	C
LOGICAL*4 l	int l;
LOGICAL*8 l	long long int l;
INTEGER*4 i	int i;
INTEGER*8 i	long long int i;
REAL*4 s	float s;
REAL*8 d	double d;
COMPLEX*8 c	complex8_t c;
COMPLEX*16 z	complex16_t c;
CHARACTER*(n) t	char t[n];
PROGRAM main	main ()
SUBROUTINE sub (...)	void sub_ (...)
SUBROUTINE SUB (...)	void sub_ (...)
INTEGER*4 FUNCTION if (...)	int if_ (...)
INTEGER*8 FUNCTION if (...)	long long int if_ (...)
REAL*4 FUNCTION sf (...)	float sf_ (...)
REAL*8 FUNCTION df (...)	double df_ (...)
COMPLEX*8 FUNCTION cf (...)	complex8_t cf_ (...)
COMPLEX*16 FUNCTION zf (...)	complex16_t zf_ (...)

Header Files

C header files for VECLIB and VECLIB8 are installed in `/opt/mlib/include` as files *veclib.h* and *veclib8.h*. A C programmer normally would not use the VECLIB8 library, which is provided for use with the `+autodbl` and `+autodbl4` switches of the Fortran compiler since these switches are not available with the C compiler. However, if the C program calls Fortran subprograms other than from VECLIB, and if that Fortran must be compiled with one of the above compiler switches, the C program must also use the VECLIB8 library as it is

not recommended to try to use both the VECLIB and the VECLIB8 library in the same program. Header files *scilib.h*, *lapack.h*, and *lapack8.h* also are provided for use with their associated libraries. The following discussion applies to these header files as well as *veclib.h* and *veclib8.h*.

Use the `#include` directive to incorporate one of these header files into your program, for example,

```
#include <veclib.h>
```

Each header file includes ANSI C function prototypes and PCC declarations for all subprograms documented in the corresponding user's guide. Using the header files is optional but, if you do not use them, you must declare the return value types of the functions you use or C will treat them as integers.

If you need to use more than one of the five header files in a C program file, put them in the order that matches the order of the corresponding `-I` options on your link line. See "Interactions Between VECLIB, SCILIB, and LAPACK" in Chapter 1 for details about how to order the `-I` options.

The header files take care of many of the interlanguage programming rules listed in the previous section. Consider the following header file entry for VECLIB function DGEMM:

```
#ifndef DGEMM
#define DGEMM dgemm_
#define dgemm dgemm_
extern void dgemm_ (char* transa, char* transb, int* m, int* n,
                   int* k, double* alpha, double a[], int* lda,
                   double b[], int* ldb, double* beta, double c[],
                   int* ldc, int len_transa, int len_transb);
#endif
```

- The header file entry for each subprogram includes `#define` statements that will append an underscore to the subprogram name if you do not code one. If you code the subprogram name in upper case and omit the underscore, the name also will be translated to lower case. To use the header file but omit some of the function prototypes, `#define` the upper case form of the name before the `#include` of the header file.
- The ANSI C function prototypes in the header files enforce call by address by explicitly declaring every argument that is not a character string or an array to be passed by address.
- The ANSI C function prototypes also enforce the additional `int` arguments at the end of the argument list corresponding to the character arguments. Note the two `char*` arguments at the beginning of the argument list in the function prototype for DGEMM and the two `int` arguments at the end, which must tell the lengths of the character strings *transa* and *transb*.
- The `typedef` statements that were given to define the two data types, `complex8_t` and `complex16_t`, are included in the header file and used

throughout wherever `COMPLEX*8` or `COMPLEX*16` variables or arrays are required.

- If you are compiling with the `cc(1)` compiler, the ANSI C prototypes are replaced by Kernighan-and-Ritchie-style declarations. For example, the prototype above is replaced with

```
extern void dgemm_ ();
```

Examples

Calling Subroutines

This section gives an example of calling VECLIB subprograms to solve a system of linear equations. This example first presents a Fortran code segment, then a corresponding C code segment. The generation of the matrix and right-hand-side values is omitted from both examples.

The problem is to solve a system of linear equations $Ax = b$, where A is a 6-by-6 matrix stored in a `double` (equivalent to type `REAL*8` in Fortran) array whose dimensions are 10 by 10, and b is a vector 6 elements long stored in a `double` array of dimension 10.

Figure A-1 Calling VECLIB Subroutines from Fortran and C

Fortran:

```

PROGRAM MAIN
INTEGER*4 LDA, N, IER, JOB
PARAMETER ( LDA = 10 )
INTEGER*4 IPVT(LDA)
REAL*8      A(LDA,LDA), B(LDA)
N = 6
JOB = 0 ! TO SOLVE COLUMN-MAJOR-STORED SYSTEM
CALL DGEFA (A, LDA, N, IPVT, IER)
IF ( IER .EQ. 0 ) THEN
    CALL DGESL (A, LDA, N, IPVT, B, JOB)
ELSE
    WRITE (6,*) "singular matrix"
END IF
END

```

C:

```

#include <stdio.h>
#include <veclib.h>
#define LDA 10
main ()
{
    int lda = LDA;
    int ier;
    int n = 6;
    int job = 1; /* to solve transpose of
                  row-major-stored system */
    int ipvt[LDA];
    double a[LDA][LDA];
    double b[LDA];

    dgefa_ (a, &lda, &n, ipvt, &ier);
    if ( ier == 0 )
        dgesl_ (a, &lda, &n, ipvt, b, &job);
    else
        printf ("singular matrix\n");
}

```

Calling VECLIB Subroutines via an Interface Function

You may find it easier to develop C interfaces to encapsulate the interlanguage programming rules for the VECLIB subprograms you use. Figure A-2 shows how an interface can be written for DGEMM.

