CONVEX VECLIB User's Guide

Seventh Edition

CONVEX

CONVEX COMPUTER CORPORATION

CONVEX VECLIB
User's Guide

Document No. 710-011030-002

Seventh Edition
February 1993

CONVEX Computer Corporation
Richardson, Texas USA

CONVEX VECLIB User's Guide
Order No. DSW-132
Seventh Edition

© 1987, 1988, 1989, 1990, 1991, 1993 CONVEX Computer Corporation
All rights reserved.

This document is copyrighted. This document may not, in whole or part, be copied, duplicated, reproduced, translated, stored electronically, or reduced to machine-readable form without prior written consent from CONVEX Computer Corporation.

Although the material contained herein has been carefully reviewed, CONVEX Computer Corporation (CONVEX) does not warrant it to be free of errors or omissions. CONVEX reserves the right to make corrections, updates, revisions or changes to the information contained herein. CONVEX does not warrant the material described herein to be free of patent infringement.

UNLESS PROVIDED OTHERWISE IN WRITING WITH CONVEX COMPUTER CORPORATION (CONVEX), THE PROGRAM DESCRIBED HEREIN IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES. THE ABOVE EXCLUSION MAY NOT BE APPLICABLE TO ALL PURCHASERS BECAUSE WARRANTY RIGHTS CAN VARY FROM STATE TO STATE. IN NO EVENT WILL CONVEX BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, INCLUDING ANY LOST PROFITS OR LOST SAVINGS, ARISING OUT OF THE USE OR INABILITY TO USE THIS PROGRAM. CONVEX WILL NOT BE LIABLE EVEN IF IT HAS BEEN NOTIFIED OF THE POSSIBILITY OF SUCH DAMAGE BY THE PURCHASER OR ANY THIRD PARTY.

CONVEX and the CONVEX logo ("C") are registered trademarks of CONVEX Computer Corporation.
C1, C2, C3, C Series, and ASAP are trademarks of CONVEX Computer Corporation.
ConvexOS and VECLIB are trademarks of CONVEX Computer Corporation.
Cray and UNICOS are registered trademarks of Cray Research, Inc.
FPS, AP120B, and APLIB are trademarks of Floating Point Systems, Inc.
IBM is a trademark of International Business Machines Corporation.
IEEE is a trademark of the Institute of Electrical and Electronics Engineers, Inc.
VAX and VMS are trademarks of Digital Equipment Corporation.
VectorPak is a trademark of The Boeing Company.

Printed in the United States of America

Revision information for
CONVEX VECLIB User's Guide

Edition	Document No.	Description
Seventh	710-011030-002	Released with CONVEX VECLIB software V8.0, February 1993. Reorganized Chapter 1. Strassen method matrix multiply subprogram descriptions were added to Chapter 3. Chapters 8 to 11 were renumbered 9 to 12. A new Chapter 8 was inserted to describe Skyline Linear Equation subprograms. Long period random number generator subprogram descriptions were added to Chapter 12. Corrected miscellaneous errors. Added information about header files, error handling capabilities, and more examples to Appendix A.
Sixth	710-011030-001	Released with CONVEX VECLIB software V7.0, August 1991.
Fifth	710-011030-000	Released with CONVEX VECLIB software V6.0, September 1990.
Fourth	740-002330-205	Released with CONVEX VECLIB software V5.0, December 1989.
Third Rev. 2	740-002330-203	Released with CONVEX VECLIB software V4.0, October 1988.
Third Rev. 1	740-002330-202	Released with CONVEX VECLIB software V3.1, March 1988.
3.0	740-002330-201	Released with CONVEX VECLIB software V3.0, December 1987.
2.0	740-002330-200	Released with CONVEX VECLIB software V2.0, June 1987.
1.0	740-001630-000	Released with CONVEX VECLIB software V1.3, 1986.

Table of Contents

1 Introduction to CONVEX VECLIB

Overview 1-1
Chapter Objectives 1-1
What You Need to Know to Use CONVEX VECLIB 1-2
Standardization 1-2
Two CONVEX VECLIB Libraries 1-2
Accessing CONVEX VECLIB 1-2
Interactions Between VECLIB, SCILIB, and LAPACK 1-3
Performance Value 1-4
Optimization 1-4
Parallel Processing 1-4
Profiling CONVEX VECLIB Applications 1-5
Optimizing with the Application Compiler 1-6
Floating Point Formats 1-6
Roundoff Effects 1-6
Data Types and Precision 1-6
VECLIB Naming Convention 1-7
Required Data Item Byte Lengths and How to Get Them 1-7
Error Handling 1-9
CONVEX VECLIB Programmer's Reference 1-10
Support Services 1-10
Suggestions 1-11
Supplemental Reading 1-11

2 Basic Vector Operations

Overview 2-1
Chapter Objectives 2-1
What You Need to Know to Use These Subprograms 2-1
BLAS Storage Conventions 2-1
Supplemental Reading 2-4
Subprogram Descriptions 2-4

3 Basic Matrix Operations

Overview 3-1
Chapter Objectives 3-1
What You Need to Know to Use These Subprograms 3-1
Subroutine Naming Convention 3-1
Supplemental Reading 3-3
Subprogram Descriptions 3-3

4 Linear Equations

Overview 4-1
Chapter Objectives 4-1
What You Need to Know to Use These Subprograms 4-2
Subroutine Naming Convention 4-2
Condition Number 4-4
Determinant and Inverse 4-4
Supplemental Reading 4-5
Subprogram Descriptions 4-5
LINPACK Subprograms not in the *CONVEX VECLIB User's Guide* 4-60

5 **Eigenvalues and Eigenvectors**
Overview 5-1
Chapter Objectives 5-1
What You Need to Know to Use These Subprograms 5-1
Supplemental Reading 5-2
Subprogram Descriptions 5-2
EISPACK Subprograms not in the *CONVEX VECLIB User's Guide* 5-14

6 **Sparse Linear Equations**
Overview 6-1
Chapter Objectives 6-1
What You Need to Know to Use These Subprograms 6-1
Sparsity and Storage Formats 6-1
Direct Versus Iterative Solution 6-3
Fill and Reordering 6-3
Stability 6-4
Global Communications Array 6-4
Error Convention 6-5
Output Controls 6-5
Paths of Control 6-5
Sample Program 6-7
Supplemental Reading 6-8
Subprogram Descriptions 6-9

7 **Sparse Eigenvalues and Eigenvectors**
Overview 7-1
Chapter Objectives 7-1
What You Need to Know to Use These Subprograms 7-1
Generalized Symmetric Eigenproblems 7-1
Sparsity and Storage Formats 7-2
Description of Sparse Eigenvalue Problems 7-3
Trust Regions and Matrix Inertias (Sturm Sequence Counts) 7-4
Convention for Returning Eigenvalues and Eigenvectors 7-5
Global Communications Array 7-6
Error Convention 7-6
Output Controls 7-6
Paths of Control 7-6
Sample Program 7-8
Supplemental Reading 7-10
Subprogram Descriptions 7-10

8 **Skyline Linear Equations**
Overview 8-1
Chapter Objectives 8-1
What You Need to Know to Use These Subprograms 8-1
Sparsity and Storage Formats 8-1
Direct Versus Iterative Solution 8-4
Fill and Reordering 8-4
Stability 8-4
Global Communications Array 8-5
Error Convention 8-5
Output Controls 8-5
Paths of Control 8-6
Sample Program 8-9
Supplemental Reading 8-10
Subprogram Descriptions 8-11

9 Fast Fourier Transforms
Overview 9-1
Chapter Objectives 9-1
What You Need to Know to Use These Subprograms 9-1
Supplemental Reading 9-2
Subprogram Descriptions 9-2

10 Correlation and Convolution Subprograms
Overview 10-1
Chapter Objectives 10-1
What You Need to Know to Use These Subprograms 10-1
Supplemental Reading 10-1
Subprogram Descriptions 10-1

11 Linear Recurrences
Overview 11-1
Chapter Objectives 11-1
What You Need to Know to Use These Subprograms 11-1
Subprogram Descriptions 11-1

12 Miscellaneous Routines
Overview 12-1
Chapter Objective 12-1
What You Need to Know to Use These Subprograms 12-1
Dynamic Memory Subprograms 12-1
Supplemental Reading 12-2
Subprogram Descriptions 12-2

Appendices

A Calling CONVEX VECLIB from C A-1
Introduction A-1
General Interlanguage Programming Rules for VECLIB A-1
Header Files A-3
Examples A-5
Linking VECLIB Subprograms into a C Program A-8
Error Handling A-9

B Calling CONVEX VECLIB From Ada B-1
Introduction B-1
General Rules B-1
Calling Procedures B-3
Calling Functions B-4
Sample VECLIB Interface Package B-6

List of Tables

1-1 VECLIB Naming Convention—Data Type 1-7
1-2 Data Item Byte Length vs. Data Type and Library 1-7
1-3 Data Item Byte Length vs. Declaration and Compiler Option 1-8
3-1 Extended BLAS Naming Convention — Data Type 3-2
3-2 Extended BLAS Naming Convention — Matrix Form 3-2
3-3 Extended BLAS Naming Convention — Computation 3-2
3-4 Extended BLAS Naming Convention — Subprogram Names 3-3
4-1 LINPACK Naming Convention — Data Type 4-2

4-2 LINPACK Naming Convention — Form or Decomposition 4-2
4-3 LINPACK Naming Convention — Computation 4-3
4-4 LINPACK Naming Convention — Subprogram Names 4-3
4-5 LINPACK Subprograms not in the *CONVEX VECLIB User's Guide* 4-60
5-1 EISPACK Subprograms not in the *CONVEX VECLIB User's Guide* 5-14
A-1 Relationship Between FORTRAN and C Declarations A-3
A-2. Libraries to Satisfy Unresolved Symbols at Link Time A-8
B-1 Relationship Between FORTRAN and Ada Declarations B-2

List of Figures

6-1 Row and Column Index Sparse Matrix Representation 6-2
6-2 Column Pointer, Row Index Sparse Matrix Representation 6-2
6-3 Paths of Control 6-6
7-1 Row and Column Index Sparse Matrix Representation 7-2
7-2 Column Pointer, Row Index Sparse Matrix Representation 7-3
7-3 Paths of Control 7-7
8-1 Row and Column Index Sparse Matrix Representation 8-2
8-2 Skyline Matrix Representation 8-3
8-3 Reverse Skyline Matrix Representation 8-3
8-4 Column Pointer, Row Index Sparse Matrix Representation 8-3
8-5 Paths of Control 8-7
A-1 Calling VECLIB Subroutines from FORTRAN and C A-5
A-2 Matrix Multiplication via an Interface Function A-6
A-3 Calling a Real-Valued Function from C A-7
A-4 Calling a Complex-Valued Function from C A-7
A-5 Default VECLIB Error Handling in a C Program A-9
A-6 Changing the VECLIB Error Handling Signal A-10
A-7 Changing the Error Handling Signal Handler A-11

Permuted Index

complex	1-dimensional FFT	9-C1DFFT
simultaneous complex	1-dimensional FFT	9-CFFTS
real-to-complex	1-dimensional FFT	9-CRC1FT
simultaneous real-to-complex	1-dimensional FFT	9-CRCFTS
	1-norm of a vector	2-SASUM
Level	2 and Level 3 BLAS error routine	3-XERBLA
complex	2-dimensional FFT	9-C2DFFT
real-to-complex	2-dimensional FFT	9-CRC2FT
	2-norm of a vector	2-SNRM2
square of	2-norm of a vector	2-SNRSQ
Level 2 and Level	3 BLAS error routine	3-XERBLA
complex	3-dimensional FFT	9-C3DFFT
real-to-complex	3-dimensional FFT	9-CRC3FT
maximum	absolute value of vector elements	2-SAMAX
minimum	absolute value of vector elements	2-SAMIN
sum of	absolute values of vector elements	2-SASUM
	accept user-provided skyline matrix solver reordering	8-DSKYOU
check	accuracy of sparse eigenvalue and eigenvector results	7-DSEVCK
scalar multiple of a vector	added to a vector	2-SAXPY
scalar times a vector	added to a vector	2-SAXPY
scalar multiple of a sparse vector	added to a vector	2-SAXPYI
scalar times a sparse vector	added to a vector	2-SAXPYI
volatile dynamic storage	allocation	12-DYNAMIC
volatile dynamic storage	allocation	12-MALLOC
non-volatile dynamic storage	allocation	12-NALLOC
	apply a Givens rotation matrix	2-SROT
	apply a modified Givens rotation matrix	2-SROTM
	apply a sparse Givens rotation matrix	2-SROTI
sort the elements of a vector into	ascending or descending order	12-SSORT
triangular	band equation solver	3-STBSV
triangular packed	band equation solver	3-STPSV
determinant of a general	band matrix	4-SGBDI
factor a general	band matrix	4-SGBFA
determinant of a positive definite symmetric	band matrix	4-SPBDI
factor a positive definite symmetric	band matrix	4-SPBFA
factor a general	band matrix and estimate its condition number	4-SGBCO
factor a positive definite symmetric	band matrix and estimate its condition number	4-SPBCO
	band matrix-vector multiply	3-SGBMV
symmetric or Hermitian	band matrix-vector multiply	3-SSBMV
symmetric or Hermitian packed	band matrix-vector multiply	3-SSPMV
triangular	band matrix-vector multiply	3-STBMV
triangular packed	band matrix-vector multiply	3-STPMV
	band matrix-vector product	3-SGBMV
symmetric or Hermitian	band matrix-vector product	3-SSBMV
symmetric or Hermitian packed	band matrix-vector product	3-SSPMV
triangular	band matrix-vector product	3-STBMV
triangular packed	band matrix-vector product	3-STPMV
solve a general	band system using a factored matrix	4-SGBSL
solve a positive definite symmetric	band system using a factored matrix	4-SPBSL
Level 2 and Level 3	BLAS error routine	3-XERBLA
	build an array of indices of vector elements = scalar	2-SLSTEQ
scalar	build an array of indices of vector elements ≥	2-SLSTGE
	build an array of indices of vector elements > scalar	2-SLSTGT
scalar	build an array of indices of vector elements ≤	2-SLSTLE
	build an array of indices of vector elements < scalar	2-SLSTLT
scalar	build an array of indices of vector elements ≠	2-SLSTNE
results	check accuracy of sparse eigenvalue and eigenvector	7-DSEVCK
	clear a vector to zero	2-SZERO
	clip vector elements to left-ended range	2-SCLIPL
	clip vector elements to right-ended range	2-SCLIPR
	clip vector elements to specified range	2-SCLIP
sparse matrix structure input by finite element or	clique	6-DSLEIE
sparse matrix value input by finite element or	clique	6-DSLEVE
sparse matrix structure input by finite element or	clique	7-DSEVIE
sparse matrix value input by finite element or	clique	7-DSEVVE
skyline matrix structure input by finite element or	clique	8-DSKYIE
skyline matrix value input by finite element or	clique	8-DSKYVE

solve a first order linear recurrence with constant	coefficient	11-SFLR1C
solve a first order linear recurrence with constant	coefficient	11-SFLR2C
sparse matrix structure input by	column	6-DSLEIC
sparse matrix value input by	column	6-DSLEVC
sparse matrix structure input by	column	7-DSEVIC
sparse matrix value input by	column	7-DSEVVC
skyline matrix structure input by	column	8-DSKYIC
skyline matrix value input by	column	8-DSKYVC
	complex 1-dimensional FFT	9-C1DFFT
simultaneous	complex 1-dimensional FFT	9-CFFTS
	complex 2-dimensional FFT	9-C2DFFT
	complex 3-dimensional FFT	9-C3DFFT
	compute the vector of partial products of a vector	11-SPPROD
	compute the vector of partial sums of a vector	11-SPSUM
factor a general band matrix and estimate its	condition number	4-SGBCO
factor a general matrix and estimate its	condition number	4-SGECO
definite symmetric band matrix and estimate its	condition number factor a positive	4-SPBCO
a positive definite symmetric matrix and estimate its	condition number factor	4-SPOCO
sparse matrix solver numeric factorization and	condition number estimation	6-DSLECO
solve a first order linear recurrence with	constant coefficient	11-SFLR1C
solve a first order linear recurrence with	constant coefficient	11-SFLR2C
	construct a Givens rotation matrix	2-SROTG
	construct a linear ramp function	2-SRAMP
	construct a modified Givens rotation matrix	2-SROTMG
sparse matrix solver output	control	6-DSLEOC
sparse eigenvalue solver output	control	7-DSEVOC
skyline matrix solver output	control	8-DSKYOC
IBM format	convert floating-point numbers from CONVEX format to	12-SC2IBM
CONVEX format	convert floating-point numbers from IBM format to	12-SIBM2C
convert floating-point numbers from IBM format to	CONVEX format	12-SIBM2C
convert floating-point numbers from	CONVEX format to IBM format	12-SC2IBM
	convolution/correlation	10-SCONV
	copy one vector to another	2-SCOPY
convolution/	correlation	10-SCONV
	count occurrences of vector elements = scalar	2-ISCTEQ
	count occurrences of vector elements ≥ scalar	2-ISCTGE
	count occurrences of vector elements > scalar	2-ISCTGT
	count occurrences of vector elements ≤ scalar	2-ISCTLE
	count occurrences of vector elements < scalar	2-ISCTLT
	count occurrences of vector elements ≠ scalar	2-ISCTNE
	deallocate skyline matrix solver working storage	8-DSKYDA
	deallocate sparse eigenvalue solver working storage	7-DSEVDA
	deallocate sparse matrix solver working storage	6-DSLEDA
non-volatile dynamic storage	deallocation	12-DALLOC
sort the elements of a vector into ascending or	descending order	12-SSORT
	determinant of a general band matrix	4-SGBDI
matrix	determinant of a positive definite symmetric band	4-SPBDI
	determinant or inverse of a general matrix	4-SGEDI
symmetric matrix	determinant or inverse of a positive definite	4-SPODI
sparse matrix value input to main	diagonal	7-DSEVVD
factorization	direct input skyline matrix solver numeric	8-DSKYDF
solve a	direct input skyline matrix system	8-DSKYDS
vector	divided by a scalar	2-SRSCL
	dot product of two vectors	2-SDOT
sparse	dot product of two vectors	2-SDOTI
weighted	dot product of two vectors	2-SWDOT
volatile	dynamic storage allocation	12-DYNAMIC
volatile	dynamic storage allocation	12-MALLOC
non-volatile	dynamic storage allocation	12-NALLOC
non-volatile	dynamic storage deallocation	12-DALLOC
non-volatile	dynamic storage reallocation	12-RALLOC
check accuracy of sparse	eigenvalue and eigenvector results	7-DSEVCK
return sparse	eigenvalue and eigenvector results	7-DSEVRC
return sparse	eigenvalue results	7-DSEVRL
one-call sparse	eigenvalue solver	7-DSEVE1
sparse	eigenvalue solver initialization	7-DSEVIN
sparse	eigenvalue solver output control	7-DSEVOC
restore sparse	eigenvalue solver problem state from a savefile	7-DSEVRS
save sparse	eigenvalue solver problem state to a savefile	7-DSEVSV
factorization sparse	eigenvalue solver reordering and symbolic	7-DSEVOR
deallocate sparse	eigenvalue solver working storage	7-DSEVDA
extract sparse	eigenvalues and eigenvectors	7-DSEVES

extract sparse	eigenvalues and eigenvectors	7-DSEVEX
matrix	eigenvalues and eigenvectors of a real symmetric	5-RS
tridiagonal matrix	eigenvalues and eigenvectors of a real symmetric	5-TQL2
	eigenvalues of a real symmetric tridiagonal matrix	5-TQLRAT
check accuracy of sparse eigenvalue and	eigenvector results	7-DSEVCK
return sparse eigenvalue and	eigenvector results	7-DSEVRC
extract sparse eigenvalues and	eigenvectors	7-DSEVES
extract sparse eigenvalues and	eigenvectors	7-DSEVEX
eigenvalues and	eigenvectors of a real symmetric matrix	5-RS
eigenvalues and	eigenvectors of a real symmetric tridiagonal matrix	5-TQL2
	end of skyline matrix structure input	8-DSKYIF
	end of sparse matrix structure input	6-DSLEIF
	end of sparse matrix structure input	7-DSEVIF
sparse matrix structure input by single	entry	6-DSLEI1
sparse matrix value input by single	entry	6-DSLEV1
sparse matrix structure input by single	entry	7-DSEVI1
sparse matrix value input by single	entry	7-DSEVV1
skyline matrix structure input by single	entry	8-DSKYI1
skyline matrix value input by single	entry	8-DSKYV1
triangular band	equation solver	3-STBSV
triangular packed band	equation solver	3-STPSV
triangular	equation solver	3-STRSV
solve a general tridiagonal system of linear	equations	4-SGTSL
solve a general tridiagonal system of linear	equations	4-SGTSV
a positive definite tridiagonal system of linear	equations solve	4-SPTSL
simultaneous triangular	equations solver	3-STRSM
VECLIB	error handler	12-XERVEC
Level 2 and Level 3 BLAS	error routine	3-XERBLA
factor a general band matrix and	estimate its condition number	4-SGBCO
factor a general matrix and	estimate its condition number	4-SGECO
factor a positive definite symmetric band matrix and	estimate its condition number	4-SPBCO
factor a positive definite symmetric matrix and	estimate its condition number	4-SPOCO
solver numeric factorization and condition number	estimation sparse matrix	6-DSLECO
	Euclidean norm of a vector	2-SNRM2
square of	Euclidean norm of a vector	2-SNRSQ
	exchange two vectors	2-SSWAP
	extract sparse eigenvalues and eigenvectors	7-DSEVES
	extract sparse eigenvalues and eigenvectors	7-DSEVEX
	factor a general band matrix	4-SGBFA
condition number	factor a general band matrix and estimate its	4-SGBCO
	factor a general matrix	4-SGEFA
number	factor a general matrix and estimate its condition	4-SGECO
	factor a positive definite symmetric band matrix	4-SPBFA
estimate its condition number	factor a positive definite symmetric band matrix and	4-SPBCO
	factor a positive definite symmetric matrix	4-SPOFA
estimate its condition number	factor a positive definite symmetric matrix and	4-SPOCO
sparse matrix solver numeric	factorization	6-DSLEFA
sparse matrix solver reordering and symbolic	factorization	6-DSLEOR
sparse eigenvalue solver reordering and symbolic	factorization	7-DSEVOR
direct input skyline matrix solver numeric	factorization	8-DSKYDF
skyline matrix solver numeric	factorization	8-DSKYFA
sparse matrix solver numeric	factorization and condition number estimation	6-DSLECO
complex 1-dimensional	FFT	9-C1DFFT
complex 2-dimensional	FFT	9-C2DFFT
complex 3-dimensional	FFT	9-C3DFFT
simultaneous complex 1-dimensional	FFT	9-CFFTS
real-to-complex 1-dimensional	FFT	9-CRC1FT
real-to-complex 2-dimensional	FFT	9-CRC2FT
real-to-complex 3-dimensional	FFT	9-CRC3FT
simultaneous real-to-complex 1-dimensional	FFT	9-CRCFTS
sparse matrix structure input by	finite element or clique	6-DSLEIE
sparse matrix value input by	finite element or clique	6-DSLEVE
sparse matrix structure input by	finite element or clique	7-DSEVIE
sparse matrix value input by	finite element or clique	7-DSEVVE
skyline matrix structure input by	finite element or clique	8-DSKYIE
skyline matrix value input by	finite element or clique	8-DSKYVE
format convert	floating-point numbers from CONVEX format to IBM	12-SC2IBM
format convert	floating-point numbers from IBM format to CONVEX	12-SIBM2C
floating-point numbers from CONVEX format to IBM	format convert	12-SC2IBM
floating-point numbers from IBM format to CONVEX	format convert	12-SIBM2C
convert floating-point numbers from IBM	format to CONVEX format	12-SIBM2C
convert floating-point numbers from CONVEX	format to IBM format	12-SC2IBM