The first `#define` statement omits the prototype definition for DGEMM from the header file. Function `dgemm` is defined so that its C usage (see the `dgemm` call in `main`) is identical to the Fortran usage described in Chapter 3. The interface follows the suggestion to recast the computation for C's row-major order by reversing the order of the matrices and their transposition options, sizes, and

leading dimensions. It also converts the more convenient call by value to call by address and appends the character argument lengths.

Figure A-2 Matrix Multiplication via an Interface Function

```
#define DGEMM 1      /* skip dgemm prototype in <veclib.h> */

#include <stdio.h>
#include <veclib.h>

#define LDA 4
#define LDB 2
#define LDC 2

void dgemm (char* transa, char* transb, int m, int n, int k,
            double alpha, double* a, int lda, double* b,
            int ldb, double beta, double* c, int ldc)
{
    extern void dgemm_ ();

    dgemm_ (transb,transa,&n,&m,&k,&alpha,b,
            &ldb,a,&lda,&beta,c,&ldc,1,1);
}

main()
{
    static double a[3][LDA] = { { 1., 0., -1., 0. },
                                { 1., 2., 3., 4. },
                                { 4., 3., -2., -1. } };

    static double b[4][LDB] = { { 1., 2. },
                                { 2., 1. },
                                { 4., -3. },
                                { 3., 4. } };

    double c[3][LDC]; /* = { { -3., 5. }
                           { 29., 11. }
                           { -1., 13. } } */

    int i, j, m = 3, n = 2, k = 4;

    dgemm ("n","n",m,n,k,1.0,a,LDA,b,LDB,0.0,c,LDC);

    for(i=0; i<m; i++) {
        for(j=0; j<n; j++) printf("%8g ",c[i][j]);
        printf("\n");
    }
}
```

Calling Functions

The method of calling a VECLIB function subprogram depends on its data type. The following examples show how to call some Fortran function subprograms from C. The function prototypes in *veclib.h* cast the function return values to the proper data types.

REAL Functions

The program in Figure A-3 computes the dot product of two double (Fortran REAL*8) vectors of length 2 using the VECLIB subprogram DDOT.

Figure A-3 **Calling a Real-Valued Function from C**

```
#include <stdio.h>
#include <veclib.h>

main ()
{
    int one = 1;
    int two = 2;
    double s;
    static double x[2] = { 1.0, -2.0};
    static double y[2] = {-3.0, -2.0};

    s = ddot_ (&two, x, &one, y, &one);
    printf ("ddot = %f\n", s);
}
```

COMPLEX Functions

Figure A-4 illustrates how to call `complex16_t` (Fortran COMPLEX*16) VECLIB function subprogram ZDOTC.

Figure A-4 **Calling a Complex-Valued Function from C**

```
#include <stdio.h>
#include <veclib.h>

main ()
{
    int one = 1;
    int two = 2;
    complex16_t s;
    static complex16_t x[2] = {{-1.0, 2.0},{ 2.0,-3.0} };
    static complex16_t y[2] = {{ 3.0,-1.0},{ 2.0, 1.0} };

    s = zdotc_ (&two, x, &one, y, &one);
    printf ("zdotc = (%f,%f)\n", s.re, s.im);
}
```

Linking VECLIB Subprograms into a C Program

You should heed the following guidelines when you use `cc` to link your compiled program with the VECLIB library.

- Most VECLIB subprograms may be linked into a C program by including `-lveclib` on the link command line as follows:

```
cc -o test test.c -lveclib
```

Similarly, VECLIB8, SCILIB, LAPACK, and LAPACK8 subprograms are linked by including `-lveclib8`, `-lscilib`, `-llapack`, or `-llapack8` on the link command line.

- If unresolved symbols are encountered while linking, they may be resolved by including the `cl` library on the command line. If `etime_` is an unresolved symbol, the `U77` library must also be included.

Finally, you may need to include the `pthread`s, `cps`, and `ail` libraries as discussed in the “Parallel Processing” section of Chapter 1.

Error Handling

VECLIB contains two standard error handlers, subroutines `XERBLA` and `XERVEC`. These subroutines are documented in this guide. In addition, the documentation for every VECLIB subprogram indicates if it calls an error handler and, if so, under what conditions.

The error handlers in `SCILIB` and `LAPACK` work the same way.

The error handlers are designed to be compatible with both the Fortran and the C runtime systems. If either error handler is called within a Fortran program, it writes an error message and possibly a call traceback into the standard error file and terminates execution with a nonzero exit status. When called within a C program, the error handler writes the error message into the standard error file and raises signal `SIGIOT`. If the program has not provided a signal handler for `SIGIOT`, a core image is produced and execution terminates with a nonzero exit status. This is illustrated in Figure A-5.

Figure A-5 Default VECLIB Error Handling in a C Program

```

% cat vsig1.c
#include <veclib.h>

main ()
{
    int n = 1;
    float alpha = 0.0;
    float x[10];
    int incx = 0;

    sflr1c_ (&n, &alpha, x, &incx);
}% cc vsig1.c -lveclib
% a.out
*****
* XERVEC: subprogram SFLR1C called with invalid value of argument number 4 *
*****
IOT trap (core dumped)
% echo $status
134

```

The fourth argument, *incx*, in the call to VECLIB subprogram SFLR1C must be nonzero. When SFLR1C detects that it is zero, it calls the error handler XERVEC, which writes the error message and raises signal SIGIOT. Since no signal handler has been enabled for this signal, the system writes a core image file and terminates the process.

NOTE

If the user has a signal handler that is enabled by the SIGIOT signal, that signal handler will be executed if default error handling is in effect.

As a C programmer, you can alter this default behavior or change the signal used to any other valid signal (as given in *signal(3c)*), as shown in the next section.

Changing the Error Handler Signal

VECLIB includes a C-callable function, *initMLIB*, that may be used to change the signal used by the standard error handlers to something other than SIGIOT. Usage is:

```

#include <signal.h>

int initMLIB(sig)
int sig;

```

Sig is 0 or one of the signal numbers or names as given in */usr/include/signal.h*. The specified signal is used instead of the default signal, SIGIOT, if one of the error handlers is called. If *sig* is 0, SIGIOT is used. In addition, a simple signal handler is installed that simply executes *exit(1)* to terminate execution with a nonzero exit status. This is the C equivalent to executing a Fortran STOP statement.