	fractional parts of vector elements	2-SFRAC
construct a linear ramp	function	2-SRAMP
	gather a sparse vector	2-SGTHR
	gather and zero a sparse vector	2-SGTHRZ
determinant of a	general band matrix	4-SGBDI
factor a	general band matrix	4-SGBFA
factor a	general band matrix and estimate its condition number	4-SGBCO
solve a	general band system using a factored matrix	4-SGBSL
determinant or inverse of a	general matrix	4-SGEDI
factor a	general matrix	4-SGEFA
factor a	general matrix and estimate its condition number	4-SGECO
	general matrix-matrix multiply	3-SGEMM
Strassen	general matrix-matrix multiply	3-SGEMMS
	general matrix-matrix product	3-SGEMM
Strassen	general matrix-matrix product	3-SGEMMS
	general matrix-vector multiply	3-SGEMV
	general matrix-vector product	3-SGEMV
	general rank-1 update	3-SGER
solve a	general system using a factored matrix	4-SGESL
solve a	general tridiagonal system of linear equations	4-SGTSL
solve a	general tridiagonal system of linear equations	4-SGTSV
VAX-compatible scalar random number	generator	12-RAN
VAX-compatible vectorized random number	generator	12-RANV
long period scalar random number	generator	12-SRAN
long period vectorized number	generator	12-SRANV
apply a	Givens rotation matrix	2-SROT
construct a	Givens rotation matrix	2-SROTG
apply a sparse	Givens rotation matrix	2-SROTI
apply a modified	Givens rotation matrix	2-SROTM
construct a modified	Givens rotation matrix	2-SROTMG
VECLIB error	handler	12-XERVEC
symmetric or	Hermitian band matrix-vector multiply	3-SSBMV
symmetric or	Hermitian band matrix-vector product	3-SSBMV
symmetric or	Hermitian matrix-matrix multiply	3-SSYMM
symmetric or	Hermitian matrix-matrix product	3-SSYMM
symmetric or	Hermitian matrix-vector multiply	3-SSYMV
symmetric or	Hermitian matrix-vector product	3-SSYMV
symmetric or	Hermitian packed band matrix-vector multiply	3-SSPMV
symmetric or	Hermitian packed band matrix-vector product	3-SSPMV
symmetric or	Hermitian packed rank-1 update	3-SSPR
symmetric or	Hermitian packed rank-2 update	3-SSPR2
symmetric or	Hermitian rank-1 update	3-SSYR
symmetric or	Hermitian rank-2 update	3-SSYR2
symmetric or	Hermitian rank-2k update	3-SSYR2K
symmetric or	Hermitian rank-k update	3-SSYRK
convert floating-point numbers from CONVEX format to	IBM format	12-SC2IBM
convert floating-point numbers from	IBM format to CONVEX format	12-SIBM2C
scalar	index of first occurrence of a vector element =	2-ISSVEQ
scalar	index of first occurrence of a vector element ≥	2-ISSVGE
scalar	index of first occurrence of a vector element >	2-ISSVGT
scalar	index of first occurrence of a vector element ≤	2-ISSVLE
scalar	index of first occurrence of a vector element <	2-ISSVLT
scalar	index of first occurrence of a vector element ≠	2-ISSVNE
maximum magnitude	index of first occurrence of a vector element of	2-ISAMAX
maximum value	index of first occurrence of a vector element of	2-ISMAX
minimum magnitude	index of first occurrence of a vector element of	2-ISAMIN
minimum value	index of first occurrence of a vector element of	2-ISMIN
build an array of	indices of vector elements = scalar	2-SLSTEQ
build an array of	indices of vector elements ≥ scalar	2-SLSTGE
build an array of	indices of vector elements > scalar	2-SLSTGT
build an array of	indices of vector elements ≤ scalar	2-SLSTLE
build an array of	indices of vector elements < scalar	2-SLSTLT
build an array of	indices of vector elements ≠ scalar	2-SLSTNE
	infinity norm of a vector	2-SAMAX
sparse matrix solver	initialization	6-DSLEIN
sparse eigenvalue solver	initialization	7-DSEVIN
skyline matrix solver	initialization	8-DSKYIN
	initialize a vector to zero	2-SZERO
	inner product of two vectors	2-SDOT
sparse	inner product of two vectors	2-SDOTI
weighted	inner product of two vectors	2-SWDOT
end of sparse matrix structure	input	6-DSLEIF

end of sparse matrix structure	input	7-DSEVIF
end of skyline matrix structure	input	8-DSKYIF
sparse matrix structure	input by column	6-DSLEIC
sparse matrix value	input by column	6-DSLEVC
sparse matrix structure	input by column	7-DSEVIC
sparse matrix value	input by column	7-DSEVVC
skyline matrix structure	input by column	8-DSKYIC
skyline matrix value	input by column	8-DSKYVC
sparse matrix structure	input by finite element or clique	6-DSLEIE
sparse matrix value	input by finite element or clique	6-DSLEVE
sparse matrix structure	input by finite element or clique	7-DSEVIE
sparse matrix value	input by finite element or clique	7-DSEVVE
skyline matrix structure	input by finite element or clique	8-DSKYIE
skyline matrix value	input by finite element or clique	8-DSKYVE
sparse matrix structure	input by matrix	6-DSLEIM
sparse matrix value	input by matrix	6-DSLEVM
sparse matrix structure	input by matrix	7-DSEVIM
sparse matrix value	input by matrix	7-DSEVVM
skyline matrix structure	input by matrix	8-DSKYIM
skyline matrix value	input by matrix	8-DSKYVM
sparse matrix structure	input by single entry	6-DSLEI1
sparse matrix value	input by single entry	6-DSLEV1
sparse matrix structure	input by single entry	7-DSEVI1
sparse matrix value	input by single entry	7-DSEVV1
skyline matrix structure	input by single entry	8-DSKYI1
skyline matrix value	input by single entry	8-DSKYV1
skyline matrix structure	input by skyline matrix	8-DSKYIS
skyline matrix value	input by skyline matrix	8-DSKYVS
direct	input skyline matrix solver numeric factorization	8-DSKYDF
solve a direct	input skyline matrix system	8-DSKYDS
sparse matrix value	input to main diagonal	7-DSEVVD
	interchange two vectors	2-SSWAP
determinant or	inverse of a general matrix	4-SGEDI
determinant or	inverse of a positive definite symmetric matrix	4-SPODI
clip vector elements to	left-ended range	2-SCLIPL
	Level 2 and Level 3 BLAS error routine	3-XERBLA
Level 2 and	Level 3 BLAS error routine	3-XERBLA
solve a general tridiagonal system of	linear equations	4-SGTSL
solve a general tridiagonal system of	linear equations	4-SGTSV
solve a positive definite tridiagonal system of	linear equations	4-SPTSL
construct a	linear ramp function	2-SRAMP
solve a first order	linear recurrence	11-SFLR1
solve a first order	linear recurrence	11-SFLR2
solve for the last term of a first order	linear recurrence	11-SFLRL
solve a second order	linear recurrence	11-SSLR2
solve a second order	linear recurrence	11-SSLR3
solve for the last term of a second order	linear recurrence	11-SSLRL
solve a first order	linear recurrence with constant coefficient	11-SFLR1C
solve a first order	linear recurrence with constant coefficient	11-SFLR2C
of first occurrence of a vector element of maximum	magnitude index	2-ISAMAX
of first occurrence of a vector element of minimum	magnitude index	2-ISAMIN
maximum	magnitude of vector elements	2-SAMAX
minimum	magnitude of vector elements	2-SAMIN
sum of	magnitudes of vector elements	2-SASUM
sparse matrix value input to	main diagonal	7-DSEVVD
apply a Givens rotation	matrix	2-SROT
construct a Givens rotation	matrix	2-SROTG
apply a sparse Givens rotation	matrix	2-SROTI
apply a modified Givens rotation	matrix	2-SROTM
construct a modified Givens rotation	matrix	2-SROTMG
determinant of a general band	matrix	4-SGBDI
factor a general band	matrix	4-SGBFA
solve a general band system using a factored	matrix	4-SGBSL
determinant or inverse of a general	matrix	4-SGEDI
factor a general	matrix	4-SGEFA
solve a general system using a factored	matrix	4-SGESL
determinant of a positive definite symmetric band	matrix	4-SPBDI
factor a positive definite symmetric band	matrix	4-SPBFA
definite symmetric band system using a factored	matrix solve a positive	4-SPBSL
or inverse of a positive definite symmetric	matrix determinant	4-SPODI
factor a positive definite symmetric	matrix	4-SPOFA
a positive definite symmetric system using a factored	matrix solve	4-SPOSL

eigenvalues and eigenvectors of a real symmetric	matrix	5-RS
and eigenvectors of a real symmetric tridiagonal	matrix eigenvalues	5-TQL2
eigenvalues of a real symmetric tridiagonal	matrix	5-TQLRAT
sparse matrix structure input by	matrix	6-DSLEIM
solve a sparse matrix system using a factored sparse	matrix	6-DSLESL
sparse matrix value input by	matrix	6-DSLEVM
sparse matrix structure input by	matrix	7-DSEVIM
sparse matrix value input by	matrix	7-DSEVVM
skyline matrix structure input by	matrix	8-DSKYIM
skyline matrix structure input by skyline	matrix	8-DSKYIS
a skyline matrix system using a factored skyline	matrix solve	8-DSKYSL
skyline matrix value input by	matrix	8-DSKYVM
skyline matrix value input by skyline	matrix	8-DSKYVS
factor a general band	matrix and estimate its condition number	4-SGBCO
factor a general	matrix and estimate its condition number	4-SGECO
factor a positive definite symmetric band	matrix and estimate its condition number	4-SPBCO
factor a positive definite symmetric	matrix and estimate its condition number	4-SPOCO
one-call usage sparse	matrix solver	6-DSLEFS
one-call usage skyline	matrix solver	8-DSKYFS
one-call usage skyline	matrix solver	8-DSKYFX
sparse	matrix solver initialization	6-DSLEIN
skyline	matrix solver initialization	8-DSKYIN
sparse	matrix solver numeric factorization	6-DSLEFA
direct input skyline	matrix solver numeric factorization	8-DSKYDF
skyline	matrix solver numeric factorization	8-DSKYFA
number estimation sparse	matrix solver numeric factorization and condition	6-DSLECO
sparse	matrix solver output control	6-DSLEOC
skyline	matrix solver output control	8-DSKYOC
restore sparse	matrix solver problem state from a savefile	6-DSLERS
restore skyline	matrix solver problem state from a savefile	8-DSKYRS
save sparse	matrix solver problem state to a savefile	6-DSLESV
save skyline	matrix solver problem state to a savefile	8-DSKYSV
skyline	matrix solver reordering	8-DSKYOR
accept user-provided skyline	matrix solver reordering	8-DSKYOU
sparse	matrix solver reordering and symbolic factorization	6-DSLEOR
retrieve sparse	matrix solver runtime statistics	6-DSLESR
retrieve skyline	matrix solver runtime statistics	8-DSKYSR
print sparse	matrix solver statistics	6-DSLEPS
print skyline	matrix solver statistics	8-DSKYPS
deallocate sparse	matrix solver working storage	6-DSLEDA
deallocate skyline	matrix solver working storage	8-DSKYDA
end of sparse	matrix structure input	6-DSLEIF
end of sparse	matrix structure input	7-DSEVIF
end of skyline	matrix structure input	8-DSKYIF
sparse	matrix structure input by column	6-DSLEIC
sparse	matrix structure input by column	7-DSEVIC
skyline	matrix structure input by column	8-DSKYIC
sparse	matrix structure input by finite element or clique	6-DSLEIE
sparse	matrix structure input by finite element or clique	7-DSEVIE
skyline	matrix structure input by finite element or clique	8-DSKYIE
sparse	matrix structure input by matrix	6-DSLEIM
sparse	matrix structure input by matrix	7-DSEVIM
skyline	matrix structure input by matrix	8-DSKYIM
sparse	matrix structure input by single entry	6-DSLEI1
sparse	matrix structure input by single entry	7-DSEVI1
skyline	matrix structure input by single entry	8-DSKYI1
skyline	matrix structure input by skyline matrix	8-DSKYIS
solve a direct input skyline	matrix system	8-DSKYDS
solve a skyline	matrix system using a factored skyline matrix	8-DSKYSL
solve a sparse	matrix system using a factored sparse matrix	6-DSLESL
reduction of a real symmetric	matrix to tridiagonal form	5-TRED1
reduction of a real symmetric	matrix to tridiagonal form	5-TRED2
sparse	matrix value input by column	6-DSLEVC
sparse	matrix value input by column	7-DSEVVC
skyline	matrix value input by column	8-DSKYVC
sparse	matrix value input by finite element or clique	6-DSLEVE
sparse	matrix value input by finite element or clique	7-DSEVVE
skyline	matrix value input by finite element or clique	8-DSKYVE
sparse	matrix value input by matrix	6-DSLEVM
sparse	matrix value input by matrix	7-DSEVVM
skyline	matrix value input by matrix	8-DSKYVM
sparse	matrix value input by single entry	6-DSLEV1

sparse	matrix value input by single entry	7-DSEVV1
skyline	matrix value input by single entry	8-DSKYV1
skyline	matrix value input by skyline matrix	8-DSKYVS
sparse	matrix value input to main diagonal	7-DSEVVD
general	matrix-matrix multiply	3-SGEMM
Strassen general	matrix-matrix multiply	3-SGEMMS
symmetric or Hermitian	matrix-matrix multiply	3-SSYMM
triangular	matrix-matrix multiply	3-STRMM
general	matrix-matrix product	3-SGEMM
Strassen general	matrix-matrix product	3-SGEMMS
symmetric or Hermitian	matrix-matrix product	3-SSYMM
triangular	matrix-matrix product	3-STRMM
band	matrix-vector multiply	3-SGBMV
general	matrix-vector multiply	3-SGEMV
symmetric or Hermitian band	matrix-vector multiply	3-SSBMV
symmetric or Hermitian packed band	matrix-vector multiply	3-SSPMV
symmetric or Hermitian	matrix-vector multiply	3-SSYMV
triangular band	matrix-vector multiply	3-STBMV
triangular packed band	matrix-vector multiply	3-STPMV
triangular	matrix-vector multiply	3-STRMV
band	matrix-vector product	3-SGBMV
general	matrix-vector product	3-SGEMV
symmetric or Hermitian band	matrix-vector product	3-SSBMV
symmetric or Hermitian packed band	matrix-vector product	3-SSPMV
symmetric or Hermitian	matrix-vector product	3-SSYMV
triangular band	matrix-vector product	3-STBMV
triangular packed band	matrix-vector product	3-STPMV
triangular	matrix-vector product	3-STRMV
	maximum absolute value of vector elements	2-SAMAX
index of first occurrence of a vector element of	maximum magnitude	2-ISAMAX
	maximum magnitude of vector elements	2-SAMAX
index of first occurrence of a vector element of	maximum value	2-ISMAX
	maximum value of vector elements	2-SMAX
	max-norm of a vector	2-SAMAX
	minimum absolute value of vector elements	2-SAMIN
index of first occurrence of a vector element of	minimum magnitude	2-ISAMIN
	minimum magnitude of vector elements	2-SAMIN
index of first occurrence of a vector element of	minimum value	2-ISMIN
	minimum value of vector elements	2-SMIN
apply a	modified Givens rotation matrix	2-SROTM
construct a	modified Givens rotation matrix	2-SROTMG
	move a vector	2-SCOPY
scalar	multiple of a sparse vector added to a vector	2-SAXPYI
scalar	multiple of a vector added to a vector	2-SAXPY
band matrix-vector	multiply	3-SGBMV
general matrix-matrix	multiply	3-SGEMM
Strassen general matrix-matrix	multiply	3-SGEMMS
general matrix-vector	multiply	3-SGEMV
symmetric or Hermitian band matrix-vector	multiply	3-SSBMV
symmetric or Hermitian packed band matrix-vector	multiply	3-SSPMV
symmetric or Hermitian matrix-matrix	multiply	3-SSYMM
symmetric or Hermitian matrix-vector	multiply	3-SSYMV
triangular band matrix-vector	multiply	3-STBMV
triangular packed band matrix-vector	multiply	3-STPMV
triangular matrix-matrix	multiply	3-STRMM
triangular matrix-vector	multiply	3-STRMV
	non-volatile dynamic storage allocation	12-NALLOC
	non-volatile dynamic storage deallocation	12-DALLOC
	non-volatile dynamic storage reallocation	12-RALLOC
infinity	norm of a vector	2-SAMAX
one	norm of a vector	2-SASUM
Euclidean	norm of a vector	2-SNRM2
square of Euclidean	norm of a vector	2-SNRSQ
convert floating-point	numbers from CONVEX format to IBM format	12-SC2IBM
convert floating-point	numbers from IBM format to CONVEX format	12-SIBM2C
sparse matrix solver	numeric factorization	6-DSLEFA
direct input skyline matrix solver	numeric factorization	8-DSKYDF
skyline matrix solver	numeric factorization	8-DSKYFA
sparse matrix solver	numeric factorization and condition number estimation	6-DSLECO
index of first	occurrence of a vector element = scalar	2-ISSVEQ
index of first	occurrence of a vector element ≥ scalar	2-ISSVGE
index of first	occurrence of a vector element > scalar	2-ISSVGT

index of first	occurrence of a vector element ≤ scalar	2-ISSVLE
index of first	occurrence of a vector element < scalar	2-ISSVLT
index of first	occurrence of a vector element ≠ scalar	2-ISSVNE
index of first	occurrence of a vector element of maximum magnitude	2-ISAMAX
index of first	occurrence of a vector element of maximum value	2-ISMAX
index of first	occurrence of a vector element of minimum magnitude	2-ISAMIN
index of first	occurrence of a vector element of minimum value	2-ISMIN
count	occurrences of vector elements = scalar	2-ISCTEQ
count	occurrences of vector elements ≥ scalar	2-ISCTGE
count	occurrences of vector elements > scalar	2-ISCTGT
count	occurrences of vector elements ≤ scalar	2-ISCTLE
count	occurrences of vector elements < scalar	2-ISCTLT
count	occurrences of vector elements ≠ scalar	2-ISCTNE
	one-call sparse eigenvalue solver	7-DSEVE1
	one-call usage skyline matrix solver	8-DSKYFS
	one-call usage skyline matrix solver	8-DSKYFX
	one-call usage sparse matrix solver	6-DSLEFS
the elements of a vector into ascending or descending	order sort	12-SSORT
solve a first	order linear recurrence	11-SFLR1
solve a first	order linear recurrence	11-SFLR2
solve for the last term of a first	order linear recurrence	11-SFLRL
solve a second	order linear recurrence	11-SSLR2
solve a second	order linear recurrence	11-SSLR3
solve for the last term of a second	order linear recurrence	11-SSLRL
solve a first	order linear recurrence with constant coefficient	11-SFLR1C
solve a first	order linear recurrence with constant coefficient	11-SFLR2C
sparse matrix solver	output control	6-DSLEOC
sparse eigenvalue solver	output control	7-DSEVOC
skyline matrix solver	output control	8-DSKYOC
triangular	packed band equation solver	3-STPSV
symmetric or Hermitian	packed band matrix-vector multiply	3-SSPMV
triangular	packed band matrix-vector multiply	3-STPMV
symmetric or Hermitian	packed band matrix-vector product	3-SSPMV
triangular	packed band matrix-vector product	3-STPMV
symmetric or Hermitian	packed rank-1 update	3-SSPR
symmetric or Hermitian	packed rank-2 update	3-SSPR2
compute the vector of	partial products of a vector	11-SPPROD
compute the vector of	partial sums of a vector	11-SPSUM
fractional	parts of vector elements	2-SFRAC
long	period scalar random number generator	12-SRAN
long	period vectorized number generator	12-SRANV
determinant of a	positive definite symmetric band matrix	4-SPBDI
factor a	positive definite symmetric band matrix	4-SPBFA
its condition number factor a	positive definite symmetric band matrix and estimate	4-SPBCO
factored matrix solve a	positive definite symmetric band system using a	4-SPBSL
determinant or inverse of a	positive definite symmetric matrix	4-SPODI
factor a	positive definite symmetric matrix	4-SPOFA
condition number factor a	positive definite symmetric matrix and estimate its	4-SPOCO
matrix solve a	positive definite symmetric system using a factored	4-SPOSL
equations solve a	positive definite tridiagonal system of linear	4-SPTSL
	print skyline matrix solver statistics	8-DSKYPS
	print sparse matrix solver statistics	6-DSLEPS
restore sparse matrix solver	problem state from a savefile	6-DSLERS
restore sparse eigenvalue solver	problem state from a savefile	7-DSEVRS
restore skyline matrix solver	problem state from a savefile	8-DSKYRS
save sparse matrix solver	problem state to a savefile	6-DSLESV
save sparse eigenvalue solver	problem state to a savefile	7-DSEVSV
save skyline matrix solver	problem state to a savefile	8-DSKYSV
band matrix-vector	product	3-SGBMV
general matrix-matrix	product	3-SGEMM
Strassen general matrix-matrix	product	3-SGEMMS
general matrix-vector	product	3-SGEMV
symmetric or Hermitian band matrix-vector	product	3-SSBMV
symmetric or Hermitian packed band matrix-vector	product	3-SSPMV
symmetric or Hermitian matrix-matrix	product	3-SSYMM
symmetric or Hermitian matrix-vector	product	3-SSYMV
triangular band matrix-vector	product	3-STBMV
triangular packed band matrix-vector	product	3-STPMV
triangular matrix-matrix	product	3-STRMM
triangular matrix-vector	product	3-STRMV
dot	product of two vectors	2-SDOT
inner	product of two vectors	2-SDOT

sparse dot	product of two vectors	2-SDOTI
sparse inner	product of two vectors	2-SDOTI
weighted dot	product of two vectors	2-SWDOT
weighted inner	product of two vectors	2-SWDOT
compute the vector of partial	products of a vector	11-SPPROD
construct a linear	ramp function	2-SRAMP
VAX-compatible scalar	random number generator	12-RAN
VAX-compatible vectorized	random number generator	12-RANV
long period scalar	random number generator	12-SRAN
clip vector elements to specified	range	2-SCLIP
clip vector elements to left-ended	range	2-SCLIPL
clip vector elements to right-ended	range	2-SCLIPR
general	rank-1 update	3-SGER
symmetric or Hermitian packed	rank-1 update	3-SSPR
symmetric or Hermitian	rank-1 update	3-SSYR
symmetric or Hermitian packed	rank-2 update	3-SSPR2
symmetric or Hermitian	rank-2 update	3-SSYR2
symmetric or Hermitian	rank-2k update	3-SSYR2K
symmetric or Hermitian	rank-k update	3-SSYRK
eigenvalues and eigenvectors of a	real symmetric matrix	5-RS
reduction of a	real symmetric matrix to tridiagonal form	5-TRED1
reduction of a	real symmetric matrix to tridiagonal form	5-TRED2
eigenvalues and eigenvectors of a	real symmetric tridiagonal matrix	5-TQL2
eigenvalues of a	real symmetric tridiagonal matrix	5-TQLRAT
non-volatile dynamic storage	reallocation	12-RALLOC
	real-to-complex 1-dimensional FFT	9-CRC1FT
simultaneous	real-to-complex 1-dimensional FFT	9-CRCFTS
	real-to-complex 2-dimensional FFT	9-CRC2FT
	real-to-complex 3-dimensional FFT	9-CRC3FT
	reciprocal scale a vector	2-SRSCL
solve a first order linear	recurrence	11-SFLR1
solve a first order linear	recurrence	11-SFLR2
solve for the last term of a first order linear	recurrence	11-SFLRL
solve a second order linear	recurrence	11-SSLR2
solve a second order linear	recurrence	11-SSLR3
solve for the last term of a second order linear	recurrence	11-SSLRL
solve a first order linear	recurrence with constant coefficient	11-SFLR1C
solve a first order linear	recurrence with constant coefficient	11-SFLR2C
form	reduction of a real symmetric matrix to tridiagonal	5-TRED1
form	reduction of a real symmetric matrix to tridiagonal	5-TRED2
skyline matrix solver	reordering	8-DSKYOR
accept user-provided skyline matrix solver	reordering	8-DSKYOU
sparse matrix solver	reordering and symbolic factorization	6-DSLEOR
sparse eigenvalue solver	reordering and symbolic factorization	7-DSEVOR
savefile	restore skyline matrix solver problem state from a	8-DSKYRS
savefile	restore sparse eigenvalue solver problem state from a	7-DSEVRS
savefile	restore sparse matrix solver problem state from a	6-DSLERS
check accuracy of sparse eigenvalue and eigenvector	results	7-DSEVCK
return sparse eigenvalue and eigenvector	results	7-DSEVRC
return sparse eigenvalue	results	7-DSEVRL
	retrieve skyline matrix solver runtime statistics	8-DSKYSR
	retrieve sparse matrix solver runtime statistics	6-DSLESR
	return sparse eigenvalue and eigenvector results	7-DSEVRC
	return sparse eigenvalue results	7-DSEVRL
clip vector elements to	right-ended range	2-SCLIPR
apply a Givens	rotation matrix	2-SROT
construct a Givens	rotation matrix	2-SROTG
apply a sparse Givens	rotation matrix	2-SROTI
apply a modified Givens	rotation matrix	2-SROTM
construct a modified Givens	rotation matrix	2-SROTMG
retrieve sparse matrix solver	runtime statistics	6-DSLESR
retrieve skyline matrix solver	runtime statistics	8-DSKYSR
savefile	save skyline matrix solver problem state to a	8-DSKYSV
savefile	save sparse eigenvalue solver problem state to a	7-DSEVSV
	save sparse matrix solver problem state to a savefile	6-DSLESV
restore sparse matrix solver problem state from a	savefile	6-DSLERS
save sparse matrix solver problem state to a	savefile	6-DSLESV
restore sparse eigenvalue solver problem state from a	savefile	7-DSEVRS
save sparse eigenvalue solver problem state to a	savefile	7-DSEVSV
restore skyline matrix solver problem state from a	savefile	8-DSKYRS
save skyline matrix solver problem state to a	savefile	8-DSKYSV
count occurrences of vector elements =	scalar	2-ISCTEQ

count occurrences of vector elements ≥	scalar	2-ISCTGE
count occurrences of vector elements >	scalar	2-ISCTGT
count occurrences of vector elements ≤	scalar	2-ISCTLE
count occurrences of vector elements <	scalar	2-ISCTLT
count occurrences of vector elements ≠	scalar	2-ISCTNE
index of first occurrence of a vector element =	scalar	2-ISSVEQ
index of first occurrence of a vector element ≥	scalar	2-ISSVGE
index of first occurrence of a vector element >	scalar	2-ISSVGT
index of first occurrence of a vector element ≤	scalar	2-ISSVLE
index of first occurrence of a vector element <	scalar	2-ISSVLT
index of first occurrence of a vector element ≠	scalar	2-ISSVNE
build an array of indices of vector elements =	scalar	2-SLSTEQ
build an array of indices of vector elements ≥	scalar	2-SLSTGE
build an array of indices of vector elements >	scalar	2-SLSTGT
build an array of indices of vector elements ≤	scalar	2-SLSTLE
build an array of indices of vector elements <	scalar	2-SLSTLT
build an array of indices of vector elements ≠	scalar	2-SLSTNE
vector divided by a	scalar	2-SRSCL
	scalar multiple of a sparse vector added to a vector	2-SAXPYI
	scalar multiple of a vector added to a vector	2-SAXPY
VAX-compatible	scalar random number generator	12-RAN
long period	scalar random number generator	12-SRAN
	scalar times a sparse vector added to a vector	2-SAXPYI
	scalar times a vector	2-SSCAL
	scalar times a vector added to a vector	2-SAXPY
reciprocal	scale a vector	2-SRSCL
	scale a vector	2-SSCAL
	scatter a sparse vector	2-SSCTR
solve a	second order linear recurrence	11-SSLR2
solve a	second order linear recurrence	11-SSLR3
solve for the last term of a	second order linear recurrence	11-SSLRL
	simultaneous complex 1-dimensional FFT	9-CFFTS
	simultaneous real-to-complex 1-dimensional FFT	9-CRCFTS
	simultaneous triangular equations solver	3-STRSM
sparse matrix structure input by	single entry	6-DSLEI1
sparse matrix value input by	single entry	6-DSLEV1
sparse matrix structure input by	single entry	7-DSEVI1
sparse matrix value input by	single entry	7-DSEVV1
skyline matrix structure input by	single entry	8-DSKYI1
skyline matrix value input by	single entry	8-DSKYV1
skyline matrix structure input by	skyline matrix	8-DSKYIS
solve a skyline matrix system using a factored	skyline matrix	8-DSKYSL
skyline matrix value input by	skyline matrix	8-DSKYVS
one-call usage	skyline matrix solver	8-DSKYFS
one-call usage	skyline matrix solver	8-DSKYFX
	skyline matrix solver initialization	8-DSKYIN
direct input	skyline matrix solver numeric factorization	8-DSKYDF
	skyline matrix solver numeric factorization	8-DSKYFA
	skyline matrix solver output control	8-DSKYOC
restore	skyline matrix solver problem state from a savefile	8-DSKYRS
save	skyline matrix solver problem state to a savefile	8-DSKYSV
	skyline matrix solver reordering	8-DSKYOR
accept user-provided	skyline matrix solver reordering	8-DSKYOU
retrieve	skyline matrix solver runtime statistics	8-DSKYSR
print	skyline matrix solver statistics	8-DSKYPS
deallocate	skyline matrix solver working storage	8-DSKYDA
end of	skyline matrix structure input	8-DSKYIF
	skyline matrix structure input by column	8-DSKYIC
clique	skyline matrix structure input by finite element or	8-DSKYIE
	skyline matrix structure input by matrix	8-DSKYIM
	skyline matrix structure input by single entry	8-DSKYI1
	skyline matrix structure input by skyline matrix	8-DSKYIS
solve a direct input	skyline matrix system	8-DSKYDS
solve a	skyline matrix system using a factored skyline matrix	8-DSKYSL
	skyline matrix value input by column	8-DSKYVC
clique	skyline matrix value input by finite element or	8-DSKYVE
	skyline matrix value input by matrix	8-DSKYVM
	skyline matrix value input by single entry	8-DSKYV1
	skyline matrix value input by skyline matrix	8-DSKYVS
	solve a direct input skyline matrix system	8-DSKYDS
	solve a first order linear recurrence	11-SFLR1
	solve a first order linear recurrence	11-SFLR2

coefficient	solve a first order linear recurrence with constant	11-SFLR1C
coefficient	solve a first order linear recurrence with constant	11-SFLR2C
	solve a general band system using a factored matrix	4-SGBSL
	solve a general system using a factored matrix	4-SGESL
equations	solve a general tridiagonal system of linear	4-SGTSL
equations	solve a general tridiagonal system of linear	4-SGTSV
a factored matrix	solve a positive definite symmetric band system using	4-SPBSL
factored matrix	solve a positive definite symmetric system using a	4-SPOSL
linear equations	solve a positive definite tridiagonal system of	4-SPTSL
	solve a second order linear recurrence	11-SSLR2
	solve a second order linear recurrence	11-SSLR3
skyline matrix	solve a skyline matrix system using a factored	8-DSKYSL
matrix	solve a sparse matrix system using a factored sparse	6-DSLESL
recurrence	solve for the last term of a first order linear	11-SFLRL
recurrence	solve for the last term of a second order linear	11-SSLRL
triangular band equation	solver	3-STBSV
triangular packed band equation	solver	3-STPSV
simultaneous triangular equations	solver	3-STRSM
triangular equation	solver	3-STRSV
one-call usage sparse matrix	solver	6-DSLEFS
one-call sparse eigenvalue	solver	7-DSEVE1
one-call usage skyline matrix	solver	8-DSKYFS
one-call usage skyline matrix	solver	8-DSKYFX
sparse matrix	solver initialization	6-DSLEIN
sparse eigenvalue	solver initialization	7-DSEVIN
skyline matrix	solver initialization	8-DSKYIN
sparse matrix	solver numeric factorization	6-DSLEFA
direct input skyline matrix	solver numeric factorization	8-DSKYDF
skyline matrix	solver numeric factorization	8-DSKYFA
estimation sparse matrix	solver numeric factorization and condition number	6-DSLECO
sparse matrix	solver output control	6-DSLEOC
sparse eigenvalue	solver output control	7-DSEVOC
skyline matrix	solver output control	8-DSKYOC
restore sparse matrix	solver problem state from a savefile	6-DSLERS
restore sparse eigenvalue	solver problem state from a savefile	7-DSEVRS
restore skyline matrix	solver problem state from a savefile	8-DSKYRS
save sparse matrix	solver problem state to a savefile	6-DSLESV
save sparse eigenvalue	solver problem state to a savefile	7-DSEVSV
save skyline matrix	solver problem state to a savefile	8-DSKYSV
skyline matrix	solver reordering	8-DSKYOR
accept user-provided skyline matrix	solver reordering	8-DSKYOU
sparse matrix	solver reordering and symbolic factorization	6-DSLEOR
sparse eigenvalue	solver reordering and symbolic factorization	7-DSEVOR
retrieve sparse matrix	solver runtime statistics	6-DSLESR
retrieve skyline matrix	solver runtime statistics	8-DSKYSR
print sparse matrix	solver statistics	6-DSLEPS
print skyline matrix	solver statistics	8-DSKYPS
deallocate sparse matrix	solver working storage	6-DSLEDA
deallocate sparse eigenvalue	solver working storage	7-DSEVDA
deallocate skyline matrix	solver working storage	8-DSKYDA
descending order	sort the elements of a vector into ascending or	12-SSORT
	sparse dot product of two vectors	2-SDOTI
check accuracy of	sparse eigenvalue and eigenvector results	7-DSEVCK
return	sparse eigenvalue and eigenvector results	7-DSEVRC
return	sparse eigenvalue results	7-DSEVRL
one-call	sparse eigenvalue solver	7-DSEVE1
	sparse eigenvalue solver initialization	7-DSEVIN
	sparse eigenvalue solver output control	7-DSEVOC
savefile restore	sparse eigenvalue solver problem state from a	7-DSEVRS
save	sparse eigenvalue solver problem state to a savefile	7-DSEVSV
factorization	sparse eigenvalue solver reordering and symbolic	7-DSEVOR
deallocate	sparse eigenvalue solver working storage	7-DSEVDA
extract	sparse eigenvalues and eigenvectors	7-DSEVES
extract	sparse eigenvalues and eigenvectors	7-DSEVEX
apply a	sparse Givens rotation matrix	2-SROTI
	sparse inner product of two vectors	2-SDOTI
solve a sparse matrix system using a factored	sparse matrix	6-DSLESL
one-call usage	sparse matrix solver	6-DSLEFS
	sparse matrix solver initialization	6-DSLEIN
	sparse matrix solver numeric factorization	6-DSLEFA
condition number estimation	sparse matrix solver numeric factorization and	6-DSLECO
	sparse matrix solver output control	6-DSLEOC

restore	sparse matrix solver problem state from a savefile	6-DSLERS
save	sparse matrix solver problem state to a savefile	6-DSLESV
factorization	sparse matrix solver reordering and symbolic	6-DSLEOR
retrieve	sparse matrix solver runtime statistics	6-DSLESR
print	sparse matrix solver statistics	6-DSLEPS
deallocate	sparse matrix solver working storage	6-DSLEDA
end of	sparse matrix structure input	6-DSLEIF
end of	sparse matrix structure input	7-DSEVIF
	sparse matrix structure input by column	6-DSLEIC
	sparse matrix structure input by column	7-DSEVIC
clique	sparse matrix structure input by finite element or	6-DSLEIE
clique	sparse matrix structure input by finite element or	7-DSEVIE
	sparse matrix structure input by matrix	6-DSLEIM
	sparse matrix structure input by matrix	7-DSEVIM
	sparse matrix structure input by single entry	6-DSLEI1
	sparse matrix structure input by single entry	7-DSEVI1
solve a	sparse matrix system using a factored sparse matrix	6-DSLESL
	sparse matrix value input by column	6-DSLEVC
	sparse matrix value input by column	7-DSEVVC
	sparse matrix value input by finite element or clique	6-DSLEVE
	sparse matrix value input by finite element or clique	7-DSEVVE
	sparse matrix value input by matrix	6-DSLEVM
	sparse matrix value input by matrix	7-DSEVVM
	sparse matrix value input by single entry	6-DSLEV1
	sparse matrix value input by single entry	7-DSEVV1
	sparse matrix value input to main diagonal	7-DSEVVD
gather a	sparse vector	2-SGTHR
gather and zero a	sparse vector	2-SGTHRZ
scatter a	sparse vector	2-SSCTR
scalar multiple of a	sparse vector added to a vector	2-SAXPYI
scalar times a	sparse vector added to a vector	2-SAXPYI
	square of 2-norm of a vector	2-SNRSQ
	square of Euclidean norm of a vector	2-SNRSQ
print sparse matrix solver	statistics	6-DSLEPS
retrieve sparse matrix solver runtime	statistics	6-DSLESR
print skyline matrix solver	statistics	8-DSKYPS
retrieve skyline matrix solver runtime	statistics	8-DSKYSR
deallocate sparse matrix solver working	storage	6-DSLEDA
deallocate sparse eigenvalue solver working	storage	7-DSEVDA
deallocate skyline matrix solver working	storage	8-DSKYDA
volatile dynamic	storage allocation	12-DYNAMIC
volatile dynamic	storage allocation	12-MALLOC
non-volatile dynamic	storage allocation	12-NALLOC
non-volatile dynamic	storage deallocation	12-DALLOC
non-volatile dynamic	storage reallocation	12-RALLOC
	Strassen general matrix-matrix multiply	3-SGEMMS
	Strassen general matrix-matrix product	3-SGEMMS
end of sparse matrix	structure input	6-DSLEIF
end of sparse matrix	structure input	7-DSEVIF
end of skyline matrix	structure input	8-DSKYIF
sparse matrix	structure input by column	6-DSLEIC
sparse matrix	structure input by column	7-DSEVIC
skyline matrix	structure input by column	8-DSKYIC
sparse matrix	structure input by finite element or clique	6-DSLEIE
sparse matrix	structure input by finite element or clique	7-DSEVIE
skyline matrix	structure input by finite element or clique	8-DSKYIE
sparse matrix	structure input by matrix	6-DSLEIM
sparse matrix	structure input by matrix	7-DSEVIM
skyline matrix	structure input by matrix	8-DSKYIM
sparse matrix	structure input by single entry	6-DSLEI1
sparse matrix	structure input by single entry	7-DSEVI1
skyline matrix	structure input by single entry	8-DSKYI1
skyline matrix	structure input by skyline matrix	8-DSKYIS
	sum of absolute values of vector elements	2-SASUM
	sum of magnitudes of vector elements	2-SASUM
	sum of vector elements	2-SSUM
compute the vector of partial	sums of a vector	11-SPSUM
	swap two vectors	2-SSWAP
sparse matrix solver reordering and	symbolic factorization	6-DSLEOR
sparse eigenvalue solver reordering and	symbolic factorization	7-DSEVOR
determinant of a positive definite	symmetric band matrix	4-SPBDI
factor a positive definite	symmetric band matrix	4-SPBFA

number factor a positive definite	symmetric band matrix and estimate its condition . . .	4-SPBCO
solve a positive definite	symmetric band system using a factored matrix	4-SPBSL
determinant or inverse of a positive definite	symmetric matrix	4-SPODI
factor a positive definite	symmetric matrix	4-SPOFA
eigenvalues and eigenvectors of a real	symmetric matrix	5-RS
factor a positive definite	symmetric matrix and estimate its condition number . .	4-SPOCO
reduction of a real	symmetric matrix to tridiagonal form	5-TRED1
reduction of a real	symmetric matrix to tridiagonal form	5-TRED2
	symmetric or Hermitian band matrix-vector multiply . .	3-SSBMV
	symmetric or Hermitian band matrix-vector product . .	3-SSBMV
	symmetric or Hermitian matrix-matrix multiply	3-SSYMM
	symmetric or Hermitian matrix-matrix product	3-SSYMM
	symmetric or Hermitian matrix-vector multiply	3-SSYMV
	symmetric or Hermitian matrix-vector product	3-SSYMV
multiply	symmetric or Hermitian packed band matrix-vector . . .	3-SSPMV
product	symmetric or Hermitian packed band matrix-vector . . .	3-SSPMV
	symmetric or Hermitian packed rank-1 update	3-SSPR
	symmetric or Hermitian packed rank-2 update	3-SSPR2
	symmetric or Hermitian rank-1 update	3-SSYR
	symmetric or Hermitian rank-2 update	3-SSYR2
	symmetric or Hermitian rank-2k update	3-SSYR2K
	symmetric or Hermitian rank-k update	3-SSYRK
solve a positive definite	symmetric system using a factored matrix	4-SPOSL
eigenvalues and eigenvectors of a real	symmetric tridiagonal matrix	5-TQL2
eigenvalues of a real	symmetric tridiagonal matrix	5-TQLRAT
solve a direct input skyline matrix	system	8-DSKYDS
solve a general tridiagonal	system of linear equations	4-SGTSL
solve a general tridiagonal	system of linear equations	4-SGTSV
solve a positive definite tridiagonal	system of linear equations	4-SPTSL
solve a general band	system using a factored matrix	4-SGBSL
solve a general	system using a factored matrix	4-SGESL
solve a positive definite symmetric band	system using a factored matrix	4-SPBSL
solve a positive definite symmetric	system using a factored matrix	4-SPOSL
solve a skyline matrix	system using a factored skyline matrix	8-DSKYSL
solve a sparse matrix	system using a factored sparse matrix	6-DSLESL
solve for the last	term of a first order linear recurrence	11-SFLRL
solve for the last	term of a second order linear recurrence	11-SSLRL
scalar	times a sparse vector added to a vector	2-SAXPYI
scalar	times a vector	2-SSCAL
scalar	times a vector added to a vector	2-SAXPY
	triangular band equation solver	3-STBSV
	triangular band matrix-vector multiply	3-STBMV
	triangular band matrix-vector product	3-STBMV
	triangular equation solver	3-STRSV
simultaneous	triangular equations solver	3-STRSM
	triangular matrix-matrix multiply	3-STRMM
	triangular matrix-matrix product	3-STRMM
	triangular matrix-vector multiply	3-STRMV
	triangular matrix-vector product	3-STRMV
	triangular packed band equation solver	3-STPSV
	triangular packed band matrix-vector multiply	3-STPMV
	triangular packed band matrix-vector product	3-STPMV
reduction of a real symmetric matrix to	tridiagonal form	5-TRED1
reduction of a real symmetric matrix to	tridiagonal form	5-TRED2
eigenvalues and eigenvectors of a real symmetric	tridiagonal matrix	5-TQL2
eigenvalues of a real symmetric	tridiagonal matrix	5-TQLRAT
solve a general	tridiagonal system of linear equations	4-SGTSL
solve a general	tridiagonal system of linear equations	4-SGTSV
solve a positive definite	tridiagonal system of linear equations	4-SPTSL
general rank-1	update	3-SGER
symmetric or Hermitian packed rank-1	update	3-SSPR
symmetric or Hermitian packed rank-2	update	3-SSPR2
symmetric or Hermitian rank-1	update	3-SSYR
symmetric or Hermitian rank-2	update	3-SSYR2
symmetric or Hermitian rank-2k	update	3-SSYR2K
symmetric or Hermitian rank-k	update	3-SSYRK
one-call	usage skyline matrix solver	8-DSKYFS
one-call	usage skyline matrix solver	8-DSKYFX
one-call	usage sparse matrix solver	6-DSLEFS
accept	user-provided skyline matrix solver reordering	8-DSKYOU
of first occurrence of a vector element of maximum	value index	2-ISMAX
of first occurrence of a vector element of minimum	value index	2-ISMIN

sparse matrix	value input by column	6-DSLEVC
sparse matrix	value input by column	7-DSEVVC
skyline matrix	value input by column	8-DSKYVC
sparse matrix	value input by finite element or clique	6-DSLEVE
sparse matrix	value input by finite element or clique	7-DSEVVE
skyline matrix	value input by finite element or clique	8-DSKYVE
sparse matrix	value input by matrix	6-DSLEVM
sparse matrix	value input by matrix	7-DSEVVM
skyline matrix	value input by matrix	8-DSKYVM
sparse matrix	value input by single entry	6-DSLEV1
sparse matrix	value input by single entry	7-DSEVV1
skyline matrix	value input by single entry	8-DSKYV1
skyline matrix	value input by skyline matrix	8-DSKYVS
sparse matrix	value input to main diagonal	7-DSEVVD
maximum absolute	value of vector elements	2-SAMAX
minimum absolute	value of vector elements	2-SAMIN
maximum	value of vector elements	2-SMAX
minimum	value of vector elements	2-SMIN
sum of absolute	values of vector elements	2-SASUM
	VAX-compatible scalar random number generator	12-RAN
	VAX-compatible vectorized random number generator	12-RANV
	VECLIB error handler	12-XERVEC
long period	vectorized number generator	12-SRANV
VAX-compatible	vectorized random number generator	12-RANV
	volatile dynamic storage allocation	12-DYNAMIC
	volatile dynamic storage allocation	12-MALLOC
	weighted dot product of two vectors	2-SWDOT
	weighted inner product of two vectors	2-SWDOT
deallocate sparse matrix solver	working storage	6-DSLEDA
deallocate sparse eigenvalue solver	working storage	7-DSEVDA
deallocate skyline matrix solver	working storage	8-DSKYDA
clear a vector to	zero	2-SZERO
initialize a vector to	zero	2-SZERO
gather and	zero a sparse vector	2-SGTHRZ

Preface

Purpose and Audience

This guide describes the CONVEX VECLIB software library and shows you how to use it. CONVEX VECLIB is a collection of FORTRAN-callable subprograms optimized for use on the CONVEX family of supercomputers. This library provides mathematical software and computational kernels for application programs.

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

- convert, develop, or optimize programs for use on CONVEX supercomputers
- optimize existing software to improve performance and increase productivity on CONVEX supercomputers

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

Organization

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

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

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

This guide is organized into the following chapters and appendices:

- Chapter 1 introduces general concepts about CONVEX 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 allocate dynamic memory, produce random numbers, convert numbers stored in CONVEX floating-point format to or from IBM floating-point format, and sort the elements of a vector in ascending or descending order.
- Appendix A describes how to call VECLIB subprograms from within C programs.
- Appendix B describes how to call VECLIB subprograms from within Ada programs.
- An index is included at the back of the manual.