Figure A-6 illustrates the use of `initMLIB`, whose function prototype is obtained from `veclib.h`.

Figure A-6 **Changing the VECLIB Error Handling Signal**

```
% cat vsig2.c
#include <signal.h>
#include <veclib.h>

main ()
{
    int n = 1;
    float alpha = 0.0;
    float x[10];
    int incx = 0;

    initMLIB (SIGUSR1);
    sflr1c_ (&n, &alpha, x, &incx);
    printf("Hello world\n");
}
% cc vsig2.c -lveclib
% a.out
*****
* XERVEC: subprogram SFLR1C called with invalid value of argument number 4 *
*****
% echo $status
1
```

In Figure A-6, the call to `initMLIB` changes VECLIB's error handling signal to `SIGUSR1` and enables VECLIB's simple signal handler for this signal. If 0 or `SIGIOT` had been used in place of `SIGUSR1`, the only differences in behavior from the program in Figure A-5 would be the core image and the exit status.

NOTE

The initialization functions for SCILIB and LAPACK also are called `initMLIB`.

Any previously established signal handler for signal `sig` will be replaced by VECLIB's default signal handler.

Supplying Your Own Signal Handler

After calling `initMLIB` to specify which signal VECLIB's standard error handlers are to use, you can set up your own signal handler for that signal via the C function `signal(3c)`. This is demonstrated by the program in Figure A-7.

Figure A-7 Changing the Error Handling Signal Handler

```
% cat vsig3.c
#include <signal.h>
#include <veclib.h>

static void veclib_sig (sig, code, scp)
    int sig, code;
    struct sigcontext *scp;
{
    printf("SIGUSR2 has occurred... exiting
    exit(123);
}

main ()
{
    int n = 1;
    float alpha = 0.0;
    float x[10];
    int incx = 0;
    int oldsig;

    initMLIB (SIGUSR2);
    oldsig = (int)signal (SIGUSR2, veclib_sig);
    sflr1c_ (&n, &alpha, x, &incx);
}% cc vsig3.c -lveclib
% a.out
*****
* XERVEC: subprogram SFLR1C called with invalid value of argument number 4 *
*****
SIGUSR2 has occurred... exiting.
% echo $status
123
```

In Figure A-7, the call to `initMLIB` changes VECLIB's error handling signal to SIGUSR2 and enables VECLIB's simple signal handler for this signal. Subsequently, the signal handler `veclib_sig` is enabled by the call to `signal`.

When VECLIB subprogram SFLR1C detects the error in arguments, it calls error handler XERVEC, which writes the error message and raises signal SIGUSR2. Finally, the system executes `veclib_sig`.

B Subprograms not in this Guide

LINPACK

Table B-1 lists the LINPACK subprograms that are included in VECLIB but are not documented in this guide. You may find information for these subprograms in the *LINPACK Users' Guide*.

Table B-1 LINPACK Subprograms not in this Guide

Name	Function
SCHDC	Cholesky Decomposition of a Symmetric Matrix
DCHDC	Cholesky Decomposition of a Symmetric Matrix
CCHDC	Cholesky Decomposition of a Hermitian Matrix
ZCHDC	Cholesky Decomposition of a Hermitian Matrix
SCHDD	Recompute the Cholesky Decomposition of a DOWNDATED Symmetric Matrix
DCHDD	Recompute the Cholesky Decomposition of a DOWNDATED Symmetric Matrix
CCHDD	Recompute the Cholesky Decomposition of a DOWNDATED Hermitian Matrix
ZCHDD	Recompute the Cholesky Decomposition of a DOWNDATED Hermitian Matrix
SCHEX	Recompute the Cholesky Decomposition of a Permuted Symmetric Matrix
DCHEX	Recompute the Cholesky Decomposition of a Permuted Symmetric Matrix
CCHEX	Recompute the Cholesky Decomposition of a Permuted Hermitian Matrix
ZCHEX	Recompute the Cholesky Decomposition of a Permuted Hermitian Matrix
SCHUD	Recompute the Cholesky Decomposition of a Updated Symmetric Matrix
DCHUD	Recompute the Cholesky Decomposition of a Updated Symmetric Matrix
CCHUD	Recompute the Cholesky Decomposition of a Updated Hermitian Matrix
ZCHUD	Recompute the Cholesky Decomposition of a Updated Hermitian Matrix
CHICO	Factor a Hermitian Indefinite Matrix and Estimate its Condition Number
ZHICO	Factor a Hermitian Indefinite Matrix and Estimate its Condition Number
CHIDI	Determinant, Inverse, and Inertia of a Hermitian Indefinite Matrix
ZHIDI	Determinant, Inverse, and Inertia of a Hermitian Indefinite Matrix
CHIFA	Factor a Hermitian Indefinite Matrix
ZHIFA	Factor a Hermitian Indefinite Matrix
CHISL	Solve Linear Equations with a Hermitian Indefinite Matrix
ZHISL	Solve Linear Equations with a Hermitian Indefinite Matrix
CHPCO	Factor a Hermitian Indefinite Packed Matrix and Estimate its Condition Number