Notational Conventions

The following conventions are used in this manual:

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

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

  fc *prog_name*.**o**

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

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

- Type in **lowercase boldface** indicates FORTRAN generic variable or array names. You should substitute actual variable or array names. The *italicized* mathematical entities and the **lowercase boldface** variable and array names usually correspond. For example, A will be a matrix and **a** will be the FORTRAN array containing the matrix:

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

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

  fc *prog_name*.**o**

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

- `UPPERCASE CONSTANT WIDTH` represents 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 by CONVEX Computer Corporation to help you:

- *CONVEX VECLIB Quick Reference* (DSW-134). This compact reference lists the name, purpose and usage for each VECLIB subprogram. Its organization is similar to the *CONVEX VECLIB User's Guide.*
- *CONVEX SCILIB User's Guide* (DSW-360). This guide provides information on the subprograms and functions provided with the CONVEX SCILIB library.
- *CONVEX SCILIB Quick Reference* (DSW-361). This compact reference lists the name, purpose and usage for each SCILIB subprogram. Its organization is similar to the *CONVEX SCILIB User's Guide.*
- *CONVEX LAPACK User's Guide* (DSW-036). This guide provides information on the subprograms provided with the CONVEX LAPACK library.
- *CONVEX FORTRAN Language Reference Manual* (DSW-037). This manual is a reference for the CONVEX FORTRAN programming language and is designed to provide a thorough working definition of the language.
- *CONVEX FORTRAN User's Guide* (DSW-038). This guide tells you how to use the CONVEX FORTRAN compiler, including how to compile, load, and execute programs.
- *CONVEX FORTRAN Optimization Guide* (DSW-034). This guide describes methods for optimizing FORTRAN programs.
- *CONVEX Performance Analyzer (CXpa) User's Guide* (DSW-251). This guide explains the operation of the CONVEX Performance Analyzer (CXpa) and the steps needed to create and interpret a CXpa profile.
- *CONVEX Application Compiler User's Guide* (DSW-401). This guide describes the CONVEX Application Compiler and how to use it to optimize programs.
- *CONVEX Interlanguage Programming Guide* (DSW-043). This guide explains how to call procedures written in one language from a program written in another language. The languages covered are FORTRAN, C, C++, and Ada.
- *CONVEX C Guide* (DSW-086). This guide describes the CONVEX C compiler.
- *CONVEX Ada User's Guide* (DSW-147). This guide describes the Ada compiler and support tools and reviews basic Ada concepts.
- *CONVEX Consultant User's Guide* (DSW-025). This guide describes the functions and operations of the CONVEX *csd* debugger, the post-mortem dump (*pmd*) facility, and the *prof, bprof,* and *gprof* profilers.
- *ConvexOS Man Pages for Users* (DSW-331). This book contains copies of Sections 1 and 7 of the online man pages. These man pages are primarily concerned with operating system information for users.

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

- Garbow, B.S., et al. 1977. "Matrix Eigensystem Routines—EISPACK Guide Extension." *Lecture Notes in Computer Science,* Vol. 51. Springer-Verlag, New York. This guide provides information on non-optimized EISPACK subprograms included in CONVEX VECLIB, but not documented in the *CONVEX VECLIB User's Guide.*

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

Ordering Documentation

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

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

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

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

Technical Assistance

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

- Within the continental U.S. call 1(800)952-0379.

- Outside the continental U.S. contact the local CONVEX office.

1

Introduction to CONVEX VECLIB

Overview

CONVEX VECLIB is a collection of FORTRAN-callable subprograms optimized for use on the CONVEX family of supercomputers, providing mathematical software and computational kernels for application programs involving arrays. These libraries contain 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 manipulating dynamic memory, sorting and generating random numbers

Although CONVEX VECLIB was designed for use with FORTRAN programs, C and Ada programs can call CONVEX VECLIB subprograms, as described in Appendices A and B, respectively.

This chapter provides information necessary for efficient use of CONVEX VECLIB, including discussions of conformance to various public-domain standards, the VECLIB and VECLIB8 library files, accessing CONVEX VECLIB subprograms, optimizations, including parallel processing and interactions with other CONVEX 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, and CONVEX support services.

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

What You Need to Know to Use CONVEX VECLIB

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

Standardization

CONVEX VECLIB conforms to a variety of existing standards. For example, VECLIB 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 CONVEX supercomputers. Note, however, that even though the user interface is standardized, internal workings of VECLIB subprograms have been specialized for CONVEX supercomputers.

Two CONVEX 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 CONVEX FORTRAN Compiler provides several compilation options, namely, **–cfc**, **–p8**, and **–pd8**, that affect the size of FORTRAN data types. For compatibility with these compiler options, CONVEX 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 any of the **–cfc**, **–p8**, or **–pd8** compiler options.

- The VECLIB8 library is a subset of the VECLIB library that is designed for use by programs that are compiled with one of the **–cfc**, **–p8**, or **–pd8** compiler options.

For more information about these options, refer to the *CONVEX FORTRAN Language Reference Manual* 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 following chapters.

Accessing CONVEX VECLIB

The VECLIB and VECLIB8 libraries are 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, by using the –l option on the *fc* command line, as follows:

fc [*options*] *file* **–lveclib**

or

fc [*options*] *file* **–lveclib8**

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

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

fc [*options*] *file* **–lveclib –llapack**

or

fc [*options*] *file* **–lveclib8 –llapack8**

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

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

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

Add the linker option **–llapack8** if CONVEX LAPACK is also used. See "Interactions Between VECLIB, SCILIB, and LAPACK" for details about how to order the –l options. Do not try to use subprograms from both **–lveclib** and **–lscilib**, or both **–llapack** and **–lscilib**, in the same program.

Interactions Between VECLIB, SCILIB, and LAPACK

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

Differences between VECLIB and LAPACK

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

Differences between VECLIB8 and LAPACK8

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

Differences between VECLIB8 and SCILIB

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

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

Differences between LAPACK8 and SCILIB

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

Performance Value

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

Optimization

CONVEX VECLIB is highly efficient. It takes full advantage of the CONVEX tightly integrated vector, scalar, and parallel 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 the CONVEX architecture or assembly-language programming. In this respect, VECLIB is an extension of the FORTRAN language.

Parallel Processing

Parallel processing is available on CONVEX C2 and C3 Series computers that have multiple processors. These systems can divide a single computational process into small streams of execution, called *threads*. The result is that you can have more than one processor executing on behalf of the same process.

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

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

If you are computing on a CONVEX C2 or C3 Series system with multiple processors, you can realize the performance benefits of parallel processing in two ways. First, you can simply call any parallelized CONVEX VECLIB subprogram and let it employ parallelism internally.

Alternatively, you can call CONVEX VECLIB subprograms in a parallelized loop or region. To obtain parallelism via the second mechanism, you need to be familiar with the concept of reentrancy and with the FORCE_PARALLEL, BEGIN_TASKS, NEXT_TASK, and END_TASKS compiler directives.

CONVEX VECLIB subprograms are reentrant, which means that they may be called several times in parallel to do independent computations without one call interfering with another. You can use this feature to call CONVEX 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 a FORCE_PARALLEL compiler directive before the loop. For example, the following code makes parallel calls to VECLIB subprogram **SAXPY**:

```
C$DIR FORCE_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 CONVEX VECLIB subprogram to perform independent operations, but where the call statements are not in a loop. The FORTRAN compiler does not automatically parallelize code outside a loop, but you can use the BEGIN_TASKS, NEXT_TASK, and END_TASKS compiler directives to tell the compiler to parallelize such code. For example, the following code makes parallel calls to subprogram SNRM2:

```
C$DIR BEGIN_TASKS
      XNORM = SNRM2 (NX, X, 1)
C$DIR NEXT_TASK
      YNORM = SNRM2 (NY, Y, 1)
C$DIR END_TASKS
```

For more information on compiler directives, including usage cautions and warnings, refer to the *CONVEX FORTRAN User's Guide* and the *CONVEX FORTRAN Optimization Guide*.

Profiling CONVEX VECLIB Applications

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

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

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

- the number of times each subprogram is called

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

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

Optimizing with the Application Compiler

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

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

Floating Point Formats

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

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

Roundoff Effects

CONVEX 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 without and with 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 CONVEX 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 CONVEX VECLIB subprograms is a coded specification of its function, within the limits of standard Fortran 77 6-character names. Usually, the first character of a subprogram name, denoted by "T," shows the predominant data type according to Table 1-1:

Table 1-1: VECLIB Naming Convention—Data Type

T	Data Type	VECLIB	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 the VECLIB8 libraries, is shown in Table 1-2:

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

T	VECLIB Argument Lengths			VECLIB8 Argument Lengths		
	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 CONVEX 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, the CONVEX FORTRAN Compiler provides several compilation options, namely, **–cfc**, **–p8**, and **–pd8**, 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 CONVEX 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 CONVEX 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 FORTRAN preprocessor (see the *CONVEX FORTRAN User's Guide*) **#define** statement may be an appropriate conversion tool. Or, for example, the *fc* command line options **–fpp –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 Compiler Option			
	none	–cfc	–p8	–pd8
INTEGER	4	8	8	8
INTEGER*4	4	8	4	4
INTEGER*8	8	8	8	8
Integer by default	4	8	8	8
REAL	4	8	8	8
REAL*4	4	8	4	4
REAL*8	8	8	8	8
Real by default	4	8	8	8
DOUBLE PRECISION	8	16	16	8
Double Precision constant	8	16	16	8
COMPLEX	8	16	16	16
COMPLEX*8	8	16	8	8
COMPLEX*16	16	16	16	16
Complex constant	8	16	16	16
DOUBLE COMPLEX	16	16	16	16
Double Complex constant	16	16	16	16
LOGICAL	4	8	8	8
LOGICAL*4	4	8	4	4
LOGICAL*8	8	8	8	8
Logical constant	4	8	8	8

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 none of the compiler options, **–cfc**, **–p8**, or **–pd8**, are 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 *CONVEX FORTRAN Language Reference Manual.*

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 ≤ 0. This is considered mathematically correct because, by definition, an empty sum is zero. Because SDOT conforms to the published Basic Linear Algebra Subprograms (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 ≤ 0 may eliminate an IF statement that would take the same branch almost all the times 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

A high-level subprogram, when it detects an error, 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 CONVEX 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 the main program is in FORTRAN, a call traceback is also written onto the standard error file. Execution is then terminated with a nonzero exit status. You may supply a version of XERBLA or 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.

CONVEX VECLIB Programmer's Reference

The *CONVEX VECLIB Programmer's Reference* is online documentation that includes information from the *CONVEX VECLIB User's Guide*. This reference contains an introduction to CONVEX VECLIB and to each set of subprograms in CONVEX 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 *CONVEX VECLIB Programmer's Reference* is not a substitute for the *CONVEX VECLIB User's Guide*; detailed information on CONVEX VECLIB is in the user's guide.

To access reference entries, use the ConvexOS command

man 3b *entry_name*

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

man 3b veclib

Support Services

CONVEX maintains a staff to provide technical help if you have difficulty. Located in the CONVEX Technical Assistance Center (TAC), these people are the primary link between you and the company, and they can assist you with any difficulties. Before contacting the TAC about a CONVEX VECLIB problem, follow this procedure to find the cause of the trouble:

- Check any error response provided by the subprogram. 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 process above. If the trouble persists, contact the TAC. Provide a small test program and expected answers to help the TAC analyze the problem. To report a software or documentation problem to the TAC, use the *contact* utility. The *contact* utility allows you to submit a problem or suggest an enhancement directly to the TAC from your own system.

For information about *contact*, use the ConvexOS command

man contact

Suggestions

You are encouraged to use the *contact* utility to suggest capabilities you would like included in future releases of VECLIB. Using *contact* helps maintain an orderly record of customer requests and facilitates timely responses. Please submit a *contact* report with a priority of 6. Your suggestion will be routed to the CONVEX Mathematical Software Group for review.

Supplemental Reading

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

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

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

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

2

Basic Vector Operations

Overview

This chapter explains how to use the 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 BLA

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

Chapter Objectives

After reading this chapter you will:

- understand BLAS storage conventions
-
- know how to handle backward storage
- know how to use increment (also called stride) arguments

What You Need to Know to Use These Subprograms

This section discusses commonly used or computationally expensive operations of linear algebra. Even though you can code most of these operations in fewer than 10 lines of FORTRAN, using 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 CONVEX 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 × N2. Then A(I,J) is associated with the same memory location as B(K) where

```
K = I + (J-1) × N1.
```

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

FORTRAN Array Argument Association

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

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

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

BLAS Indexing Conventions

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

A vector in the BLAS is defined by three quantities:

1. The vector length.
2. The array or starting element within an array.
3. The increment, sometimes called the ***stride,*** which defines the number of storage units between successive vector elements.

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

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

INCX = 0 is 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 I is (N,B(I,1),LDB).

Examples

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

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

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

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

```
REAL*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)
```

Supplemental Reading

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.

Subprogram Descriptions

Index of the Element of a Vector of Maximum Magnitude
ISAMAX, IDAMAX, IIAMAX, ICAMAX, IZAMAX 2-7

Index of the Element of a Vector of Minimum Magnitude
ISAMIN, IDAMIN, IIAMIN, ICAMIN, IZAMIN 2-9

Count the Number of Occurrences of Selected Elements of a Vector
ISCTxx, IDCTxx, IICTxx, ICCTxx, IZCTxx (xx = EQ, GE, GT, LE, LT, or NE) 2-11

Index of the Maximum Element of a Vector
ISMAX, IDMAX, IIMAX 2-13

Index of the Minimum Element of a Vector
ISMIN, IDMIN, IIMIN 2-15

Search a Vector for a Specified Element
ISSVxx, IDSVxx, IISVxx, ICSVxx, IZSVxx (xx = EQ, GE, GT, LE, LT, or NE) 2-17

Maximum of Magnitudes of the Elements of a Vector
SAMAX, DAMAX, IAMAX, SCAMAX, DZAMAX 2-19

Minimum of Magnitudes of the Elements of a Vector
SAMIN, DAMIN, IAMIN, SCAMIN, DZAMIN 2-21

Sum of Magnitudes of the Elements of a Vector
SASUM, DASUM, IASUM, SCASUM, DZASUM 2-23

Elementary Vector Operation
SAXPY, DAXPY, CAXPY, ZAXPY, CAXPYC, ZAXPYC 2-25

Sparse Elementary Vector Operation
SAXPYI, DAXPYI, CAXPYI, ZAXPYI 2-28

Two-Sided Vector Clip
SCLIP, DCLIP, ICLIP 2-30

Left-Sided Vector Clip
SCLIPL, DCLIPL, ICLIPL 2-32

Right-Sided Vector Clip
SCLIPR, DCLIPR, ICLIPR 2-34

Copy Vector
SCOPY, DCOPY, ICOPY, CCOPY, CCOPYC, ZCOPY, ZCOPYC 2-36

Dot Product of Two Vectors
SDOT, DDOT, CDOTC, CDOTU, ZDOTC, ZDOTU 2-39

Sparse Dot Product of Two Vectors
SDOTI, DDOTI, CDOTCI, CDOTUI, ZDOTCI, ZDOTUI 2-42

Extract Fractional Parts of the Elements of a Vector
SFRAC, DFRAC 2-44

Gather a Sparse Vector
SGTHR, DGTHR, IGTHR, CGTHR, ZGTHR 2-46

Gather and Zero a Sparse Vector
SGTHRZ, DGTHRZ, IGTHRZ, CGTHRZ, ZGTHRZ 2-48

Build a List of Indices of Selected Vector Elements
SLSTxx, DLSTxx, ILSTxx, CLSTxx, ZLSTxx (xx = EQ, GE, GT, LE, LT, or NE) 2-50

Value of the Maximum Element of a Vector
SMAX, DMAX, IMAX 2-53

Value of the Minimum Element of a Vector
SMIN, DMIN, IMIN 2-55

Euclidean Norm of a Vector
SNRM2, DNRM2, SCNRM2, DZNRM2 2-57

Square of the Euclidean Norm of a Vector
SNRSQ, DNRSQ, SCNRSQ, DZNRSQ 2-59

Generate a Linear Ramp
SRAMP, DRAMP, IRAMP 2-61

Apply a Givens Rotation to Two Vectors
SROT, DROT, CROT, CSROT, ZROT, ZDROT 2-63

Construct a Givens Rotation
SROTG, DROTG, CROTG, ZROTG 2-66

Apply a Sparse Givens Rotation to Two Vectors
SROTI, DROTI 2-68

Apply a Modified Givens Rotation
SROTM, DROTM 2-70

Construct a Modified Givens Rotation
SROTMG, DROTMG 2-72

Scale a Vector
SRSCL, DRSCL, CRSCL, CSRSCL, ZRSCL, ZDRSCL 2-74

Scale a Vector
SSCAL, DSCAL, CSCAL, CSSCAL, CSCALC, ZSCAL, ZDSCAL, ZSCALC 2-76

Scatter a Sparse Vector
SSCTR, DSCTR, ISCTR, CSCTR, ZSCTR 2-78

Sum of the Elements of a Vector
SSUM, DSUM, ISUM, CSUM, ZSUM 2-80

Swap Two Vectors
SSWAP, DSWAP, ISWAP, CSWAP, ZSWAP 2-82

Weighted Dot Product of Two Vectors
SWDOT, DWDOT, CWDOTC, CWDOTU, ZWDOTC, ZWDOTU 2-84

Zero Vector
SZERO, DZERO, IZERO, CZERO, ZZERO 2-87

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

Purpose

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

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

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

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

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

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

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

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

Usage

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, IIAMAX, n, incx, x(lenx)
i = IIAMAX (n, x, incx)

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

Input

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

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

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

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

Output

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

Notes

The handling of **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)
```

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

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

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

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

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

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

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

This definition is not used because of computational speed.

The vector 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, IIAMIN, n, incx, x(lenx)
i = IIAMIN (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** ≤ 0, the subprograms do not reference **x**.

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

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

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

Output

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

Notes

The handling of **incx** < 0 differs between ISAMIN in 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)
```

Count Selected Vector Elements ISCTxx/IDCTxx/IICTxx/.../IZCTxx

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** ≤ 0, the subprograms do not reference **x**.

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

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

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

a The scalar a.

Output

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

FORTRAN Equivalent

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

Example

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

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

Purpose

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

$$x_i = \max\left\{x_j : j = 1, 2, ..., n\right\}.$$

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

Usage

VECLIB:

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

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

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

VECLIB8:

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

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

Input

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

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

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

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

Output

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

Notes

The handling of **incx** < 0 differs between ISMAX in 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)
```

Index of Minimum Element of Vector — ISMIN/IDMIN/IIMIN

Purpose

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

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

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

Usage

VECLIB:

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

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

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

VECLIB8:

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

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

Input

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

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

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

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

Output

i — If **n** ≤ 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)
```

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

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

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

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

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)×|**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)
```

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

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

$$s = \|x\|_\infty = \max\left\{|x_i| : i = 1, 2, \ldots, n\right\}.$$

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

$$s = \max\left\{|Re(x_i)| + |Im(x_i)| : i = 1, 2, \ldots, n\right\}$$

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

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

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)
```

SAMAX/DAMAX/IAMAX/SCAMAX/DZAMAX **Continued**

Input

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

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

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

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

Output

s If **n** ≤ 0, then **s** = 0. Otherwise, **s** is the 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)
```

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

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

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

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

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

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

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

s is computed instead of t since it is faster because it does not require any square roots. Since $t \le s \le \sqrt{2}t$, 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** ≤ 0, the subprograms do not reference **x**.

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

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

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

Output

s If **n** ≤ 0, then **s** = ∞, 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)
```

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

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 $t \le s \le \sqrt{2}t$, 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 **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x.

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

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

Output

s If **n** $\leq$ 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)
```

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** ≤ 0, the subprograms do not reference **x** or **y**.

a The scalar a.

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

incx Increment for the array **x**:

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

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

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

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

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

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

Output

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

Notes

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

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

FORTRAN Equivalent

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

Example 1 Compute the REAL*8 elementary vector operation

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

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

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

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

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

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

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

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

a The scalar a.

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

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

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

and

$$\mathbf{indx}(i) \ne \mathbf{indx}(j), \quad 1 \le i \ne j \le \mathbf{m},$$

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

	y	Array containing the elements of y, $\mathbf{y}(i) = y_i$.
Output	**y**	If $\mathbf{m} \le 0$ or $\mathbf{a} = 0$, then **y** is unchanged. Otherwise, $ax + y$ overwrites the input. Only the elements of **y** whose indices are included in **indx** are changed.

Notes

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

FORTRAN Equivalent

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

Example

Compute the REAL*8 elementary vector operation

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

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

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

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

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

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** ≤ 0, the subprograms do not reference **x** or **y**.

a The scalar a.

b The scalar b.

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

incx Increment for the array **x**:

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

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

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

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

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

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

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

Output

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

Notes

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

FORTRAN Equivalent

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

Example

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

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

Purpose

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

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

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** ≤ 0, the subprograms do not reference **x** or **y**.

a The scalar a.

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

incx Increment for the array **x**:

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

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

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

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

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

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

Output

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

Notes

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

FORTRAN Equivalent

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

Example

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

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

Purpose

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

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

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

Usage

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

incx Increment for the array **x**:

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

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

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

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

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

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

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

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

FORTRAN Equivalent

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

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

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

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)
```

Continued SCOPY/DCOPY/ICOPY/CCOPY/CCOPYC/.../ZCOPYC

Input

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

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

incx Increment for the array **x**:

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

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

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

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

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

Output

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

Notes

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

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

FORTRAN Equivalent

```
      SUBROUTINE SCOPY (N, X,INCX, Y,INCY)
      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
```

SCOPY/DCOPY/ICOPY/CCOPY/CCOPYC/.../ZCOPYC Continued

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)
```

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

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** ≤ 0, the subprograms do not reference **x** or **y**.

x — Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x. x is used in conjugated form by CDOTC and ZDOTC, and in unconjugated form by the other subprograms. Refer to "Purpose."

incx Increment for the array **x**:

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

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

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

incy Increment for the array **y**:

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

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

Output

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

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

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

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

Notes

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

FORTRAN Equivalent

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

Example 1 Compute the REAL*8 dot product

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

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

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

Example 2 Compute the REAL*8 dot product

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

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

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

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 $\{k_1, k_2, ..., k_m\}$ be the indices of these elements. (While some interesting elements of x may be zero, all *uninteresting* elements are assumed to be zero.) Let y be an ordinary n-vector. If x is represented by arrays **x** and **indx** such that **indx**(i) = k_i and **x**(i) = x_{k_i} , then these subprograms compute

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

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, $\mathbf{m} \le \mathbf{n}$. If $\mathbf{m} \le 0$, 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

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

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

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

Output

s The resulting value of the dot product. If $\mathbf{m} \le 0$, then $\mathbf{s} = 0$. Otherwise, $s = \sum_{i=1}^{m} \mathbf{x}(i) \times \mathbf{y}(\mathbf{indx}(i))$ unless the subprogram name is CDOTCI or ZDOTCI, in which case $s = \sum_{i=1}^{m} \text{CONJG}(\mathbf{x}(i)) \times \mathbf{y}(\mathbf{indx}(i))$.

FORTRAN Equivalent

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

Example

Compute the REAL*8 sparse dot product

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

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

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

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:

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

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

VECLIB8:

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

Input

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

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

incx — Increment for the array **x**:

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

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

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

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

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

Output

y — Array of length **leny** = (**n**–1)×|**incy**|+1 containing the n-vector y. If **n** ≤ 0, **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)
```

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

Purpose

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

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

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

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

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

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

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

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

Output

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

Notes

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

FORTRAN Equivalent

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

Example

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

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

Purpose

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

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

$$\begin{cases} \mathbf{x}_i = y_{k_i} \\ y_{k_i} = 0 \end{cases} \quad i = 1, 2, ..., m.$$

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

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

Usage

VECLIB:

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

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

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

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

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

VECLIB8:

```
INTEGER*8 m, indx(m)
REAL*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, $\mathbf{m} \le \mathbf{n}$, where **n** is the length of **y**. If $\mathbf{m} \le 0$, the subprograms do not reference **x**, **indx**, or **y**.

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

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

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

and

$$\mathbf{indx}(i) \ne \mathbf{indx}(j), \quad 1 \le i \ne j \le \mathbf{m},$$

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

Output

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

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

Notes

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

FORTRAN Equivalent

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

Example

Gather y into x and reset y to zero, where y is a vector with nonzero elements y_1, y_4, y_5, and y_9 stored in 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)
```

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** ≤ 0, the subprograms do not reference **x** or **indx**.

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

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

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

a The scalar a.

Output

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

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

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

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 vectorizes, 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
```

Purpose

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

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

The vector 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** $\le$ 0, the subprograms do not reference **x**.

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

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

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

Output

s If **n** $\le$ 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
```

SMAX/DMAX/IMAX **Continued**

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)
```

Purpose

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

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

The vector 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** ≤ 0, the subprograms do not reference **x**.

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

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

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

Output

s If **n** ≤ 0, then **s** $= \infty$, 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
```

SMIN/DMIN/IMIN **Continued**

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)
```

Purpose

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

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

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

Usage

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** ≤ 0, the subprograms do not reference **x**.

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

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

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

Output

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

FORTRAN Equivalent

```
      REAL*4 FUNCTION SNRM2 (N, X,INCX)
      REAL*4 T,X(*)
      SNRM2 = 0.0
      IF ( N .LE. 0 ) RETURN
      T = 0.0
      IX = 1
      INCXA = ABS ( INCX )
      disable overflow and underflow traps
      DO 10 I = 1, N
         T = T + X(IX) ** 2
         IX = IX + INCXA
   10 CONTINUE
      IF ( no overflow occurred ) THEN
        IF ( T .GT. N * 2.0 ** -104 .OR. no underflow occurred ) THEN
           SNRM2 = SQRT ( T )
           RETURN
        ELSE
           SCALE = 2.0 ** 72
        END IF
      ELSE
         SCALE = 0.5 ** 72
      END IF
      T = 0.0
      IX = 1
      DO 20 I = 1, N
         T = T + ( SCALE * X(IX) ) ** 2
         IX = IX + INCXA
   20 CONTINUE
      reenable overflow trap if originally enabled
      SNRM2 = SQRT ( T ) / SCALE
      RETURN
      END
```

Example

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

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

Euclidean Norm Squared — SNRSQ/DNRSQ/SCNRSQ/DZNRSQ

Purpose

Given a real, integer, 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** ≤ 0, the subprograms do not reference **x**.

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

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

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

Output

s — If **n** ≤ 0, then **s** = 0. Otherwise, **s** is the 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)
```

Generate Linear Ramp

SRAMP/DRAMP/IRAMP

Purpose

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

$$x_i = a+(i\text{-}1)\times h, \quad i = 1, 2, \ldots, n.$$

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

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

Output

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

Notes

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

FORTRAN Equivalent

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

Example Generate the linear ramp x with initial value 0 and slope $\pi/9$, 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)
```

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

Purpose

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

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

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

Usually, c and s have been determined by the companion subprogram SROTG, 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** ≤ 0, the subprograms do not reference **x** or **y**.

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

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

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

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

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

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

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

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

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

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

c The scalar c.

s The scalar s.

Output

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

Notes

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

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

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

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
```

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 on your CONVEX supercomputer.

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.

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

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

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

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

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

Usage

VECLIB:

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

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

VECLIB8:

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

Input

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

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

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

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

and

$$\mathbf{indx}(i) \ne \mathbf{indx}(j), \quad 1 \le i \ne j \le \mathbf{m},$$

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 $\mathbf{m} \leq 0$ 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)
```

Purpose

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

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

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

The vectors 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 SROTM (n, x, incx, y, incy, param)

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

VECLIB8:

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

Input

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

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

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

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

Use **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 this chapter.

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

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

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

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

Notes

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

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

Example 1

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

```
      INTEGER*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.OD0
   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
```

Purpose

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

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

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

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

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

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

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

H is chosen to have the same numerical stability as the standard Givens rotation but better computational efficiency. Thus, H will usually have two elements equal to ± 1. When this is true, computing H and applying it to a pair of n-vectors requires $\sim 2n$ floating-point multiplications, $\sim 2n$ floating-point additions, and no square roots. Companion 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, are manipulated by SROTMG or DROTMG as the modified Givens rotations are constructed, then are 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:

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

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

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

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

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

Notes

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 on your CONVEX supercomputer.

Example

Construct a modified Givens plane rotation that will rotate vectors $d_1 x$ and $d_2 y$ in such a way as to annihilate $d_2 y_1$. 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.

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 CSRSCL (n, a, x, incx)
```

Input

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

a The scalar a, **a** ≠ 0.

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

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

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

Output

x If **n** ≤ 0, then **x** is unchanged. Otherwise, $x \div a$ replaces the input.

Notes

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

A divide-by-zero error will occur if **a** = 0 and **n** > 0.

FORTRAN Equivalent

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

Example

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

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

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

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

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

Usage

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** ≤ 0, the subprograms do not reference **x**.

a The scalar a.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the n-vector x. x is used in conjugated form by CSCALC and ZSCALC, and in unconjugated form by the other subprograms. Refer to "Purpose."

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

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

Output

x If **n** ≤ 0, then **x** is unchanged. Otherwise, ax replaces the input.

Notes

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

FORTRAN Equivalent

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

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 $m \le n$ *interesting* (usually nonzero) elements, and let $\{k_1, k_2, \ldots, k_m\}$ be the indices of these elements. If x is represented by arrays **x** and **indx** such that **indx**(i) = k_i and **x**(i) = x_{k_i}, then

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

Usage

VECLIB:

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

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

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

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

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

VECLIB8:

```
INTEGER*8  m, indx(m)
REAL*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** $\le$ **n**, where **n** is the length of **y**. If **m** ≤ 0, the subprograms do not reference **x**, **indx**, or **y**.

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

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

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

and

$$\mathbf{indx}(i) \ne \mathbf{indx}(j), \quad 1 \le i \ne j \le \mathbf{m},$$

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

Output **y** Array containing the elements of y, $\mathbf{y}(i) = y_i$. If $\mathbf{m} \leq 0$, 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)
```

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** $\le$ 0, the subprograms do not reference **x**.

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

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

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

Output s If $n \le 0$, 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)
```

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

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

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

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

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

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

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

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

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

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

Output

x and **y** If **n** ≤ 0, then **x** and **y** are unchanged. Otherwise, x and y are interchanged in **x** and **y**.

Notes

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

FORTRAN Equivalent

```
      SUBROUTINE SSWAP (N, X,INCX, Y,INCY)
      REAL*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)
```

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** ≤ 0, the subprograms do not reference **w**, **x**, or **y**.

w Array of length **lenw** = (**n**–1)× |**incw**|+1 containing the n-vector w.

incw Increment for the array **w**:

incw ≥ 0	w is stored forward in array **w**, i.e., w_i is stored in **w**((i–1)× **incw**+1).
incw < 0	w is stored backward in array **w**, i.e., w_i is stored in **w**((i–**n**)× **incw**+1).

Use **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 **lenx** = (**n**–1)× |**incx**|+1 containing the n-vector x. x is used in conjugated form by CWDOTC and ZWDOTC, and in unconjugated form by the other subprograms. Refer to "Purpose."

incx Increment for the array **x**:

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

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

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

incy Increment for the array **y**:

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

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

Output

s The resulting value of the weighted dot product. If **n** ≤ 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(I,1),20,Y,1)
```

Clear Vector SZERO/DZERO/IZERO/CZERO/ZZERO

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:

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

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

INTEGER*4 n, incx, x(lenx)
CALL IZERO (n, x, incx)

INTEGER*4  n, incx
COMPLEX*8 x(lenx)
CALL CZERO (n, x, incx)

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

VECLIB8:

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

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**, **incx** $\neq$ 0. 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

x Array of length **lenx** = $($**n**$-1)\times|$**incx**$|+1$ containing the n-vector x which has been set to zero. If **n** $\leq$ 0, then **x** is unchanged. Otherwise, $x \leftarrow 0$.

FORTRAN Equivalent

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

Example

Zero the REAL*8 vector 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 CONVEX 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.

Chapter Objectives

After reading this chapter you will:

- be familiar with the Extended BLAS subroutine naming convention
- know what operations the Extended BLAS performs
- know how to use the described subprograms

What You Need to Know to Use These Subprograms

Subroutine Naming Convention

The Extended BLAS uses a subroutine naming convention that encodes the function of each subroutine into its name. Extended BLAS subprogram names consist of four, five, or six characters in the form TXXY, TXXYY, or TXXYYY.

The first letter in the naming convention indicates one of the four FORTRAN data types, as shown in Table 3-1:

Table 3-1: Extended BLAS Naming Convention — Data Type

T	Data Type
S	Single Precision REAL
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.

Supplemental Reading

Dongarra, J.J., J. DuCroz, S. Hammarling, and R. Hanson. "An Extended Set of Fortran Basic Linear Algebra Subprograms." *ACM Transactions on Mathematical Software.* March, 1988. Vol. 14, No. 1.

Dongarra, J.J., J. DuCroz, S. Hammarling, and I. Duff. "A Set of Level 3 Basic Linear Algebra Subprograms." *ACM Transactions on Mathematical Software.* March, 1990. Vol. 16, No. 1.

Higham, Nicholas J. "Is Fast Matrix Multiplication of Practical Use?" *SIAM News.* November, 1990. Vol. 23, No. 6.

Subprogram Descriptions

General Band Matrix-Vector Multiply
SGBMV, DGBMV, CGBMV, ZGBMV 3-5

General Matrix-Matrix Multiply
SGEMM, DGEMM, CGEMM, ZGEMM 3-9

Strassen Matrix-Matrix Multiply
SGEMMS, DGEMMS, CGEMMS, ZGEMMS 3-12

General Matrix-Vector Multiply
SGEMV, DGEMV, CGEMV, ZGEMV 3-16

General Rank-1 Update
SGER, DGER, CGERC, CGERU, ZGERC, ZGERU 3-19

Symmetric or Hermitian Band Matrix-Vector Multiply
SSBMV, DSBMV, CHBMV, ZHBMV 3-22

Symmetric or Hermitian Matrix-Vector Multiply
SSPMV, DSPMV, CHPMV, ZHPMV 3-26

Symmetric or Hermitian Rank-1 Update
SSPR, DSPR, CHPR, ZHPR 3-30

Symmetric or Hermitian Rank-2 Update
SSPR2, DSPR2, CHPR2, ZHPR2 3-33

Symmetric or Hermitian Matrix-Matrix Multiply
SSYMM, DSYMM, CHEMM, CSYMM, ZHEMM, ZSYMM 3-37

Symmetric or Hermitian Matrix-Vector Multiply
SSYMV, DSYMV, CHEMV, ZHEMV 3-41

Symmetric or Hermitian Rank-1 Update
SSYR, DSYR, CHER, ZHER 3-44

Symmetric or Hermitian Rank-2 Update
SSYR2, DSYR2, CHER2, ZHER2 3-47

Symmetric or Hermitian Rank-2k Update
SSYR2K, DSYR2K, CHER2K, CSYR2K, ZHER2K, ZSYR2K 3-50

Symmetric or Hermitian Rank-k Update
SSYRK, DSYRK, CHERK, CSYRK, ZHERK, ZSYRK 3-54

Triangular Band Matrix-Vector Multiply
STBMV, DTBMV, CTBMV, ZTBMV 3-58

Solve Triangular Band System
STBSV, DTBSV, CTBSV, ZTBSV 3-62

Triangular Matrix-Vector Multiply
STPMV, DTPMV, CTPMV, ZTPMV 3-66

Solve One Triangular System
STPSV, DTPSV, CTPSV, ZTPSV 3-70

Triangular Matrix-Matrix Multiply
STRMM, DTRMM, CTRMM, ZTRMM 3-74

Triangular Matrix-Vector Multiply
STRMV, DTRMV, CTRMV, ZTRMV 3-77

Solve Simultaneous Triangular Systems
STRSM, DTRSM, CTRSM, ZTRSM 3-80

Solve One Triangular System
STRSV, DTRSV, CTRSV, ZTRSV 3-83

BLAS Error Handler
XERBLA 3-86

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** ≥ 0. If **m** $= 0$, the subprograms do not reference **ab**, **x**, or **y**.

n Number of columns in matrix A, **n** ≥ 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, $0 \le$ **ku** $<$ **n**.

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

ab Array containing the m-by-n band matrix A in the compressed form described above. If a_{ij} is in the band, it is stored in **ab(ku+1+**$i-j,j$**)**. The columns of A are stored in the columns of **ab**, and the diagonals of A are stored in rows 1 through **kl+ku+1**.

ldab The leading dimension of array **ab** as declared in the calling program unit, with **ldab** $\ge$ **kl+ku+1**.

x Array containing the vector x. The number of elements of x and the value of **lenx**, the dimension of the array **x**, depend on **trans**:

'N' or 'n'	x has n elements	**lenx** = (**n**–1)×|**incx**|+1
otherwise	x has m elements	**lenx** = (**m**–1)×|**incx**|+1

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

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

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

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

beta The scalar β.

y Array containing the vector y. The number of elements of y and the value of **leny**, the dimension of the array **y**, depend on **trans**:

'N' or 'n'	y has m elements	**leny** = (**m**–1)×|**incy**|+1
otherwise	y has n elements	**leny** = (**n**–1)×|**incy**|+1

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

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

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

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

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

Output

y The updated y vector replaces the input.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

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

trans ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c',
m < 0,
n < 0,
kl < 0,
ku < 0,
ldab $<$ **kl+ku+1**,
incx $= 0$, and
incy $= 0$.

Actual character arguments in a subroutine call may be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **trans** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1

Form the REAL*4 matrix-vector product $y = Ax$, where A is a 9 by 6 real band matrix whose lower bandwidth is 2 and whose upper bandwidth is 3. A is stored in an array AB whose dimensions are 10 by 10, x is a real vector 6 elements long stored in an array X of dimension 10, and y is a real vector 9 elements long stored in an array Y, also of dimension 10.

```
CHARACTER*1 TRANS
INTEGER*4   M,N,KL,KU,LDAB,INCX,INCY
REAL*4      ALPHA,BETA,AB(10,10),X(10),Y(10)
TRANS = 'N'
M = 9
N = 6
KL = 2
KU = 3
ALPHA = 1.0
BETA = 0.0
LDAB = 10
INCX = 1
INCY = 1
CALL SGBMV (TRANS,M,N,KL,KU,ALPHA,AB,LDAB,X,INCX,BETA,Y,INCY)
```

Example 2

Form the REAL*8 matrix-vector product $y = \frac{1}{2}y - \rho A^T x$, where ρ is a real scalar, A is a 6-by-9 real band matrix whose lower bandwidth is 1 and whose upper bandwidth is 2. A is stored in an array AB whose dimensions are 10 by 10, x is a real vector 6 elements long stored in an array X of dimension 10, and y is a real vector 9 elements long stored in an array Y, also of dimension 10.

```
INTEGER*4 M,N,KL,KU,LDAB
REAL*8    RHO,AB(10,10),X(10),Y(10)
M = 9
N = 6
KL = 1
KU = 2
LDAB = 10
CALL DGBMV ('TRANSPOSE',M,N,KL,KU,-RHO,AB,LDAB,X,1,0.5D0,Y,1)
```

Purpose These subprograms compute the matrix-matrix product AB, where A is an m-by-k matrix, and B is a k-by-n matrix. Optionally, A may be replaced by A^T or A^*, where A is a k-by-m matrix, and B may be replaced by B^T or B^*, where B is an n-by-k matrix. Here, A^T and B^T are the transposes and A^* and B^* are the conjugate-transposes of A and B, respectively. The product may be stored in the result matrix (which is always of size m by n) or optionally may be added to or subtracted from it. This is handled in a convenient, but general, way by two scalar arguments, α and β, which are used as multipliers of the matrix product and the result matrix. Specifically, these subprograms compute matrix products of the forms

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

Usage **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 **lda** $\geq$ max(the number of rows of A,1).

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

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

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

beta The scalar β.

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

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

Output

c The updated C matrix replaces the input.

Notes

These subprograms conform to specifications of the Level 3 BLAS.

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

transa ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c',
transb ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c',
m < 0,
n < 0,
k < 0,
lda too small,
ldb too small, and
ldc < max(**m**,1).

Actual character arguments in a subroutine call may be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved, for example, by coding the **transa** and **transb** arguments as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1

Form the REAL*4 matrix product $C = AB$, where A is a 9-by-6 real matrix stored in an array A whose dimensions are 10 by 10, B is a 6-by-8 real matrix stored in an array B of dimension 10 by 10, and C is a 9-by-8 real matrix stored in an array C, also of dimension 10 by 10.

```
CHARACTER*1 TRANSA,TRANSB
INTEGER*4   M,N,K,LDA,LDB,LDC
REAL*4      ALPHA,BETA,A(10,10),B(10,10),C(10,10)
TRANSA = 'N'
TRANSB = 'N'
M = 9
N = 8
K = 6
ALPHA = 1.0
BETA = 0.0
LDA = 10
LDB = 10
LDC = 10
CALL SGEMM (TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)
```

Example 2

Form the REAL*8 matrix product $C = \frac{1}{2}C + \rho A^T B$, where ρ is a real scalar, A is a 6-by-9 real matrix stored in an array A whose dimensions are 10 by 10, B is a 6-by-8 real matrix stored in an array B of dimension 10 by 10, and C is a 9-by-8 real matrix stored in an array C, also of dimension 10 by 10.

```
INTEGER*4 M,N,K,LDA,LDB,LDC
REAL*8    RHO,A(10,10),B(10,10),C(10,10)
M = 9
N = 8
K = 6
LDA = 10
LDB = 10
LDC = 10
CALL DGEMM ('TRAN','NONTRAN',M,N,K,RHO,A,LDA,B,LDB,0.5D0,C,LDC)
```

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

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

Usage

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 **lda** $\geq$ max(the number of rows of A,1).
	b	Array containing the matrix B, whose size is indicated by **transb**: 'N' or 'n' B is a k-by-n matrix otherwise B is an n-by-k matrix
	ldb	The leading dimension of array **b** as declared in the calling program unit, with **ldb** $\geq$ max(the number of rows of B,1).
	beta	The scalar β.
	c	Array containing the m-by-n matrix C. Not used as input if **beta** = 0.
	ldc	The leading dimension of array **c** as declared in the calling program unit, with **ldc** $\geq$ max(**m**,1).
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', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Form the REAL*8 matrix product $C = AB$, where A is a 900-by-600 real matrix stored in an array A whose dimensions are 1000 by 1000, B is a 600-by-800 real matrix stored in an array B of dimension 1000 by 1000, and C is a 900-by-800 real matrix stored in an array C, also of dimension 1000 by 1000.

```
      CHARACTER*1 TRANSA,TRANSB
      INTEGER*4   M,N,K,LDA,LDB,LDC
      REAL*8      ALPHA,BETA,A(1000,1000),B(1000,1000),
     &            C(1000,1000)
      TRANSA = 'N'
      TRANSB = 'N'
      M = 900
      N = 800
      K = 600
      ALPHA = 1.0
      BETA = 0.0
      LDA = 1000
      LDB = 1000
      LDC = 1000
      CALL DGEMMS (TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,
     &        BETA,C,LDC)
```

Example 2 Form the COMPLEX*16 matrix product $C = \frac{1}{2}C + \rho A^*B$, where ρ is a complex scalar, A is a 600-by-900 complex matrix stored in an array A whose dimensions are 1000 by 1000, B is a 600-by-800 complex matrix stored in an array B of dimension 1000 by 1000, and C is a 900-by-800 complex matrix stored in an array C, also of dimension 1000 by 1000.

```
      INTEGER*4  M,N,K,LDA,LDB,LDC
      COMPLEX*16 RHO,A(1000,1000),B(1000,1000),C(1000,1000)
      M = 900
      N = 800
      K = 600
      LDA = 1000
      LDB = 1000
      LDC = 1000
      CALL ZGEMMS ('CONJ','NORMAL',M,N,K,RHO,A,LDA,B,LDB,
     &        (0.5D0,0.0D0),C,LDC)
```

Purpose

These subprograms compute the matrix-vector products Ax, A^Tx, and A^*x, where A is an m-by-n matrix, A^T is the transpose of A, and A^* is the conjugate transpose of A. The product may be stored in the result array, or optionally added to or subtracted from it. This is handled in a convenient, but general, way by two scalar arguments, α and β, which are used as multipliers of the matrix-vector product and the result vector. Specifically, these subprograms compute matrix-vector products of the forms

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

Usage

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 $y \leftarrow \alpha A^Tx + \beta y$
'C' or 'c'	Compute $y \leftarrow \alpha A^*x + \beta y$

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

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

n Number of columns in matrix A, **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 m elements	**lenx** = (**m**–1)× |**incx**|+1
otherwise	x has n elements	**lenx** = (**n**–1)× |**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)×**incx**+1).

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

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

beta The scalar β.

y Array containing the vector y. The number of elements of y and the value of **leny**, the dimension of the array **y**, depend on **trans**:

'N' or 'n'	y has n elements	**leny** = (**n**–1)× |**incy**|+1
otherwise	y has m elements	**leny** = (**m**–1)× |**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)×**incy**+1).

incy < 0 y is stored backward in array **y**, i.e., if **trans** = 'N' or 'n', then y_i is stored in **y**((i-**n**)×**incy**+1); otherwise, y_i is stored in **y**((i-**m**)×**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). Réfer 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,
lda < max(**m**,1),
incx = 0, and
incy = 0.

Actual character arguments in a subroutine call may be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **trans** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Form the REAL*4 matrix-vector product $y = Ax$, where A is a 9-by-6 real matrix stored in an array **A** whose dimensions are 10 by 10, x is a real vector 6 elements long stored in an array X of dimension 10, and y is a real vector 9 elements long stored in an array Y, also of dimension 10.

```
CHARACTER*1 TRANS
INTEGER*4   M,N,LDA,INCX,INCY
REAL*4      ALPHA,BETA,A(10,10),X(10),Y(10)
TRANS = 'N'
M = 9
N = 6
ALPHA = 1.0
BETA = 0.0
LDA = 10
INCX = 1
INCY = 1
CALL SGEMV (TRANS,M,N,ALPHA,A,LDA,X,INCX,BETA,Y,INCY)
```

Example 2 Form the REAL*8 matrix-vector product $y = \frac{1}{2}y - \rho A^T x$, where ρ is a real scalar, A is a 6-by-9 real matrix stored in an array A whose dimensions are 10 by 10, x is a real vector 6 elements long stored in an array X of dimension 10, and y is a real vector 9 elements long stored in an array Y, also of dimension 10.