Name	Function
ZHPCO	Factor a Hermitian Indefinite Packed Matrix and Estimate its Condition Number
CHPDI	Determinant, Inverse, and Inertia of a Hermitian Indefinite Packed Matrix
ZHPDI	Determinant, Inverse, and Inertia of a Hermitian Indefinite Packed Matrix
CHPFA	Factor a Hermitian Indefinite Packed Matrix
ZHPFA	Factor a Hermitian Indefinite Packed Matrix
CHPSL	Solve Linear Equations with a Hermitian Indefinite Packed Matrix
ZHPSL	Solve Linear Equations with a Hermitian Indefinite Packed Matrix
SPPCO	Factor a Positive Definite Packed Matrix and Estimate its Condition Number
DPPCO	Factor a Positive Definite Packed Matrix and Estimate its Condition Number
CPPCO	Factor a Positive Definite Packed Matrix and Estimate its Condition Number
ZPPCO	Factor a Positive Definite Packed Matrix and Estimate its Condition Number
SPPDI	Determinant and Inverse of a Positive Definite Packed Matrix
DPPDI	Determinant and Inverse of a Positive Definite Packed Matrix
CPPDI	Determinant and Inverse of a Positive Definite Packed Matrix
ZPPDI	Determinant and Inverse of a Positive Definite Packed Matrix
SPPFA	Factor a Positive Definite Packed Matrix
DPPFA	Factor a Positive Definite Packed Matrix
CPPFA	Factor a Positive Definite Packed Matrix
ZPPFA	Factor a Positive Definite Packed Matrix
SPPSL	Solve Linear Equations with a Positive Definite Packed Matrix
DPPSL	Solve Linear Equations with a Positive Definite Packed Matrix
CPPSL	Solve Linear Equations with a Positive Definite Packed Matrix
ZPPSL	Solve Linear Equations with a Positive Definite Packed Matrix
SQRDC	QR Decomposition of a General Rectangular Matrix
DQRDC	QR Decomposition of a General Rectangular Matrix
CQRDC	QR Decomposition of a General Rectangular Matrix
ZQRDC	QR Decomposition of a General Rectangular Matrix
SQRSL	Solve Linear Equations using the QR Decomposition
DQRSL	Solve Linear Equations using the QR Decomposition
CQRSL	Solve Linear Equations using the QR Decomposition
ZQRSL	Solve Linear Equations using the QR Decomposition
SSICO	Factor a Symmetric Indefinite Matrix and Estimate its Condition Number
DSICO	Factor a Symmetric Indefinite Matrix and Estimate its Condition Number
CSICO	Factor a Symmetric Indefinite Matrix and Estimate its Condition Number
ZSICO	Factor a Symmetric Indefinite Matrix and Estimate its Condition Number
SSIDI	Determinant, Inverse, and Inertia of a Symmetric Indefinite Matrix
DSIDI	Determinant, Inverse, and Inertia of a Symmetric Indefinite Matrix

Name	Function
CSIDI	Determinant, Inverse, and Inertia of a Symmetric Indefinite Matrix
ZSIDE	Determinant, Inverse, and Inertia of a Symmetric Indefinite Matrix
SSIFA	Factor a Symmetric Indefinite Matrix
DSIFA	Factor a Symmetric Indefinite Matrix
CSIFA	Factor a Symmetric Indefinite Matrix
ZSIFA	Factor a Symmetric Indefinite Matrix
SSISL	Solve Linear Equations with a Symmetric Indefinite Matrix
DSISL	Solve Linear Equations with a Symmetric Indefinite Matrix
CSISL	Solve Linear Equations with a Symmetric Indefinite Matrix
ZSISL	Solve Linear Equations with a Symmetric Indefinite Matrix
SSPCO	Factor a Symmetric Indefinite Packed Matrix and Estimate its Condition Number
DSPCO	Factor a Symmetric Indefinite Packed Matrix and Estimate its Condition Number
CSPCO	Factor a Symmetric Indefinite Packed Matrix and Estimate its Condition Number
ZSPCO	Factor a Symmetric Indefinite Packed Matrix and Estimate its Condition Number
SSPDI	Determinant, Inverse, and Inertia of a Symmetric Indefinite Packed Matrix
DSPDI	Determinant, Inverse, and Inertia of a Symmetric Indefinite Packed Matrix
CSPDI	Determinant, Inverse, and Inertia of a Symmetric Indefinite Packed Matrix
ZSPDI	Determinant, Inverse, and Inertia of a Symmetric Indefinite Packed Matrix
SSPFA	Factor a Symmetric Indefinite Packed Matrix
DSPFA	Factor a Symmetric Indefinite Packed Matrix
CSPFA	Factor a Symmetric Indefinite Packed Matrix
ZSPFA	Factor a Symmetric Indefinite Packed Matrix
SSPSL	Solve Linear Equations with a Symmetric Indefinite Packed Matrix
DSPSL	Solve Linear Equations with a Symmetric Indefinite Packed Matrix
CSPSL	Solve Linear Equations with a Symmetric Indefinite Packed Matrix
ZSPSL	Solve Linear Equations with a Symmetric Indefinite Packed Matrix
SSVDC	Singular Value Decomposition of a General Rectangular Matrix
DSVDC	Singular Value Decomposition of a General Rectangular Matrix
CSVDC	Singular Value Decomposition of a General Rectangular Matrix
ZSVDC	Singular Value Decomposition of a General Rectangular Matrix
STRCO	Estimate the Condition Number of a Triangular Matrix
DTRCO	Estimate the Condition Number of a Triangular Matrix
CTRCO	Estimate the Condition Number of a Triangular Matrix
ZTRCO	Estimate the Condition Number of a Triangular Matrix
STRDI	Determinant and Inverse of a Triangular Matrix
DTRDI	Determinant and Inverse of a Triangular Matrix
CTRDI	Determinant and Inverse of a Triangular Matrix

LINPACK

Name	Function
ZTRDI	Determinant and Inverse of a Triangular Matrix
STRSL	Solve Linear Equations with a Triangular Matrix
DTRSL	Solve Linear Equations with a Triangular Matrix
CTRSL	Solve Linear Equations with a Triangular Matrix
ZTRSL	Solve Linear Equations with a Triangular Matrix

EISPACK

Table B-2 lists the EISPACK subprograms that are included in VECLIB but are not documented in this guide. These are documented in the following references:

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.

You may also find information for these subprograms in the *EISPACK Guide* and the *EISPACK Guide Extension*.