```
INTEGER*4 M,N,LDA
REAL*8    RHO,A(10,10),X(10),Y(10)
M = 9
N = 6
LDA = 10
CALL DGEMV ('TRANSPOSE',M,N,-RHO,A,LDA,X,1,0.5D0,Y,1)
```

Rank-1 Update SGER/DGER/CGERC/CGERU/ZGERC/ZGERU

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 **lenx** = (**m**−1)×|**incx**|+1 containing the m-vector x.

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

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

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

y Array of length **leny** = (**n**–1)×|**incy**|+1 containing the n-vector y. y is used in conjugated form by CGERC and ZGERC, and in unconjugated form by the other subprograms. Refer to "Purpose."

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

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

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

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

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

Output

a The updated A matrix replaces the input.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

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

m < 0,
n < 0,
lda < max(**m**,1),
incx = 0, and
incy = 0.

Example 1 Apply a REAL*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)
```

Purpose

These subprograms compute the matrix-vector product Ax where A is an n-by-n real symmetric or complex Hermitian band matrix and x is a real or complex n-vector. The product may be stored in the result array, or, optionally, be added to or subtracted from it. This is handled in a convenient, but general, way by two scalar arguments, α and β, which are used as multipliers of the matrix-vector product and the result vector. Specifically, these subprograms compute the matrix-vector product of the form

$$y \leftarrow \alpha Ax + \beta y.$$

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

SSBMV 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** ≥ **kd**+1.

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

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

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

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

beta The scalar β.

y Array of length **leny** = (**n**–1)×|**incy**|+1 containing the n-vector y. Not used as input if **beta** = 0.

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

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

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

Output

y The updated y vector replaces the input.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

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

uplo ≠ '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)
```

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**×(**n**+1)/2 containing the upper or lower triangle, as specified by **uplo**, of an n-by-n real symmetric or complex Hermitian matrix A, stored by columns in the packed form described above.

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

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

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

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

beta The scalar β.

y Array of length **leny** = (**n**-1)×|**incy**|+1 containing the n-vector y. Not used as input if **beta** = 0.

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

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

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

Output

y The updated y vector replaces the input.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

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

uplo ≠ '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 $y = \frac{1}{2}y - \rho Ax$, where ρ is a complex scalar, A is a 9-by-9 complex Hermitian matrix whose lower triangle is stored in packed form in an array AP of dimension 55, x is a complex vector 9 elements long stored in an array X of dimension 10, and y is a complex vector 9 elements long stored in an array Y, also of dimension 10.

```
INTEGER*4 N
COMPLEX*8 RHO,AP(55),X(10),Y(10)
N = 9
CALL CHPMV ('LOWER',N,-RHO,AP,X,1,(0.5D0,0.0D0),Y,1)
```

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 **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
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

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

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

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

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

ap Array of length **lenap** = **n**×(**n**+1)/2 containing the upper or lower triangle, as specified by **uplo**, of an n-by-n real symmetric or complex Hermitian matrix A, stored by columns in the packed form described above.

Output

ap The upper or lower triangle of the updated A matrix, as specified by **uplo**, replaces the input.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

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

Purpose

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

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

where A is an n-by-n real symmetric or complex Hermitian matrix 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** ≥ 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 **lenx** = (**n**–1)×|**incx**|+1 containing the n-vector x.

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

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

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

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

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

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

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

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

ap Array of length **lenap** = **n**×(**n**+1)/2 containing the upper or lower triangle, as specified by **uplo**, of an n-by-n real symmetric or complex Hermitian matrix A, stored by columns in the packed form described above.

Output

ap The upper or lower triangle of the updated A matrix, as specified by **uplo**, replaces the input.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

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

uplo ≠ 'L' or 'l' or 'U' or 'u',
n < 0,
incx = 0, and
incy = 0.

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

Example 1

Apply a REAL*4 symmetric rank-2 update $xy^T + x^Ty$ to A, where A is a 9-by-9 real symmetric matrix whose upper triangle is stored in packed form in an array AP of dimension 55, x is a real vector 9 elements long stored in an array X of dimension 10, and y is a real vector 9 elements long stored in an array Y also of dimension 10.

```
CHARACTER*1 UPLO
INTEGER*4   N,INCX,INCY
REAL*4      ALPHA,AP(55),X(10),Y(10)
UPLO = 'U'
N = 9
ALPHA = 1.0
INCX = 1
INCY = 1
CALL SSPR2 (UPLO,N,ALPHA,X,INCX,Y,INCY,AP)
```

Example 2 Apply a COMPLEX*8 Hermitian rank-2 update $\alpha xy^* + \bar{\alpha} yx^*$ to A, where A is a 9-by-9 complex Hermitian matrix whose lower triangle is stored in packed form in an array AP of dimension 55, α is a complex scalar, x is a complex vector 9 elements long stored in an array X of dimension 10, and y is a complex vector 9 elements long stored in an array Y of dimension 10.

```
INTEGER*4 N
COMPLEX*8 ALPHA,AP(55),X(10),Y(10)
N = 9
CALL CHPR2 ('LOWER',N,ALPHA,X,1,Y,1,AP)
```

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 A is the left or right matrix operand:

'L' or 'l' A is the left matrix operand, i.e. compute $C \leftarrow \alpha AB + \beta C$

'R' or 'r' A is the right matrix operand, i.e. compute $C \leftarrow \alpha BA + \beta C$

uplo Upper/lower triangular storage option for A:

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

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

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** ≥ 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 ≠ 'L' or 'l' or 'R' or 'r',
uplo ≠ 'L' or 'l' or 'U' or 'u',
m < 0,
n < 0,
lda too small,
ldb < max(**m**,1), 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 $C = \frac{1}{2}BA - \rho C$, where ρ is a scalar, A is an 8-by-8 complex Hermitian matrix whose lower triangle is stored in the lower triangle of an array A of dimension 10 by 10, B is a 6-by-8 complex matrix stored in an array whose dimensions are 10 by 10, and C is a 6-by-8 complex matrix stored in an array C, also of dimension 10 by 10.

```
INTEGER*4 M,N,LDA,LDB,LDC
COMPLEX*8 HALF,RHO,A(10,10),B(10,10),C(10,10)
M = 6
N = 8
LDA = 10
LDB = 10
LDC = 10
HALF = (0.5,0.0)
CALL CHEMM ('RIGHT','LOWER',M,N,-RHO,A,LDA,B,LDB,HALF,C,LDC)
```

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)×|**incx**|+1 containing the n-vector x.

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

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

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

beta The scalar β.

y Array of length **leny** = (**n**–1)×|**incy**|+1 containing the n-vector y. Not used as input if **beta** = 0.

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

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

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

Output **y** The updated y vector replaces the input.

Notes These subprograms conform to specifications of the Level 2 BLAS.

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

uplo $\neq$ 'L' or 'l' or 'U' or 'u',
n < 0,
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 $y = \frac{1}{2}y - \rho Ax$, where ρ is a complex scalar, A is a 9-by-9 complex Hermitian matrix whose lower triangle is stored in the lower triangle of an array A whose dimensions are 10 by 10, x is a complex vector 9 elements long stored in an array X of dimension 10, and y is a complex vector 9 elements long stored in an array Y, also of dimension 10.

```
INTEGER*4 N,LDA
COMPLEX*8 RHO,A(10,10),X(10),Y(10)
N = 9
LDA = 10
CALL CHEMV ('LOWER',N,-RHO,A,LDA,X,1,(0.5,0.0),Y,1)
```

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

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

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

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

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

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

Output

a The upper or lower triangle of the updated A matrix, as specified by **uplo**, replaces the upper or lower triangle of the input, respectively. The other triangle of **a** is unchanged.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

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

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

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

Example 1 Apply a REAL*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)
```

Purpose

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

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

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

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

SSYR2 or DSYR2	A is a real symmetric matrix
CHER2 or ZHER2	A is a complex Hermitian matrix

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** ≥ 0. If **n** = 0, the subprograms do not reference **a**, **x**, or **y**.
	alpha	The scalar α. If **alpha** = 0, the subprograms do not reference **a**, **x**, or **y**.
	x	Array of length **lenx** = (**n**–1)×|**incx**|+1 containing the n-vector x.
	incx	Increment for the array **x**, **incx** ≠ 0: **incx** > 0 x is stored forward in array **x**, i.e., x_i is stored in **x**((i-1)×**incx**+1). **incx** < 0 x is stored backward in array **x**, i.e., x_i is stored in **x**((i-**n**)×**incx**+1). Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., if x_i is stored in **x**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.
	y	Array of length **leny** = (**n**–1)×|**incy**|+1 containing the n-vector y.
	incy	Increment for the array **y**, **incy** ≠ 0: **incy** > 0 y is stored forward in array **y**, i.e., y_i is stored in **y**((i-1)×**incy**+1). **incy** < 0 y is stored backward in array **y**, i.e., y_i is stored in **y**((i-**n**)×**incy**+1). Use **incy** = 1 if the vector y is stored contiguously in array **y**, i.e., if y_i is stored in **y**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.
	a	Array whose upper or lower triangle, as specified by **uplo**, contains the upper or lower triangle of an n-by-n real symmetric or complex Hermitian matrix A. The other triangle of **a** is not referenced.
	lda	The leading dimension of array **a** as declared in the calling program unit, with **lda** ≥ max(**n**,1).
Output	**a**	The upper or lower triangle of the updated A matrix, as specified by **uplo**, replaces the upper or lower triangle of the input, respectively. The other triangle of **a** is unchanged.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

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

uplo ≠ 'L' or 'l' or 'U' or 'u',
n < 0,
lda < max(**n**,1),
incx = 0, and
incy = 0.

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

Example 1

Apply a REAL*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)
```

Purpose

These subprograms apply a symmetric or Hermitian rank-$2k$ update to a real symmetric, complex symmetric, or complex Hermitian matrix; specifically they compute the following operations:

for symmetric C: $C \leftarrow \alpha AB^T + \bar{\alpha} BA^T + \beta C$ and $C \leftarrow \alpha A^T B + \bar{\alpha} B^T A + \beta C$

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

'T' and 't' are invalid in subprograms CHER2K and ZHER2K, and 'C' and 'c' are invalid in subprograms CSYR2K and ZSYR2K. In subprograms SSYR2K and DSYR2K, 'C' and 'c' have the same meaning as 'T' and 't'.

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

k Number of rows or columns in matrices A and B, depending on **trans**; refer to the description of **a** for details. **k** $\geq$ 0; if **k** = 0, the subprograms do not reference **a** or **b**.

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

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

'N' or 'n'	A is n by k
otherwise	A is k by n

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

b Array containing matrix B, which is the same size as matrix A. Refer to the description of **a** above for details.

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

beta The scalar β.

c Array whose upper or lower triangle, as specified by **uplo**, contains the upper or lower triangle of the n-by-n symmetric or Hermitian matrix C. Not used as input if **beta** = 0.

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

Output

c The upper or lower triangle of the updated matrix C, as specified by **uplo**, replaces the upper or lower triangle of the input, respectively. The other triangle of **c** is unchanged.

Notes

These subprograms conform to specifications of the Level 3 BLAS.

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

uplo ≠ 'L' or 'l' or 'U' or 'u',
trans ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c',
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)
```

SSYRK/DSYRK/CHERK/CSYRK/ZHERK/ZSYRK — Rank-k Update

Purpose

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

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

where α and β are scalars, C is an n-by-n real symmetric, complex symmetric, or complex Hermitian matrix, and A is a matrix whose size, either n by k or k by n, depends on which form of the update is requested. Here, A^T and A^* are the transpose and conjugate transpose of A, respectively.

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

SSYRK 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 CHER2K and ZHER2K, and 'C' and 'c' are invalid in subprograms CSYR2K and ZSYR2K. In subprograms SSYRK and DSYRK, 'C' and 'c' have the same meaning as 'T' and 't'.

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

k Number of rows or columns in matrix A, **k** ≥ 0, depending on **trans**; refer to description of **A** for details. If **k** = 0, the subprograms do not reference **a**.

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

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

'N' or 'n'	A is n by k
otherwise	A is k by n

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

beta The scalar β.

c Array whose upper or lower triangle, as specified by **uplo**, contains the upper or lower triangle of the n-by-n symmetric or Hermitian matrix C. Not used as input if **beta** = 0.

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

Output

c The upper or lower triangle of the updated C matrix, as specified by **uplo**, replaces the upper or lower triangle of the input, respectively. The other triangle of **c** is unchanged.

Notes

These subprograms conform to specifications of the Level 3 BLAS.

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

uplo ≠ 'L' or 'l' or 'U' or 'u',
trans ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c',
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)
```

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

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

A lower-triangular band matrix is a matrix whose strict upper triangle is zero, and whose nonzero lower-triangular elements all are on or fairly near the principal diagonal. Specifically, $a_{ij} \neq 0$ only if $0 \leq i-j \leq kd$ for some integer kd. In contrast, an upper-triangular band matrix is a matrix whose strict lower triangle is zero, and whose nonzero upper-triangular elements all are on or fairly near the principal diagonal, i.e., with $a_{ij} \neq 0$ only if $0 \leq j-i \leq kd$.

Matrix Storage

Triangular band matrices are stored in a compressed form that takes advantage of knowing the positions of the only elements that may be nonzero. The following examples illustrate the storage of triangular band matrices.

Lower triangular storage. If A is a 9-by-9 lower-triangular band matrix with bandwidth $kd = 3$, for example,

11	0	0	0	0	0	0	0	0
21	22	0	0	0	0	0	0	0
31	32	33	0	0	0	0	0	0
41	42	43	44	0	0	0	0	0
0	52	53	54	55	0	0	0	0
0	0	63	64	65	66	0	0	0
0	0	0	74	75	76	77	0	0
0	0	0	0	85	86	87	88	0
0	0	0	0	0	96	97	98	99

the lower triangular band part of A is stored in an array **ab** with at least $kd+1 = 4$ rows and 9 columns:

11	22	33	44	55	66	77	88	99
21	32	43	54	65	76	87	98	*
31	42	53	64	75	86	97	*	*
41	52	63	74	85	96	*	*	*

where asterisks represent elements in the kd-by-kd triangle at the lower-right corner of **ab** that are not referenced. Thus, if a_{ij} is an element within the band of A, it is stored in **ab**$(1+i-j,j)$. Therefore, the columns of A are stored in the columns of **ab**, and the diagonals of A are stored in the rows of **ab**, with the principal diagonal in the first row, the first subdiagonal in the second row, and so on.

Upper triangular storage. If A is a 9-by-9 upper-triangular band matrix with bandwidth $kd = 3$, for example,

11	12	13	14	0	0	0	0	0
0	22	23	24	25	0	0	0	0
0	0	33	34	35	36	0	0	0
0	0	0	44	45	46	47	0	0
0	0	0	0	55	56	57	58	0
0	0	0	0	0	66	67	68	69
0	0	0	0	0	0	77	78	79
0	0	0	0	0	0	0	88	89
0	0	0	0	0	0	0	0	99

the upper triangular band part of A is stored in an array **ab** with at least $kd+1 = 4$ rows and 9 columns:

*	*	*	14	25	36	47	58	69
*	*	13	24	35	46	57	68	79
*	12	23	34	45	56	67	78	89
11	22	33	44	55	66	77	88	99

where asterisks represent elements in the kd-by-kd triangle at the upper-left corner of **ab** that are not referenced. Thus, if a_{ij} is an element within the band of A, it is stored in **ab**$(kd+1+i-j,j)$. Therefore, the columns of A are stored in the columns of **ab**, and the diagonals of A are stored in the rows of **ab**, with the principal diagonal in row $kd+1$, the first superdiagonal starting in the second position in row kd, and so on.

Usage

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)×|**incx**|+1 containing the input vector x.

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

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

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

Output

x The updated x vector replaces the input.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

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

uplo ≠ 'L' or 'l' or 'U' or 'u',
trans ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c',
diag ≠ 'N' or 'n' or 'U' or 'u',
n < 0,
kd < 0,
ldab < **kd**+1, and
incx = 0.

Actual character arguments in a subroutine call may be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **trans** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1

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','TRANSPOSE','NONUNIT',N,KD,AB,LDAB,X,1)
```

Purpose

Given an n-by-n upper- or lower-triangular band matrix A, and an n-vector x, these subprograms overwrite x with the solution y to the system of linear equations $Ay = x$. This is the forward elimination or back substitution step of Gaussian elimination for band matrices. Optionally, A may be replaced by A^T, the transpose of A, or by A^*, the conjugate transpose of A.

A lower-triangular band matrix is a matrix whose strict upper triangle is zero, and whose nonzero lower-triangular elements all are on or fairly near the principal diagonal. Specifically, $a_{ij} \neq 0$ only if $0 \leq i-j \leq kd$ for some integer kd. In contrast, an upper-triangular band matrix is a matrix whose strict lower triangle is zero, and whose nonzero upper-triangular elements all are on or fairly near the principal diagonal, but with $a_{ij} \neq 0$ only if $0 \leq j-i \leq kd$.

Specifically, these subprograms compute

$$x \leftarrow A^{-1}x, \quad x \leftarrow A^{-T}x, \quad \text{and} \quad x \leftarrow A^{-*}x$$

where A^{-T} is the inverse of the transpose of A, and A^{-*} is the inverse of the conjugate transpose of A.

These subprograms are more primitive than the LINPACK band equation solvers. As such, they are intended to supplement 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 **ldab** $\geq$ **kd**+1.

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

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

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

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

Output x The solution vector of the triangular band system replaces the input.

Notes These subprograms conform to specifications of the Level 2 BLAS.

The subprograms do not check for singularity of matrix A. A is singular if **diag** = 'N' or 'n' and some $a_{ii} = 0$. This condition causes a division by zero to occur. Therefore, the program must detect singularity and take appropriate action to avoid a problem before calling any of these subprograms.

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

uplo ≠ 'L' or 'l' or 'U' or 'u',
trans ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c',
diag ≠ 'N' or 'n' or 'U' or 'u',
n < 0,
kd < 0,
ldab < **kd**+1, and
incx = 0.

Actual character arguments in a subroutine call may be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **trans** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Perform REAL*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)
```

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

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

Matrix Storage

You supply the upper or lower triangle of A, stored column-by-column in packed form in a 1-dimensional array. This saves memory compared to storing the entire matrix.

The following examples illustrate the packed storage of a triangular matrix.

Upper triangular matrix. If A is

11	12	13	14
0	22	23	24
0	0	33	34
0	0	0	44

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

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

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

Lower triangular matrix. If A is

11	0	0	0
21	22	0	0
31	32	33	0
41	42	43	44

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

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

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

Usage **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** ≥ 0. If **n** = 0, the subprograms do not reference **ap** or **x**.

ap Array of length **lenap** = **n**×(**n**+1)/2 containing the n-by-n triangular matrix A, stored by columns in the packed form described above. Space must be left for the diagonal elements of A even when **diag** is supplied as 'U' or 'u'.

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

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

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

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

Output

x The updated x vector replaces the input.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

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

uplo ≠ 'L' or 'l' or 'U' or 'u',
trans ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c',
diag ≠ 'N' or 'n' or 'U' or 'u',
n < 0, and
incx = 0.

Actual character arguments in a subroutine call may be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **trans** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1

Form the REAL*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','TRANSPOSE','NONUNIT',N,AP,X,1)
```

Purpose

Given an n-by-n upper- or lower-triangular matrix A stored in packed form as described in "Matrix Storage," and an n-vector x, these subprograms overwrite x with the solution y to the system of linear equations $Ay = x$. This is the forward elimination or back substitution step of Gaussian elimination. Optionally, A may be replaced by A^T, the transpose of A, or by A^*, the conjugate transpose of A. Specifically, these subprograms compute

$$x \leftarrow A^{-1}x, \quad x \leftarrow A^{-T}x, \quad \text{and} \quad x \leftarrow A^{-*}x$$

where A^{-T} is the inverse of the transpose of A, and A^{-*} is the inverse of the conjugate transpose of A.

These subprograms are more primitive than the LINPACK linear equation solvers. As such, they are intended to supplement but not replace 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

$$\begin{matrix} 11 & 12 & 13 & 14 \\ 0 & 22 & 23 & 24 \\ 0 & 0 & 33 & 34 \\ 0 & 0 & 0 & 44 \end{matrix}$$

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

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

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

Lower triangular matrix. If A is

$$\begin{matrix} 11 & 0 & 0 & 0 \\ 21 & 22 & 0 & 0 \\ 31 & 32 & 33 & 0 \\ 41 & 42 & 43 & 44 \end{matrix}$$

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

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

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

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 **lenap** = **n**×(**n**+1)/2 containing the n-by-n triangular matrix A, stored by columns in the packed form described above. Space must be left for the diagonal elements of A even when **diag** is supplied as 'U' or 'u'.

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

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

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

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

Output **x** The solution vector of the triangular system replaces the input.

Notes These subprograms conform to specifications of the Level 2 BLAS.

The subprograms do not check for singularity of matrix A. A is singular if **diag** = 'N' or 'n' and some $a_{ii} = 0$. This condition will cause a division by zero to occur. Therefore, the program must detect singularity and take appropriate action to avoid a problem before calling any of these subprograms.

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

uplo ≠ 'L' or 'l' or 'U' or 'u',
trans ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c',
diag ≠ 'N' or 'n' or 'U' or 'u',
n < 0, and
incx = 0.

Actual character arguments in a subroutine call may be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **trans** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Perform REAL*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)
```

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

$$B \leftarrow \alpha AB, \quad B \leftarrow \alpha A^T B, \quad B \leftarrow \alpha A^* B,$$
$$B \leftarrow \alpha BA, \quad B \leftarrow \alpha BA^T, \quad B \leftarrow \alpha BA^*.$$

Matrix Storage

For these subprograms, you supply A in a two-dimensional array large enough to hold a square matrix. The other triangle of the array is not referenced. If A has an unstored unit diagonal (see input argument **diag**), then the diagonal elements of the array also will not be referenced.

Usage

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** ≥ 0. If **m** $= 0$, the subprograms do not reference **a** or **b**.

n Number of columns in matrix B, **n** ≥ 0. If **n** $= 0$, the subprograms do not reference **a** or **b**.

alpha The scalar α. If **alpha** $= 0$, the subprograms compute $B \leftarrow 0$ without referencing **a**.

a Array whose upper or lower triangle, as specified by **uplo**, contains the upper- or lower-triangular matrix A, whose size is indicated by **side**:

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

The other triangle of **a** is not referenced. Not used as input if **alpha** $= 0$.

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

b Array containing the m-by-n matrix B. Not used as input if **alpha** $= 0$.

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

Output

b The indicated matrix product replaces the input.

Notes

These subprograms conform to specifications of the Level 3 BLAS.

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

side ≠ 'L' or 'l' or 'R' or 'r',
uplo ≠ 'L' or 'l' or 'U' or 'u',
transa ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c',
diag ≠ 'N' or 'n' or 'U' or 'u',
m < 0,
n < 0,
lda too small, 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 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)
```

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

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

Matrix Storage

For these subprograms, you supply A in a two-dimensional array large enough to hold a square matrix. The other triangle of the array is not referenced. If A has an unstored unit diagonal (see input argument **diag**), then the diagonal elements of the array also will not be referenced.

Usage

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)×|**incx**|+1 containing the input vector x.

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

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

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

Output

x The updated x vector replaces the input.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

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

uplo $\neq$ 'L' or 'l' or 'U' or 'u',
trans $\neq$ 'N' or 'n' or 'T' or 't' or 'C' or 'c',
diag $\neq$ 'N' or 'n' or 'U' or 'u',
n < 0,
lda < max(**n**,1), and
incx = 0.

Actual character arguments in a subroutine call may be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **trans** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1

Form the REAL*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)
```

Purpose

Given a scalar α, an upper- or lower-triangular matrix A, and an m-by-n matrix B, these subprograms compute either of the matrix solutions $\alpha A^{-1}B$ or αBA^{-1}. The size of A, either m by m or n by n, depends on which matrix solution is requested. Optionally, A^{-1} may be replaced by A^{-T}, the inverse of the transpose of A, or by A^{-*}, the inverse of the conjugate transpose of A. The resulting matrix solution overwrites the input B matrix. Specifically, these subprograms compute matrix solutions of the forms

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

Matrix Storage

For these subprograms, you supply A in a two-dimensional array large enough to hold a square matrix. The other triangle of the array is not referenced. If A has an unstored unit diagonal (see input argument **diag**), then the diagonal elements of the array also will not be referenced.

Usage

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, $B \leftarrow \alpha BA^{-1}$

uplo Upper/lower triangular option for A:

'L' or 'l'	A is a lower-triangular matrix
'U' or 'u'	A is an upper-triangular matrix

transa Transposition option for A:

'N' or 'n'	Use matrix A^{-1}
'T' or 't'	Use A^{-T}, the inverse of the transpose of A
'C' or 'c'	Use A^{-*}, the inverse of the conjugate transpose of A

In the real subprograms, 'C' and 'c' have the same meaning as 'T' and 't'.

diag Specifies whether the A matrix is unit triangular, i.e., $a_{ii} = 1$, or not:

'N' or 'n'	The diagonal of A is stored in the array
'U' or 'u'	The diagonal of A consists of unstored ones

When **diag** is supplied as 'U' or 'u', the diagonal elements of A are not referenced.

m Number of rows in matrix B, **m** ≥ 0. If **m** = 0, the subprograms do not reference **a** or **b**.

n Number of columns in matrix B, **n** ≥ 0. If **n** = 0, the subprograms do not reference **a** or **b**.

alpha The scalar α. If **alpha** = 0, the subprograms compute $B \leftarrow 0$ without referencing **a**.

a Array whose upper or lower triangle, as specified by **uplo**, contains the upper- or lower-triangular matrix A, whose size is indicated by **side**:

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

The other triangle of **a** is not referenced. Not used as input if **alpha** = 0.

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

b Array containing the m-by-n matrix B. Not used as input if **alpha** = 0.

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

Output **b** The indicated matrix solution replaces the input.

Notes

These subprograms conform to specifications of the Level 3 BLAS.

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

side ≠ 'L' or 'l' or 'R' or 'r',
uplo ≠ 'L' or 'l' or 'U' or 'u',
transa ≠ 'N' or 'n' or 'T' or 't' or 'C' or 'c',
diag ≠ 'N' or 'n' or 'U' or 'u',
m < 0,
n < 0,
lda too small, 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)
```

Purpose

Given an n-by-n upper- or lower-triangular matrix A, and an n-vector x, these subprograms overwrite x with the solution y to the system of linear equations $Ay = x$. This is the forward elimination or back substitution step of Gaussian elimination. Optionally, A may be replaced by A^T, the transpose of A, or by A^*, the conjugate transpose of A. Specifically, these subprograms compute

$$x \leftarrow A^{-1}x, \quad x \leftarrow A^{-T}x, \quad \text{and} \quad x \leftarrow A^{-*}x$$

where A^{-T} is the inverse of the transpose of A, and A^{-*} is the inverse of the conjugate transpose of A.

These subprograms are more primitive than the LINPACK linear equation solvers. As such, they are intended to supplement but not replace 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 **lenx** = (**n**–1)×|**incx**|+1 containing the right-hand-side n-vector x.

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

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

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

Output

x The solution vector of the triangular system replaces the input.

Notes

These subprograms conform to specifications of the Level 2 BLAS.

The subprograms do not check for singularity of matrix A. A is singular if **diag** = 'N' or 'n' and some $a_{ii} = 0$. This condition will cause a division by zero to occur. Therefore, the program must detect singularity and take appropriate action to avoid a problem before calling any of these subprograms.

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

uplo $\neq$ 'L' or 'l' or 'U' or 'u',
trans $\neq$ 'N' or 'n' or 'T' or 't' or 'C' or 'c',
diag $\neq$ 'N' or 'n' or 'U' or 'u',
n < 0,
lda $<$ max(**n**,1), and
incx $= 0$.

Actual character arguments in a subroutine call may be longer than the corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **trans** argument as 'NORMAL' or 'NONTRANS' for 'N', 'TRANSPOSE' for 'T', or 'CTRANS' for 'C'. Refer to "Example 2."

Example 1 Perform REAL*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)
```

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, a call traceback is also written onto the standard error file. XERBLA then terminates execution with a nonzero exit status.

You may supply a version of XERBLA that alters this action. Be aware that other subprograms, including many in CONVEX LAPACK, also call XERBLA. All BLAS, VECLIB, SCILIB, and LAPACK subprograms that call XERBLA follow the **CALL XERBLA** statement with a **RETURN** statement, so your version of XERBLA can exit with a **RETURN** statement. However, many of those subprograms do not have a status response variable in their argument list that could be used to alert the caller. If you write an XERBLA that does not end with a **STOP** statement, you need some other mechanism to detect errors occurring in those subprograms. One such mechanism is a flag in a common block that is set by your XERBLA and tested by the calling program after calls where errors could be detected.

Usage

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. Although VECLIB includes all LINPACK subprograms, only those subprograms optimized for use on CONVEX supercomputers are described in this chapter. Table 4-5 at the end of this chapter lists the LINPACK subprograms that are included in VECLIB but are not documented in the *CONVEX VECLIB Users' Guide*. You may find information for these subprograms in the *LINPACK Users' Guide* included in the VECLIB documentation set.

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 *CONVEX LAPACK User's Guide*. We recommend that you use LAPACK subprograms rather than LINPACK subprograms in new programs. Future optimization efforts will be directed to LAPACK rather than LINPACK.

This chapter explains how to use 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

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 so that it is usually necessary to call two subprograms to perform the above operations. One subprogram is called to process the matrix and another is called to process a particular right-hand side. This division of labor significantly reduces computer time when there is a sequence of problems involving the same matrix but different right-hand sides. It also allows you the flexibility to choose between subprograms that are fast but use a less reliable, elementary test for singularity and subprograms that are slightly slower but use a significantly more reliable test involving an estimate of the condition number of the coefficient matrix.

Condition Number

The condition number, $\kappa(A)$, of the coefficient matrix A measures the sensitivity of the solution x of the system of linear equations $Ax = b$ to errors in the matrix A and the right-hand side b. If δA and δb represent the errors in A and b, respectively, and if $\| \ \|$ represents any vector norm and its subordinate matrix norm, the error δx in x that results from solving $(A+\delta A)(x+\delta x) = b+\delta b$ instead of $Ax = b$ is bounded by

$$\frac{\|\delta x\|}{\|x\|} \leq \frac{\kappa(A)}{1 - \|A^{-1}\| \, \|\delta A\|} \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|b\|} \right).$$

A standard result of numerical analysis shows that the roundoff error introduced by the solution process may be modeled by taking $\|\delta A\|/\|A\|$ and $\|\delta b\|/\|b\|$ to be small multiples of the computer's machine epsilon. Computational singularity of A results in $\kappa(A) = \infty$. A more common situation occurs when A is not numerically singular but is ill-conditioned. When a matrix is ill-conditioned, $\kappa(A)$ is large, so small errors in the matrix and right-hand side and small roundoff errors introduced during the solution process itself are magnified greatly in the solution.

Since $1 < \kappa(A) \leq \infty$, it is more convenient to compute the reciprocal condition number, $1/\kappa(A)$, than $\kappa(A)$ itself. The reciprocal condition number has the interpretation that if $1/\kappa(A)$ approximately equals 10^{-d}, elements of x can be expected to have d fewer significant digits of accuracy than the elements of A or b. Consequently, if errors in the coefficient matrix and right-hand side exceed $1/\kappa(A)$, or if $1/\kappa(A)$ is negligible compared to 1.0, then x may have no significant digits at all.

Determinant and Inverse

Subprograms for computing the determinant and inverse of a matrix are provided, although it is almost never necessary to compute either one. While papers and reference books extensively use the notation "$\det(A) \neq 0$" to mean "A is nonsingular," 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.

Supplemental Reading

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

Forsythe, G., and C.B. Moler. *Computer Solution of Linear Algebraic Systems.* Englewood Cliffs, NJ: Prentice-Hall, Inc. 1967.

Subprogram Descriptions

Factor a General Band Matrix and Estimate its Condition Number
SGBCO, DGBCO, CGBCO, ZGBCO 4-6

Determinant of a General Band Matrix
SGBDI, DGBDI, CGBDI, ZGBDI 4-10

Factor a General Band Matrix
SGBFA, DGBFA, CGBFA, ZGBFA 4-13

Solve Linear Equations with a General Band Matrix
SGBSL, DGBSL, CGBSL, ZGBSL 4-17

Factor a General Matrix and Estimate its Condition Number
SGECO, DGECO, CGECO, ZGECO 4-20

Determinant and Inverse of a General Matrix
SGEDI, DGEDI, CGEDI, ZGEDI 4-23

Factor a General Matrix
SGEFA, DGEFA, CGEFA, ZGEFA 4-26

Solve Linear Equations with a General Matrix
SGESL, DGESL, CGESL, ZGESL 4-28

Solve Linear Equations with a Tridiagonal Matrix
SGTSL, DGTSL, CGTSL, ZGTSL 4-31

Solve Linear Equations with a Tridiagonal Matrix
SGTSV, DGTSV, CGTSV, ZGTSV 4-33

Factor a Positive Definite Band Matrix and Estimate its Condition Number
SPBCO, DPBCO, CPBCO, ZPBCO 4-36

Determinant of a Positive Definite Band Matrix
SPBDI, DPBDI, CPBDI, ZPBDI 4-40

Cholesky Factorization of a Positive Definite Band Matrix
SPBFA, DPBFA, CPBFA, ZPBFA 4-43

Solve Linear Equations with a Positive Definite Band Matrix
SPBSL, DPBSL, CPBSL, ZPBSL 4-46

Factor a Positive Definite Matrix and Estimate its Condition Number
SPOCO, DPOCO, CPOCO, ZPOCO 4-48

Determinant and Inverse of a Positive Definite Matrix
SPODI, DPODI, CPODI, ZPODI 4-51

Cholesky Factorization of a Positive Definite Matrix
SPOFA, DPOFA, CPOFA, ZPOFA 4-54

Solve Linear Equations with a Positive Definite Matrix
SPOSL, DPOSL, CPOSL, ZPOSL 4-56

Solve Linear Equations with a Positive Definite Tridiagonal Matrix
SPTSL, DPTSL, CPTSL, ZPTSL 4-58

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, SGTSV, DGTSL, DGTSV, CGTSL, CGTSV, ZGTSL, or ZGTSV. 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. This is done by presenting the original matrix to the subprogram in an array large enough to satisfy the additional storage requirements. Thus, for the above matrix, A is given in an array **ab** with at least $2kl + ku + 1 = 8$ rows and $n = 9$ columns as follows:

```
 *   *   *   *   *   +   +   +   +
 *   *   *   *   +   +   +   +   +
 *   *   *  14  25  36  47  58  69
 *   *  13  24  35  46  57  68  79
 *  12  23  34  45  56  67  78  89
11  22  33  44  55  66  77  88  99
21  32  43  54  65  76  87  98   *
31  42  53  64  75  86  97   *   *
```

The asterisks in the $(kl + ku)$-by-$(kl + ku)$ triangle at the upper left corner and in the ku-by-ku triangle at the lower right corner represent elements of **ab** that are not referenced, and the plus signs in the first kl rows indicate elements that may be filled in during the factorization. Thus, if a_{ij} is an element within the band of A, then it is stored in **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 2**kl**+**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** ≥ 2**kl**+**ku**+1.

n The order of matrix A, **n** > 0.

kl The lower bandwidth of A, i.e., the number of nonzero diagonals below the principal diagonal in the band, 0 ≤ **kl** < **n**.

ku The upper bandwidth of A, i.e., the number of nonzero diagonals above the principal diagonal in the band, 0 ≤ **ku** < **n**. These subprograms are more efficient if **kl** ≤ **ku**. This usually can be arranged since the factors used by these subprograms can be used to solve either $Ax = b$ or $A^*x = b$.

Working Storage

work An array of size **n**, used for work space.

Output

ab The triangular factors replace the input matrix. **ab** must be preserved between the condition number estimation call and any solve or determinant call.

ipvt The pivot information necessary to construct the permutations in the lower-triangular factor, L. **ipvt** must be preserved between the condition number estimation call and any solve or determinant call.

rcond An estimate of the reciprocal condition number, $1/\kappa(A)$. If **rcond** is small enough so that the logical expression

1.0 + rcond .EQ. 1.0

is true, then A can be regarded as singular to working precision.

Notes These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

Example Factor and estimate the reciprocal condition number of the 9-by-9 REAL*8 general band matrix A whose lower bandwidth is 2 and whose upper bandwidth is 3. A is stored as illustrated above in array **AB** whose dimensions are 8 by 10.

```
INTEGER*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.OD0 + RCOND .EQ. 1.OD0 ) THEN
  handle singular matrix
END IF
```

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** ≥ 2**kl**+**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**.

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 $\det(A) = \mathbf{det}(1) \times 10^{\mathbf{det}(2)}$. This expression may underflow or overflow if evaluated; on the CONVEX supercomputer, underflows automatically flush to zero, but overflows normally terminate execution. For REAL*4 and COMPLEX*8, overflow cannot occur if **det**(2) ≤ 37. For REAL*8 and COMPLEX*16, overflow cannot occur if **det**(2) ≤ 306. If evaluation is safe, an efficient way to do it is with the statement

$$\det(A) = \mathbf{det(1)} \ * \ \mathbf{10.0} \ ** \ \mathbf{INT(det(2))}$$

The value stored in **det(2)** is an integer in REAL or COMPLEX form. **det(1)** is normalized so that either **det**(1) = 0 or $1 \le |Re(\mathbf{det}(1))| + |Im(\mathbf{det}(1))| < 10$, where $Re(z)$ and $Im(z)$ are the real and imaginary parts of z; $Re(z) = z$ and $Im(z) = 0$ if z is real.

Notes

These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

It is almost never necessary to compute 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
```

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, SGTSV, DGTSL, DGTSV, CGTSL, CGTSV, ZGTSL, or ZGTSV. 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 following matrix A of order $n = 9$ and lower and upper bandwidths $kl = 2$ and $ku = 3$, respectively:

11	12	13	14	0	0	0	0	0
21	22	23	24	25	0	0	0	0
31	32	33	34	35	36	0	0	0
0	42	43	44	45	46	47	0	0
0	0	53	54	55	56	57	58	0
0	0	0	64	65	66	67	68	69
0	0	0	0	75	76	77	78	79
0	0	0	0	0	86	87	88	89
0	0	0	0	0	0	97	98	99

When Gaussian elimination is performed on a general band matrix, pivoting introduces nonzero elements outside the band. L can be stored with a lower bandwidth of kl, but U requires an upper bandwidth of $kl + ku$. You must, therefore, provide storage for the extra kl diagonals. This is done by presenting the original matrix to the subprogram in an array large enough to satisfy the additional storage requirements. Thus, for the above matrix, A is given in an array **ab** with at least $2kl + ku + 1 = 8$ rows and $n = 9$ columns as follows:

*	*	*	*	*	+	+	+	+
*	*	*	*	+	+	+	+	+
*	*	*	14	25	36	47	58	69
*	*	13	24	35	46	57	68	79
*	12	23	34	45	56	67	78	89
11	22	33	44	55	66	77	88	99
21	32	43	54	65	76	87	98	*
31	42	53	64	75	86	97	*	*

The asterisks in the $(kl + ku)$-by-$(kl + ku)$ triangle at the upper left corner and in the ku-by-ku triangle at the lower right corner represent elements of **ab** that are not referenced, and the plus signs in the first kl rows indicate elements that may be filled in during the factorization. Thus, if a_{ij} is an element within the band of A, then it is stored in $\mathbf{ab}(kl + ku + 1 + i - j, j)$. Therefore, the columns of A are stored in the columns of **ab**, and the diagonals of A are stored in the rows of **ab**, such that the principal diagonal is stored in row $kl + ku + 1$ of **ab**.

Usage

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 2**kl**+**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** ≥ 2**kl**+**ku**+1.

n The order of matrix A, **n** > 0.

kl The lower bandwidth of A, i.e., the number of nonzero diagonals below the principal diagonal in the band, 0 ≤ **kl** < **n**.

ku The upper bandwidth of A, i.e., the number of nonzero diagonals above the principal diagonal in the band, 0 ≤ **ku** < **n**. These subprograms are more efficient if **kl** ≤ **ku**. This usually can be arranged because factors used by these subprograms can be used to solve either $Ax = b$ or $A^*x = b$.

Output

ab The triangular factors replace the input matrix. **ab** must be preserved between the factorization call and any solve or determinant call.

ipvt The pivot information necessary to construct the permutations in the lower triangular factor, L. **ipvt** must be preserved between the factorization call and any solve or determinant call.

ier Status response:

ier = 0	Normal return.
ier = **k** ≠ 0	if u_{kk}=0. (u_{kk} is the k-th element on the diagonal of upper triangular matrix U). Technically, this is not an error condition for these subprograms, but it does indicate that A is computationally singular and that a division by zero will occur if the factorization is used to solve a system of linear equations.

Notes

These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

Example Factor the 9-by-9 REAL*8 general band matrix A whose lower bandwidth is 2 and whose upper bandwidth is 3. A is stored, as illustrated above, in array AB whose dimensions are 8 by 10.

```
INTEGER*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
```

Solve Band Linear Equations — SGBSL/DGBSL/CGBSL/ZGBSL

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** ≥ 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**.

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 ≠ 0 solve $A^*x = b$

Output

b The solution vector x overwrites the right-hand-side vector b.

Notes

These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

Example

Solve a system of linear equations $Ax = b$, where A is a 9-by-9 REAL*8 general band matrix whose lower bandwidth is 2 and whose upper bandwidth is 3. A is stored in array AB whose dimensions are 8 by 10. b is a vector 9 elements long stored in an array B of dimension 10. The more robust, but slightly slower, condition number estimation subprogram is used to factor the coefficient matrix.

```
INTEGER*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
```

Purpose

These subprograms compute the triangular factorization and estimate the condition number of a general dense n-by-n matrix A. Specifically, given A, these subprograms determine an n-by-n permutation matrix P, an n-by-n unit lower-triangular matrix L, and an n-by-n upper-triangular matrix U, such that

$$PA = LU$$

and compute an estimate of $\kappa(A)$, the condition number of A. Refer to "Condition Number" in the introduction to this chapter for a discussion of $\kappa(A)$. When a matrix is ill-conditioned, $\kappa(A)$ is large, so small errors in the matrix and right-hand side, and small roundoff errors introduced during the solution process itself, are magnified greatly in the solution.

Since $1 < \kappa(A) \leq \infty$, these subprograms actually compute the reciprocal condition number, $1/\kappa(A)$. The reciprocal condition number has the interpretation that if $1/\kappa(A)$ approximately equals 10^{-d}, elements of x can be expected to have d fewer significant digits of accuracy than the elements of A or b. Consequently, if errors in the coefficient matrix and right-hand side exceed $1/\kappa(A)$, or if $1/\kappa(A)$ is negligible compared to 1.0, then x may have no significant digits at all.

A set of companion subprograms computes the triangular factorization of a matrix without estimating its condition number. These companion subprograms are faster but provide a less reliable indication of singularity.

The triangular factors may be used to solve a system of linear equations, $Ax = b$, by successively solving $L(Ux) = Pb$. The determinant of A can be computed as $\det(A) = \det(P) \times \det(L) \times \det(U)$. The inverse of A may be formed as $A^{-1} = U^{-1}L^{-1}P$. These operations are performed by a set of companion 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** ≥ max(**n**,1).

n The order of matrix A, **n** ≥ 0.

Working storage

work An array of size **n**, used for work space.

Output

a The triangular factors replace the input matrix: the strict lower triangle of **a** contains the strict lower triangle of L and the upper triangle of **a** contains U. **a** must be preserved between the condition number estimation call and any solve, determinant, or inverse call.

ipvt The pivot information necessary to construct the permutation matrix P. **ipvt** must be preserved between the condition number estimation call and any solve, determinant, or inverse call.

rcond An estimate of the reciprocal condition, $1/\kappa(A)$. If **rcond** is small enough so that the logical expression

1.0 + rcond .EQ. 1.0

is true, then A can be regarded as singular to working precision. If **rcond** is zero, then the companion subprograms for solving and computing the inverse may divide by zero.

Notes

These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

The triangular factors are stored in a different format from the format used by the standard LINPACK subprograms, but are compatible with the 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
```

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) × $10^{\text{det}(2)}$. This expression may underflow or overflow if evaluated; on the CONVEX supercomputer, underflows automatically flush to zero, but overflows normally terminate execution. For REAL*4 and COMPLEX*8, overflow cannot occur if **det**(2) ≤ 37. For REAL*8 and COMPLEX*16, overflow cannot occur if **det**(2) ≤ 306. If evaluation is safe, an efficient way to do it is with the statement

det(A) = **det(1) * 10.0 ** INT(det(2))**

Refer to "Example 2."

The value stored in **det(2)** is an integer in REAL or COMPLEX form. **det(1)** is normalized so that either **det**(1) = 0 or $1 \leq |Re(\textbf{det}(1))| + |Im(\textbf{det}(1))| < 10$, where $Re(z)$ and $Im(z)$ are the real and imaginary parts of z; $Re(z) = z$ and $Im(z) = 0$ if z is real.

Notes

These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

It is almost never necessary to compute either the determinant or the inverse of a matrix. While papers and reference books extensively use the notation "det(A) ≠ 0" to mean "A is nonsingular," 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 DGECO (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
```

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** ≥ max(**n**,1).

n The order of matrix A, **n** ≥ 0.

Output

a The triangular factors replace the input matrix; the strict lower triangle of **a** contains the strict lower triangle of L and the upper triangle of **a** contains U. **a** must be preserved between the factorization call and any solve, determinant, or inverse call.

ipvt The pivot information necessary to construct the permutation matrix P. **ipvt** must be preserved between the factorization call and any solve, determinant, or inverse call.

ier Status response:

ier = 0	Normal return.
ier = k ≠ 0	if u_{kk}=0. (u_{kk} is the k-th element on the diagonal of upper triangular matrix U). Technically, this is not an error condition for these subprograms, but it does indicate that A is computationally singular and that a division by zero will occur if the factorization is used to solve a system of linear equations or to compute the matrix inverse.

Notes

These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

The triangular factors are stored in a different format from the format used by the standard LINPACK subprograms, but are compatible with the 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
```

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** ≥ max(**n**,1).

n The order of matrix A, **n** ≥ 0.

ipvt The pivot information necessary to construct the permutation matrix P as computed by the companion factorization or condition number estimation subprogram. **ipvt** must have been preserved between the factorization or condition number call and the solve call.

b The right-hand-side vector b.

job Option flag:

job = 0 solve $Ax = b$
job ≠ 0 solve $A^*x = b$

Output

b The solution vector x overwrites the right-hand-side vector b.

Notes

These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

Example 1 Solve a system of linear equations $Ax = b$, where A is a 6-by-6 REAL*8 matrix stored in array A whose dimensions are 10 by 10, and b is a vector 6 elements long stored in an array B of dimension 10. The more robust, but slightly slower condition number estimation subprogram is used to factor the coefficient matrix.