Table B-2 EISPACK Subprograms not in this Guide

Name	Upper
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

Name	Upper
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
ORTBAK	Back Transform Eigenvectors following ORTHES
ORTHES	Reduce a Real General Matrix to Real Upper Hessenberg Form
ORTTRAN	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

Name	Upper
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

EISPACK

Index

A

accessing VECLIB 4
allocate dynamic own array 604, 607
arithmetic format 10
ASAP, automatic self allocating processors 7
automatic reordering 474
automatic self allocating processors (ASAP) 7

B

backward storage 20
band matrix 129, 150, 193, 198, 237, 242, 246, 251,
268, 271, 274, 278, 281, 285, 301
Basic Linear Algebra Subprograms 3, 17
bibliography xxiii
BLAS 3, 17, 229
BLAS indexing conventions 19
BLAS, Extended 3, 125, 231, 305
BLAS, Level 1 3
BLAS, Level 2 3, 125, 231, 305
BLAS, Level 3 3, 125, 231, 305
BLAS, Sparse 3, 17

C

C header files 627
C language 625
C1DFFT 501
C2DFFT 507
C3DFFT 511
CAXPY 47
CAXPYC 47
CAXPYI 50
CCHDC 235, 306, 637
CCHDD 235, 306, 637
CCHEX 235, 306, 637
CCHUD 235, 306, 637
CCOPY 62
CCOPYC 62
CDOTC 65
CDOTCI 69
CDOTU 65
CDOTUI 69
-cfc compiler option 3
CFFTS 515
CGBCO 237
CGBDI 242
CGBFA 246
CGBMV 129
CGBSL 251

CGECO 254
CGEDI 257
CGEFA 261
CGEMM 134
CGEMMS 6, 138
CGEMV 143
CGERC 147
CGERU 147
CGESL 264
CGTHR 74
CGTHRZ 76
CGTSL 268
CGTSLV 271
CHBMV 150
check accuracy of eigenvalue/eigenvector results
438
CHEMM 169
CHEMV 173
CHER 177
CHER2 180
CHER2K 184
CHERK 189
CHICO 235, 306, 637
CHIDI 235, 306, 637
CHIFA 235, 306, 637
CHISL 235, 306, 637
Cholesky factorization 274, 281, 288, 295
CHPCO 235, 306, 637
CHPDI 235, 306, 637
CHPFA 235, 306, 637
CHPMV 155
CHPR 160
CHPR2 164
CHPSL 235, 306, 637
clear vector 123
clip vector 53, 56, 59
CLSTEQ 79
CLSTNE 79
condition number 234, 237, 254, 274, 288, 357
control extent of parallelism 603
convolution 555, 559
correlation 555, 559
correlation and convolution subprograms 553
count vector elements 28
CPBCO 274
CPBDI 278
CPBFA 281
CPBSL 285
CPOCO 288
CPODI 291
CPOFA 295
CPOSL 298

CPPCO 235, 306, 637
 CPTSL 301
 cputime 599
 CRC1FT 523
 CRC2FT 529
 CRC3FT 535
 CRCFTS 542
 CROT 93
 CROTG 97
 CRSCL 107
 CSCAL 109
 CSCALC 109
 CSCTR 112
 CSPFA 235, 306, 637
 CSPSL 235, 306, 637
 CSROT 93
 CSRSCL 107
 CSSCAL 109
 CSUM 114
 CSVDC 235, 306, 637
 CSWAP 116
 CSYMM 169
 CSYR2K 184
 CSYRK 189
 CTBMV 193
 CTBSV 198
 CTPMV 204
 CTPSV 208
 CTRCO 235, 306, 637
 CTRDI 235, 306, 637
 CTRMM 213
 CTRMV 217
 CTRSL 235, 306, 637
 CTRSM 221
 CTRSV 225
 CWDOTC 119
 CWDOTU 119
 CXpa 9
 CZERO 123

D

D1DFFT 504
 D2DFFT 509
 D3DFFT 513
 DALLOC 600
 DAMAX 38
 DAMIN 41
 DASUM 44
 DAXPY 47
 DAXPYI 50
 DCHDC 235, 306, 637
 DCHDD 235, 306, 637
 DCHEX 235, 306, 637
 DCHUD 235, 306, 637
 DCLIP 53
 DCLIPL 56

DCLIPR 59
 DCONV 555, 559
 DCOPY 62
 DDOT 65
 DDOTI 69
 deallocate dynamic own array 600
 deallocate working storage 363, 440, 493
 determinant 235, 242, 257, 278, 291
 determine permitted extend of parallelism 602
 DFFTS 519
 DFLR1C 564
 DFLR1M 566
 DFLR1P 566
 DFLR2C 569
 DFLR2M 572
 DFLR2P 572
 DFLRLM 576
 DFLRLP 576
 DFRAC 72
 DFT 500
 DGBCO 237
 DGBDI 242
 DGBFA 246
 DGBMV 129
 DGBSL 251
 DGECO 254
 DGEDI 257
 DGEFA 261
 DGEMM 134
 DGEMMS 138
 DGEMV 143
 DGER 147
 DGESL 264
 DGTHR 74
 DGTHRZ 76
 DGTSL 268
 DGTSLV 271
 discrete Fourier transform 500
 DLSTEQ 79
 DLSTGE 79
 DLSTGT 79
 DLSTLE 79
 DLSTLT 79
 DLSTNE 79
 DMAX 83
 DMIN 85
 DNRM2 87
 DNRSQ 89
 documentation, online 15
 documentation, ordering xxv
 dot product 65, 69
 dot product, weighted 119
 DPBCO 274
 DPBDI 278
 DPBFA 281
 DPBSL 285
 DPOCO 288

DPODI 291
 DPOFA 295
 DPOSL 298
 DPPCO 235, 306, 637
 DPPROD 580
 DPSUM 583
 DPTSL 301
 DRAMP 91
 DRAN 614
 DRANV 616
 DRC1FT 526
 DRC2FT 532
 DRC3FT 538
 DRCFTS 547
 DROT 93
 DROTG 97
 DROTI 99
 DROTM 101
 DROTMG 104
 DRSCS 107
 DSBMV 150
 DSCAL 109
 DSCTR 112
 DSEVCK 438
 DSEVDA 440
 DSEVE1 389
 DSEVES 422
 DSEVEX 427
 DSEVI1 399
 DSEVIC 401
 DSEVIE 403
 DSEVIF 408
 DSEVIM 405
 DSEVIN 397
 DSEVOC 441
 DSEVOR 409
 DSEVPS 442
 DSEVRC 434
 DSEVRL 436
 DSEVRS 443
 DSEVSV 444
 DSEVV1 410
 DSEVVC 412
 DSEVVD 414
 DSEVVE 416
 DSEVVM 419
 DSKYDA 493
 DSKYDF 489
 DSKYDS 491
 DSKYFA 486
 DSKYFS 457
 DSKYFX 460
 DSKYI1 464
 DSKYIC 465
 DSKYIE 467
 DSKYIF 473
 DSKYIM 469

DSKYIN 463
 DSKYIS 471
 DSKYOC 494
 DSKYOR 474
 DSKYOU 475
 DSKYPS 495
 DSKYRS 496
 DSKYSL 487
 DSKYSR 497
 DSKYSV 498
 DSKYV1 476
 DSKYVC 478
 DSKYVE 480
 DSKYVM 482
 DSKYVS 484
 DSLECO 357
 DSLEDA 363
 DSLEFA 359
 DSLEFF 364
 DSLEFS 333
 DSLEI1 339
 DSLEIC 340
 DSLEIE 342
 DSLEIF 346
 DSLEIM 344
 DSLEIN 338
 DSLEOC 366
 DSLEOP 367
 DSLEOR 347
 DSLEPS 368
 DSLERD 369
 DSLERS 371
 DSLESL 361
 DSLESR 372
 DSLESV 373
 DSLEV1 349
 DSLEVC 350
 DSLEVE 352
 DSLEVM 354
 DSLR2 586
 DSLR3 589
 DSLRL 592
 DSORT 619
 DSPFA 235, 306, 637
 DSPMV 155
 DSPR 160
 DSPR2 164
 DSPSL 235, 306, 637
 DSUM 114
 DSVDC 235, 306, 637
 DSWAP 116
 DSYMM 169
 DSYMV 173
 DSYR 177
 DSYR2 180
 DSYR2K 184
 DSYRK 189

DTBMV 193
 DTBSV 198
 DTPMV 204
 DTPSV 208
 DTRCO 235, 306, 637
 DTRDI 235, 306, 637
 DTRMM 213
 DTRMV 217
 DTRSL 235, 306, 637
 DTRSM 221
 DTRSV 225
 DWDOT 119
 dynamic storage allocation 600, 604, 607
 dynamic storage deallocation 600
 DZAMAX 38
 DZAMIN 41
 DZASUM 44
 DZERO 123
 DZNRM2 87
 DZNRSQ 89

E

eigenvalues 305, 307, 310, 313, 316, 318
 eigenvalues, sparse 375
 eigenvectors 305, 307, 310, 318
 eigenvectors, sparse 375
 EISPACK 3, 305
 EISPACK Guide 2
 EISPACK Guide Extension 2
 end of matrix structure input 346, 408, 473
 envelope data structure 445, 447
 error handler 229
 error handling, VECLIB 14
 error reporting 14
 Euclidean norm 87, 89
 Extended BLAS 3, 125, 231, 305

F

factorization, numeric 357, 359, 486, 489
 factorization, sparse 357, 359, 486, 489
 factorization, symbolic 347
 fast Fourier transforms 499
 FFT, one-dimensional 501, 504, 523, 526
 FFT, one-dimensional, simultaneous 515, 519, 542, 547
 FFT, real-to-complex 523, 526, 529, 532, 535, 538, 542, 547
 FFT, three-dimensional 511, 513, 535, 538
 FFT, two-dimensional 507, 509, 529, 532
 filtering 555, 559
 find 35, 79
 first order linear recurrence 564, 566, 569, 572, 576
 floating-point format 10
 Fortran array argument association 19
 Fortran storage of arrays 18

forward storage 19
 fractional part 72
 further reference xxiii

G

gather 74, 76
 Givens rotation 93, 97, 99
 Givens rotation, modified 101, 104

H

header files for C 627
 high-level subprograms 14

I

IAMAX 38
 IAMIN 41
 IASUM 44
 ICAMAX 5, 6, 22
 ICAMIN 25
 ICCTEQ 28
 ICCTNE 28
 ICLIP 53
 ICLIPL 56
 ICLIPR 59
 ICOPY 62
 ICSVEQ 35
 ICSVNE 35
 IDAMAX 22
 IDAMIN 25
 IDCTEQ 28
 IDCTGE 28
 IDCTGT 28
 IDCTLE 28
 IDCTLT 28
 IDCTNE 28
 IDMAX 31
 IDMIN 33
 IDSVEQ 35
 IDSVGE 35
 IDSVGT 35
 IDSVLE 35
 IDSVLT 35
 IDSVNE 35
 IEEE arithmetic format 10
 ier 14
 IGTHR 74
 IGTHRZ 76
 IIMAX 22
 IIMIN 25
 IICTEQ 28
 IICTGE 28
 IICTGT 28
 IICTLE 28

- IICLT 28
- IICTNE 28
- IIMAX 31
- IIMIN 33
- IISVEQ 35
- IISVGE 35
- IISVGT 35
- IISVLE 35
- IISVLT 35
- IISVNE 35
- ILSTEQ 79
- ILSTGE 79
- ILSTGT 79
- ILSTLE 79
- ILSTLT 79
- ILSTNE 79
- IMAX 83
- IMIN 85
- increment 19
- increment arguments 20
- INCX 19
- index 35, 79
- index of maximum 31
- index of maximum absolute value 22
- index of minimum 33
- index of minimum absolute value 25
- initConvexMLIB 634
- initialize skyline linear equations 463
- initialize sparse eigenvalues/eigenvectors 397
- initialize sparse linear equations 338
- initialize vector to zero 123
- inner product 65, 69
- inner product, weighted 119
- input, matrix structure, by column 340, 401, 465
- input, matrix structure, by finite element 342, 403, 467
- input, matrix structure, by matrix 344, 405, 469
- input, matrix structure, by single entry 339, 399, 464
- input, matrix structure, by skyline matrix 471
- input, matrix structure, end of 346, 408, 473
- input, matrix value, by column 350, 412, 478
- input, matrix value, by finite element 352, 416, 480
- input, matrix value, by matrix 354, 419, 482
- input, matrix value, by single entry 349, 410, 476
- input, matrix value, by skyline matrix 484
- input, matrix value, to main diagonal 414
- interlanguage programming 625
- inverse 235, 257, 291
- IPSUM 583
- IRAMP 91
- ISAMAX 5, 6, 22
- ISAMIN 5, 25
- ISCTEQ 28
- ISCTGE 28
- ISCTGT 28
- ISCTLE 28
- ISCTLT 28

- ISCTNE 28
- ISCTR 112
- ISMAX 5, 31
- ISMIN 5, 33
- ISORT 619
- ISSVEQ 35
- ISSVGE 35
- ISSVGT 35
- ISSVLE 35
- ISSVLT 35
- ISSVNE 35
- ISUM 114
- ISWAP 116
- IZAMAX 22
- IZAMIN 25
- IZCTEQ 28
- IZCTNE 28
- IZERO 123
- IZSVEQ 35
- IZSVNE 35

L

- Lanczos method 375
- LAPACK 5, 229, 625, 628, 633
- Level 1 BLAS 3
- Level 2 BLAS 3, 125, 229, 231, 305
- Level 3 BLAS 3, 125, 229, 231, 305
- linear equations 231
- linear equations, skyline 445
- linear equations, sparse 321
- linear recurrence 564, 566, 569, 572, 576, 586, 589, 592
- LINPACK 3, 231
- locate 35, 79
- low-level subprograms 14
- LU factorization 237, 246, 254, 261

M

- man pages 15
- matrix inverse 235, 257, 291
- matrix structure input by column 340, 401, 465
- matrix structure input by finite element 342, 403, 467
- matrix structure input by matrix 344, 405, 469
- matrix structure input by single entry 339, 399, 464
- matrix structure input by skyline matrix 471
- matrix value input by column 350, 412, 478
- matrix value input by finite element 352, 416, 480
- matrix value input by matrix 354, 419, 482
- matrix value input by single entry 349, 410, 476
- matrix value input by skyline matrix 484
- matrix value input to main diagonal 414
- matrix-matrix multiply 134, 169
- matrix-matrix multiply, triangular 213
- matrix-vector multiply 129, 143, 150, 155, 173, 193,

204, 217
 maximum 22, 31, 38, 83
 measure cpu time used 599
 measure wallclock time used 621
 message level 366, 441, 494
 minimum 25, 33, 41, 85
 mlib_getnumthreads 602
 mlib_setnumthreads 603
 modified Givens rotation 101, 104

N

NALLOC 604
 native arithmetic format 10
 nonvolatile dynamic memory 600, 604, 607
 norm 38, 44, 87, 89
 numeric factorization 357, 359, 486, 489

O

one-call usage 333
 one-call usage for skyline equations 457, 460
 online documentation 15
 ordering documentation xxv
 output control 366, 441, 494
 output unit 366, 441, 494

P

-p8 compiler option 3
 packed matrix 155, 160, 164, 204, 208
 parallel processing 7
 partial product 580
 partial sum 583
 -pd8 compiler option 3
 performance analysis 9
 polynomial, evaluate 577
 positive definite matrix 274, 278, 281, 285, 288, 291, 295, 298, 301
 print statistics 368, 442, 495
 product, partial 580
 profile data structure 445, 447
 profiling 9

R

RALLOC 607
 ramp function 91
 RAN 610
 random-number generator 610, 612, 614, 616
 rank-2k update 184
 rank-k update 189
 rank-one update 147, 160, 177
 rank-two update 164, 180
 RANV 612
 reallocate dynamic own array 607

recurrence, first order 564, 566, 569, 572, 576
 recurrence, linear 564, 566, 569, 572, 576, 586, 589, 592
 recurrence, second order 586, 589, 592
 reordering 347, 409
 reordering, automatic 474
 reordering, user-supplied 475
 restore sparse eigenvalue/eigenvector state 443
 restore sparse linear equations state 371, 496
 retrieve runtime statistics 372, 497
 return diagonal elements of matrix in its current form 369
 return eigenvalue results 436
 return sparse eigenvalue/eigenvector results 434
 RS 307
 runtime statistics 372, 497

S

S1DFFT 504
 S2DFFT 509
 S3DFFT 513
 SAMAX 38
 SAMIN 41
 SASUM 44
 save sparse eigenvalue/eigenvector state 444
 save sparse linear equation state 373, 498
 SAXPY 47
 SAXPYI 50
 SCAMAX 38
 SCAMIN 41
 SCASUM 44
 scatter 112
 SCHDC 235, 306, 637
 SCHDD 235, 306, 637
 SCHEX 235, 306, 637
 SCHUD 235, 306, 637
 SCILIB 5, 229, 625, 628, 633
 SCLIP 53
 SCLIPL 56
 SCLIPR 59
 SCNRM2 87
 SCNRSQ 89
 SCONV 555, 559
 SCOPY 62
 SDOT 65
 SDOTI 69
 search vector 35, 79
 second order linear recurrence 586, 589, 592
 selecting the reordering scheme 367
 SFFTS 519
 SFLR1C 564
 SFLR1M 566
 SFLR1P 566
 SFLR2C 569
 SFLR2M 572
 SFLR2P 572

SFLRLM 576
SFLRLP 576
SFRAC 72
SGBCO 237
SGBDI 242
SGBFA 246
SGBMV 129
SGBSL 251
SGECO 254
SGEDI 257
SGEFA 261
SGEMM 134
SGEMMS 6, 138
SGEMV 143
SGER 147
SGESL 264
SGTHR 74
SGTHRZ 76
SGTSL 268
SGTSLV 271
skyline data structure 445
skyline equations, one-call usage 457, 460
skyline linear equations 445
skyline linear equations, initialize 463
SLSTEQ 79
SLSTGE 79
SLSTGT 79
SLSTLE 79
SLSTLT 79
SLSTNE 79
SMAX 83
SMIN 85
SNRM2 87
SNRSQ 89
solve right hand side vector 487
solve right-hand side vector 491
solve sparse linear equations 361
sort array 619
sparse 17, 50, 69, 74, 76, 79, 99, 112, 321, 375
Sparse BLAS 3, 17
sparse eigenextraction 422, 427
sparse eigenvalues and eigenvectors 375, 389
sparse eigenvalues/eigenvectors, initialize 397
sparse linear equations 321
sparse linear equations, initialize 338
SPBCO 274
SPBDI 278
SPBFA 281
SPBSL 285
specify singularity treatment 364
SPOCO 288
SPODI 291
SPOFA 295
SPOSL 298
SPPCO 235, 306, 637
SPPROD 580
SPSUM 583
SPTSL 301
SRAMP 91
SRAN 614
SRANV 616
SRC1FT 526
SRC2FT 532
SRC3FT 538
SRCFTS 547
SROT 93
SROTG 97
SROTI 99
SROTM 101
SROTMG 104
SRSCL 107
SSBMV 150
SSCAL 109
SSCTR 112
SSLR2 586
SSLR3 589
SSLRL 592
SSORT 619
SSPFA 235, 306, 637
SSPMV 155
SSPR 160
SSPR2 164
SSPSL 235, 306, 637
SSUM 114
SSVDC 235, 306, 637
SSWAP 116
SSYMM 169
SSYMV 173
SSYR 177
SSYR2 180
SSYR2K 184
SSYRK 189
standardization 3
STBMV 193
STBSV 198
storage, backward 20
storage, forward 19
STPMV 204
STPSV 208
Strassen matrix-matrix multiply 138
STRCO 235, 306, 637
STRDI 235, 306, 637
stride 19
stride arguments 20
STRMM 213
STRMV 217
STRSL 235, 306, 637
STRSM 221
STRSV 225
Subprograms not in this Guide 637
sum, partial 583
supplemental reading xxiii
SWDOT 119
symbolic factorization 347, 409

SZERO 123

T

technical assistance center, TAC xxv

thread, definition 7

TQL2 310

TQLRAT 313

TRED1 316

TRED2 318

triangular factorization 237, 246, 254, 261

triangular matrix-matrix multiply 213

triangular solve 198, 208, 221, 225

tridiagonal linear equations 268, 271, 301

U

user-supplied reordering 475

V

variable-band data structure 445, 447

VECLIB 3, 5

VECLIB error handling 14

VECLIB8 3

VectorPak 3

W

walltime 621

X

XERBLA 229

XERVEC 622

Z

Z1DFFT 501

Z2DFFT 507

Z3DFFT 511

ZAXPY 47

ZAXPYC 47

ZAXPYI 50

ZCHDC 235, 306, 637

ZCHDD 235, 306, 637

ZCHEX 235, 306, 637

ZCHUD 235, 306, 637

ZCOPY 62

ZCOPYC 62

ZDOTC 65

ZDOTCI 69

ZDOTU 65

ZDOTUI 69

ZDROT 93

ZDRSCL 107

ZDSCAL 109

zero vector 123

ZFFTS 515

ZGBCO 237

ZGBDI 242

ZGBFA 246

ZGBMV 129

ZGBSL 251

ZGECO 254

ZGEDI 257

ZGEFA 261

ZGEMM 134

ZGEMMS 138

ZGEMV 143

ZGERC 147

ZGERU 147

ZGESL 264

ZGTHR 74

ZGTHRZ 76

ZGTSL 268

ZGTSLV 271

ZHBMV 150

ZHEMM 169

ZHEMV 173

ZHER 177

ZHER2 180

ZHER2K 184

ZHERK 189

ZHICO 235, 306, 637

ZHIDI 235, 306, 637

ZHIFA 235, 306, 637

ZHISL 235, 306, 637

ZHPCO 235, 306, 637

ZHPDI 235, 306, 637

ZHPFA 235, 306, 637

ZHPMV 155

ZHPR 160

ZHPR2 164

ZHPSL 235, 306, 637

ZLSTEQ 79

ZLSTNE 79

ZPBCO 274

ZPBDI 278

ZPBFA 281

ZPBSL 285

ZPOCO 288

ZPODI 291

ZPOFA 295

ZPOSL 298

ZPPCO 235, 306, 637

ZPTSL 301

ZRC1FT 523

ZRC2FT 529

ZRC3FT 535

ZRCFTS 542

ZROT 93

ZROTG 97
ZRSCL 107
ZSCAL 109
ZSCALC 109
ZSCTR 112
ZSPDI 235, 306, 637
ZSPFA 235, 306, 637
ZSPSL 235, 306, 637
ZSUM 114
ZSVDC 235, 306, 637
ZSWAP 116
ZSYMM 169
ZSYR2K 184
ZSYRK 189
ZTBMV 193
ZTBSV 198
ZTPMV 204
ZTPSV 208
ZTRCO 235, 306, 637
ZTRDI 235, 306, 637
ZTRMM 213
ZTRMV 217
ZTRSL 235, 306, 637
ZTRSM 221
ZTRSV 225
ZWDOTC 119
ZWDOTU 119
ZZERO 123



CONVEX
PRESS

B5649-90007