```
INTEGER*4 LDA,N,IPVT(10),JOB
REAL*8    A(10,10),B(10),RCOND,WORK(10)
LDA = 10
N = 6
JOB = 0
CALL DGECO (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**.

Solve Tridiagonal Linear Equations — SGTSL/DGTSL/CGTSL/ZGTSL

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 SGTSV, DGTSV, CGTSV, and ZGTSV, except that the latter are designed for use on tridiagonal matrices that are known to be diagonally dominant. A matrix is diagonally dominant if

$$|a_{ii}| \geq \sum_{j \neq i} |a_{ij}|, \qquad i = 1, 2, \ldots, n.$$

When a matrix is diagonally dominant, partial pivoting is not required to ensure numerical stability. Eliminating partial pivoting admits vectorized and parallelized algorithms, so SGTSV, DGTSV, CGTSV, and ZGTSV are faster than these subprograms. For efficiency, use SGTSV, DGTSV, CGTSV, or ZGTSV 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 following tridiagonal matrix of order $n = 7$:

$$\begin{matrix} 11 & 12 & 0 & 0 & 0 & 0 & 0 \\ 21 & 22 & 23 & 0 & 0 & 0 & 0 \\ 0 & 32 & 33 & 34 & 0 & 0 & 0 \\ 0 & 0 & 43 & 44 & 45 & 0 & 0 \\ 0 & 0 & 0 & 54 & 55 & 56 & 0 \\ 0 & 0 & 0 & 0 & 65 & 66 & 67 \\ 0 & 0 & 0 & 0 & 0 & 76 & 77 \end{matrix}$$

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**$(i) = a_{i,i-1}$, $i = 2, 3, \ldots,$ **n**. On return, **dl** is destroyed, including **dl**(1).

d Array containing the principal diagonal of the tridiagonal matrix, **d**$(i) = a_{ii}$, $i = 1, 2, \ldots,$ **n**. On return, **d** is destroyed.

du Array containing the superdiagonal of the tridiagonal matrix, **du**$(i) = a_{i,i+1}$, $i = 1, 2, \ldots,$ **n**-1. On return, **du** is destroyed, including **du**(**n**).

b The right-hand-side vector b.

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 ≠ 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
```

Solve Tridiagonal Linear Equations — SGTSV/DGTSV/CGTSV/ZGTSV

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

$$|a_{ii}| \geq \sum_{j \neq i} |a_{ij}|, \qquad i = 1, 2, \ldots, n.$$

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 $n = 7$:

$$\begin{matrix} 11 & 12 & 0 & 0 & 0 & 0 & 0 \\ 21 & 22 & 23 & 0 & 0 & 0 & 0 \\ 0 & 32 & 33 & 34 & 0 & 0 & 0 \\ 0 & 0 & 43 & 44 & 45 & 0 & 0 \\ 0 & 0 & 0 & 54 & 55 & 56 & 0 \\ 0 & 0 & 0 & 0 & 65 & 66 & 67 \\ 0 & 0 & 0 & 0 & 0 & 76 & 77 \end{matrix}$$

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 SGTSV (n, dl, d, du, b, ier)

INTEGER*4  n, ier
REAL*8     dl(n), d(n), du(n), b(n)
CALL DGTSV (n, dl, d, du, b, ier)
```

```
INTEGER*4  n, ier
COMPLEX*8  dl(n), d(n), du(n), b(n)
CALL CGTSV (n, dl, d, du, b, ier)
INTEGER*4    n, ier
COMPLEX*16 dl(n), d(n), du(n), b(n)
CALL ZGTSV (n, dl, d, du, b, ier)
```

VECLIB8:

```
INTEGER*8 n, ier
REAL*8       dl(n), d(n), du(n), b(n)
CALL SGTSV (n, dl, d, du, b, ier)

INTEGER*8    n, ier
COMPLEX*16 dl(n), d(n), du(n), b(n)
CALL CGTSV (n, dl, d, du, b, ier)
```

Input

n The order of matrix A, **n** > 0.

dl Array containing the subdiagonal of the tridiagonal matrix, **dl**(i) = $a_{i,i-1}$, $i = 2, 3, \ldots, n$. On return, **dl** is destroyed, including **dl**(1).

d Array containing the principal diagonal of the tridiagonal matrix, **d**(i) = a_{ii}, $i = 1, 2, \ldots, n$. On return, **d** is destroyed.

du Array containing the superdiagonal of the tridiagonal matrix, **du**(i) = $a_{i,i+1}$, $i = 1, 2, \ldots, n$-1. On return, **du** is destroyed, including **du**(n).

b The right-hand-side vector b.

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** ≤ 0.
ier = –2	A divide-by-zero was encountered; not available in all implementations.

Notes

These subprograms have different functionality and usage than those with the same names in CONVEX LAPACK. Be sure to load VECLIB before LAPACK if you want these subroutines and use both libraries.

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 DGTSV (N,DL,D,DU,B,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
END IF
```

Purpose

These subprograms compute the Cholesky factorization and estimate the condition number of an n-by-n positive definite band matrix A stored in a two-dimensional array. A matrix A is positive definite if and only if it is Hermitian; that is, A is equal to A^*, its conjugate transpose, and the quadratic form x^*Ax is positive for all nonzero vectors x. (The conjugate transpose of a real matrix or vector is simply the transpose.)

A positive definite band matrix is a positive definite matrix whose nonzero elements all are fairly near the principal diagonal. Specifically, $a_{ij} = 0$ if $|i-j| > kd$ for some integer kd. The smallest such kd for a given matrix is called the half bandwidth, and $2kd+1$ is called the total bandwidth.

Tridiagonal matrices are the special case $kd = 1$. They can be handled more efficiently by the subprograms SPTSL, 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 $0 \le j-i \le kd$, then a_{ij} is stored in **ab**($\mathbf{kd}+1+i-j,j$). Columns of the upper triangle of A are stored in the columns of **ab**, and diagonals of the upper triangle of A are stored in the rows of **ab**.

ldab The leading dimension of array **ab** as declared in the calling program unit, with **ldab** $\ge$ **kd**+1.

n The order of matrix A, **n** > 0.

kd The half bandwidth of A, i.e., the number of diagonals above the principal diagonal in the band, $0 \le$ **kd** $<$ **n**.

Working storage

work An array of size **n**, used for work space.

Output

ab The Cholesky factor R replaces the input matrix. The factorization is not complete if **ier** is nonzero. **ab** must be preserved between the condition number estimation call and any solve or determinant call.

rcond An estimate of the reciprocal condition number, $1/\kappa(A)$, if **ier** is zero; unchanged from its input value if **ier** is nonzero. If **ier** is zero and **rcond** is so small that the logical expression

1.0 + rcond .EQ. 1.0

is true, A can be regarded as singular to working precision.

ier Status response:

ier = 0	Normal return—factorization complete.
ier = k ≠ 0	The leading submatrix of order k is not computationally positive definite, possibly because of roundoff error.

Notes

These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

Example Factor the 7-by-7 REAL*8 positive definite band matrix A whose half bandwidth is 2 and whose upper triangle is stored in the upper triangle of array AB whose dimensions are 5 by 10, and estimate its reciprocal condition number.

```
INTEGER*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
```

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 **ldab** $\geq$ **n**+1.

n The order of matrix A, **n** ≥ 0.

kd The half bandwidth of A, i.e., the number of diagonals above the principal diagonal in the band, $0 \leq$ **kd** $<$ **n**.

Output

det The determinant of A, in the form $\det(A) = \mathbf{det}(1) \times 10^{\mathbf{det}(2)}$. This expression may underflow or overflow if evaluated; on the CONVEX supercomputer, underflows automatically flush to zero, but overflows normally terminate execution. For REAL*4 and COMPLEX*8, overflow cannot occur if **det**(2) ≤ 37. For REAL*8 and COMPLEX*16, overflow cannot occur if **det**(2) ≤ 306. If evaluation is safe, an efficient way to do it is with the statement

$$\det(A) = \mathbf{det(1) * 10.0 ** INT(det(2))}$$

The value stored in **det(2)** is an integer in REAL form. **det(1)** is normalized so that **det(1)** = 0 or 1 $\leq$ **det**(1) $<$ 10.

Notes

These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

It is almost never necessary to compute the determinant of a matrix. While it is true that papers and reference books make extensive use of the notation "$\det(A) \neq 0$" to mean "A is nonsingular," 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
```

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 $0 \le j-i \le kd$, 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** ≥ **kd**+1.

n The order of matrix A, **n** > 0.

kd The half bandwidth of A, i.e., the number of diagonals above the principal diagonal in the band, 0 ≤ **kd** < **n**.

Output

ab The Cholesky factor R replaces the input matrix. The factorization is not complete if **ier** is nonzero. **ab** must be preserved between the condition number estimation call and any solve or determinant call.

ier Status response:

ier = 0	Normal return—factorization complete.
ier = k ≠ 0	The leading submatrix of order k is not computationally positive definite, possibly because of roundoff error.

Notes

These subprograms are usage-compatible with the standard LINPACK subprograms with the same names.

Example

Factor the 7-by-7 REAL*8 positive definite band matrix A whose half bandwidth is 2 and whose upper triangle is stored in the upper triangle of array AB whose dimensions are 5 by 10, and estimate its reciprocal condition number.

```
INTEGER*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
```

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 **ldab** ≥ **kd**+1.

n The order of matrix A, **n** ≥ 0.

kd The half bandwidth of A, i.e., the number of diagonals above the principal diagonal in the band, 0 ≤ **kd** < **n**.

b The right-hand-side vector b.

Output

b The solution vector x overwrites the right-hand-side vector b.

Notes

These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

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**.

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** ≥ max(**n**,1).

n The order of matrix A, **n** ≥ 0.

Working storage

work An array of size **n**, used for work space.

Output

a The Cholesky factor R replaces the input matrix A in the upper triangle of **a**. The strict lower triangle of **a** is unchanged. The factorization is not complete if **ier** is nonzero. **a** must be preserved between the condition number estimation call and any solve, determinant, or inverse call.

rcond An estimate of the reciprocal condition number, $1/\kappa(A)$, if **ier** is zero; unchanged from its input value if **ier** is nonzero. If **ier** is zero and **rcond** is so small that the logical expression

1.0 + rcond .EQ. 1.0

is true, then A can be regarded as singular to working precision.

ier Status response:

ier = 0	Normal return—factorization complete.
ier = k ≠ 0	The leading submatrix of order k is not computationally positive definite, possibly because of roundoff error.

Notes These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

Example Factor the 6-by-6 REAL*8 positive definite matrix A whose upper triangle is stored in the upper triangle of array A whose dimensions are 10 by 10, and estimate its reciprocal condition number.

```
INTEGER*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
```

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** ≥ max(**n**,1).

n The order of matrix A, **n** ≥ 0.

job Option flag:

job = 1	compute only A^{-1}
job = 10	compute only det(A)
job = 11	compute both A^{-1} and det(A)

Output

a Unchanged if A^{-1} is not requested. Otherwise, the upper triangle of A^{-1} overwrites the Cholesky factor of the coefficient matrix. The strict lower triangle of **a** is never changed.

det Not referenced if the determinant is not requested. Otherwise, the determinant of A, in the form det(A) = **det(1)** × $10^{\mathbf{det(2)}}$. This expression may underflow or overflow if evaluated; on the CONVEX supercomputer, underflows automatically flush to zero, but overflows normally terminate execution. For REAL*4 and COMPLEX*8, overflow cannot occur if **det(2)** ≤ 37. For REAL*8 and COMPLEX*16, overflow cannot occur if **det(2)** ≤ 306. If evaluation is safe, an efficient way to do it is with the statement

det(A) = **det(1) * 10.0 ** INT(det(2))**

Refer to "Example 2."

The value stored in **det(2)** is an integer in REAL form. **det(1)** is normalized so that **det(1)** = 0 or 1 ≤ **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) ≠ 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
```

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 itself are magnified greatly in the solution. A set of companion subprograms computes the Cholesky factorization of a matrix and also estimates its condition number. These companion subprograms provide a more reliable indication of singularity. The small amount of additional time they require is usually worthwhile, especially when developing a program or encountering stability or convergence problems.

The triangular factors may be used to solve a system of linear equations, $Ax = b$, by successively solving $R^*(Rx) = b$. The determinant of A can be computed as $\det(A) = \det(R)^2$. The inverse of A may be formed as $A^{-1} = R^{-1}R^{-*}$, where R^{-*} is the conjugate transpose of the inverse of R. These operations are performed by a set of companion 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** ≥ max(**n**,1).

n The order of matrix A, **n** ≥ 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 ≠ 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
```

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 CPOSL (a, lda, n, b)

INTEGER*4    lda, n
COMPLEX*16 a(lda, n), b(n)
CALL ZPOSL (a, lda, n, b)
```

VECLIB8:

```
INTEGER*8 lda, n
REAL*8       a(lda, n), b(n)
CALL SPOSL (a, lda, n, b)
```

```
INTEGER*8    lda, n
COMPLEX*16 a(lda, n), b(n)
CALL CPOSL (a, lda, n, b)
```

Input

a Array containing the Cholesky factor R of the **n**-by-**n** positive definite coefficient matrix A in its upper triangle, as computed by the companion factorization or condition number estimation subprogram. The upper triangle of **a** must have been preserved between the factorization or condition number call and the solve call.

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

n The order of matrix A, **n** ≥ 0.

b The right-hand-side vector b.

Output

b The solution vector x overwrites the right-hand-side vector b.

Notes

These subprograms are usage compatible with the standard LINPACK subprograms with the same names.

Example

Solve a system of linear equations $Ax = b$, where A is a 6-by-6 REAL*8 positive definite matrix whose upper triangle is stored in array A whose dimensions are 10 by 10, and where b is a vector 6 elements long stored in an array B of dimension 10. The more robust, but slightly slower condition number estimation subprogram is used to factor the coefficient matrix.

```
INTEGER*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 DPOSL (A,LDA,N,B)
ELSE
   handle singular matrix
END IF
```

If the coefficient matrix A is determined to be positive definite and nonsingular, the solution vector x overwrites the right-hand-side b in array **b**.

SPTSL/.../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 $n = 7$:

$$\begin{matrix} 11 & 12 & 0 & 0 & 0 & 0 & 0 \\ 12 & 22 & 23 & 0 & 0 & 0 & 0 \\ 0 & 23 & 33 & 34 & 0 & 0 & 0 \\ 0 & 0 & 34 & 44 & 45 & 0 & 0 \\ 0 & 0 & 0 & 45 & 55 & 56 & 0 \\ 0 & 0 & 0 & 0 & 56 & 66 & 67 \\ 0 & 0 & 0 & 0 & 0 & 67 & 77 \end{matrix}$$

then the principal diagonal is stored in array **d** and the superdiagonal is stored in array **e** as follows:

i	d(i)	e(i)
1	11	12
2	22	23
3	33	34
4	44	45
5	55	56
6	66	67
7	77	*

The asterisk represents an element that is not referenced.

Usage

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, $\mathbf{d}(i) = a_{ii}$, $i = 1, 2, \ldots, \mathbf{n}$. For CPTSL and ZPTSL, only the real parts of **d** are used. On return, **d** is destroyed.

e Array containing the superdiagonal of the tridiagonal matrix, $\mathbf{e}(i) = a_{i,i+1}$, $i = 1, 2, \ldots, \mathbf{n}$-1.

b The right-hand-side vector b.

Output

b The solution vector x overwrites the right-hand-side vector b.

Notes

These subprograms are usage-compatible with the standard LINPACK subprograms with the same names.

Caution is necessary since these subprograms do not detect error conditions. An inaccurate solution may be computed or a division by zero may occur if the matrix is indefinite or singular.

Example

Solve a system of linear equations $Ax = b$, where A is a 7-by-7 REAL*8 positive definite tridiagonal matrix. The principal diagonal of A is stored in array D, and the superdiagonal is stored in array E. b is a vector 7 elements long stored in an array B.

```
INTEGER*4 N
REAL*8     D(10),E(10),B(10)
N = 7
CALL DPTSL (N,D,E,B)
```

LINPACK Subprograms not in the *CONVEX VECLIB User's Guide*

Although VECLIB includes all LINPACK subprograms, the following nonoptimized subprograms are not documented in the *CONVEX VECLIB User's Guide*. The *LINPACK Users' Guide*, included in the VECLIB documentation set, documents these subprograms.

Table 4-5: LINPACK Subprograms not in the *VECLIB User's 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
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

Name	Function
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
CSIDI	Determinant, Inverse, and Inertia of a Symmetric Indefinite Matrix
ZSIDI	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

LINPACK Subprograms not in the *VECLIB User's Guide* **Continued**

Name	Function
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
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

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. Although all EISPACK subprograms are included in VECLIB, only upgraded ones are described in this chapter. Table 5-1 at the end of this chapter lists the subprograms that are included in VECLIB but not documented in the *CONVEX VECLIB User's Guide*. You may find information for these subprograms in the *EISPACK Guide* and the *EISPACK Guide Extension* included in the VECLIB documentation set.

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 *CONVEX LAPACK User's Guide*. We recommend that you use LAPACK subprograms rather than EISPACK subprograms in new programs. Future optimization efforts will be directed to LAPACK rather than EISPACK.

This chapter explains how to use 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, $Ax = \lambda x$

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.

Supplemental Reading

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

Parlett, B.N. *The Symmetric Eigenproblem.* Englewood Cliffs, NJ: Prentice-Hall, Inc. 1980.

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

Wilkinson, J.H. *The Algebraic Eigenproblem.* New York: Oxford University Press. 1965.

Subprogram Descriptions

Eigenvalues and Eigenvectors of a Real Symmetric Matrix
RS 5-3

Eigenvalues and Eigenvectors of a Real Symmetric Tridiagonal Matrix
TQL2 5-5

Eigenvalues of a Real Symmetric Tridiagonal Matrix
TQLRAT 5-8

Reduce a Real Symmetric Matrix to Real Symmetric Tridiagonal Form
TRED1 5-10

Reduce a Real Symmetric Matrix to Real Symmetric Tridiagonal Form
TRED2 5-12

Eigenvalues and Eigenvectors of a Real Symmetric Matrix — RS

Purpose

This subprogram computes eigenvalues or eigenvalues and eigenvectors of a full real symmetric n-by-n matrix A. Specifically, given A, this subprogram determines n scalars, λ_i, $i = 1, 2, ..., n$, 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:

```
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)
```

VECLIB8:

```
INTEGER*8 ldax, n, job, ier
REAL*8    a(ldax, n), w(n), x(ldax, n), work1(n), work2(n)
CALL RS (ldax, n, a, w, job, x, work1, work2, ier)
```

Input

ldax The leading dimension of arrays **a** and **x** as declared in the calling program unit, with **ldax** ≥ max(**n**,1).

n The order of matrix A, **n** ≥ 0.

a Array containing the diagonal and lower triangle of the **n**-by-**n** matrix A. Elements in the strict upper triangle are not referenced.

job Option flag:

- **job** = 0 compute eigenvalues only
- **job** ≠ 0 compute eigenvalues and eigenvectors

Working storage

work1 Array of size **n**, used for work space.

work2 Array of size **n**, used for work space.

Output

a The lower triangle is destroyed if **job** = 0. Not modified if **job** ≠ 0.

w The eigenvalues λ_i of A in ascending order if **ier** = 0 is returned.

x Not referenced if eigenvectors are not requested. In this case, **x** can be a dummy variable. Otherwise, eigenvectors of A if **ier** = 0 is returned. The j-th column of **x** contains the eigenvector x_j of A corresponding to the eigenvalue in **w**(j), j = 1, 2, ..., **n**. Eigenvectors are normalized to have Euclidean length = 1.

ier Status response:

ier = 0	Normal return.
ier = k, $1 \le k \le$ **n**	if calculation of the k-th eigenvalue failed to converge. **w**(1), **w**(2), ..., **w**(k-1) are eigenvalues, but are not necessarily the smallest and are not necessarily sorted. If eigenvectors are requested, the first k-1 columns of **x** are eigenvectors corresponding to the first k-1 elements of **w**.
ier = 10**n**	if **n** > **ldax**. No eigenvalues or eigenvectors are returned.

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
```

Eigenvalues and Eigenvectors of a Real Symmetric Matrix — TQL2

Purpose

This subprogram computes eigenvalues and eigenvectors of a tridiagonal real symmetric n-by-n matrix. Eigenvalues and eigenvectors of a full real symmetric matrix can also be computed by this subprogram if TRED2 has been used to reduce the full matrix to tridiagonal form.

Specifically, given a tridiagonal real symmetric matrix A or output of TRED2 applied to a full real symmetric matrix A, this subprogram determines scalars, λ_i, $i = 1, 2, \ldots, n$, and nonzero vectors, x_i, $i = 1, 2, \ldots, n$, such that

$$Ax_i = \lambda_i x_i.$$

Matrix Storage

The following example illustrates the storage of symmetric tridiagonal matrices. Consider the following symmetric tridiagonal matrix of order $n = 7$:

$$\begin{matrix} 11 & 21 & 0 & 0 & 0 & 0 & 0 \\ 21 & 22 & 32 & 0 & 0 & 0 & 0 \\ 0 & 32 & 33 & 43 & 0 & 0 & 0 \\ 0 & 0 & 43 & 44 & 54 & 0 & 0 \\ 0 & 0 & 0 & 54 & 55 & 65 & 0 \\ 0 & 0 & 0 & 0 & 65 & 66 & 76 \\ 0 & 0 & 0 & 0 & 0 & 76 & 77 \end{matrix}$$

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 **ldx** ≥ max(n,1).

n — The order of matrix A, **n** ≥ 0.

d — Array containing the diagonal elements of the **n**-by-**n** symmetric tridiagonal matrix A.

	e	Array containing the subdiagonal elements of A in elements **e**(2) through **e**(**n**). **e**(1) is not used as input.
	x	If A is a tridiagonal matrix, **x** must be initialized to the **n**-by-**n** identity matrix. If A is a full matrix, **x** contains the transformation matrix produced by TRED2 in reducing the full matrix to tridiagonal form.
Output	**d**	Eigenvalues, λ_i, $i = 1, 2, \ldots, n$, of A overwrite the input if **ier** = 0 is returned. Eigenvalues have been sorted into ascending order.
	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), $j = 1, 2, \ldots,$ **n**. Eigenvectors are normalized to have Euclidean length = 1.
	ier	Status response:

ier = 0	Normal return.
ier = k, $1 \le k \le$ **n**	if calculation of the k-th eigenvalue failed to converge. **d**(1), **d**(2), ..., **d**(k-1) are eigenvalues, but are not necessarily the smallest and are not necessarily sorted. The first k-1 columns of **x** are eigenvectors corresponding to the first k-1 elements of **d**.

Notes

This subprogram is usage compatible with the standard 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
```

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, $i = 1, 2, ..., n$, for which there exist corresponding nonzero vectors, x_i, $i = 1, 2, ..., n$, 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, $i = 1, 2, ..., n$, 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
```

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** ≥ max(**n**,1).

n The order of matrix A, **n** ≥ 0.

a Array containing the diagonal and lower triangle of the **n**-by-**n** matrix A. Elements in the strict upper triangle are not referenced.

Output

a The diagonal and lower triangle are destroyed.

d Array containing diagonal elements of the tridiagonal matrix T.

e Array containing the subdiagonal elements of T in elements **e**(2) through **e**(**n**). **e**(1) = 0.

e2 Array containing squares of subdiagonal elements of T in elements **e**(2) through **e**(**n**). **e2**(1) = 0.

Notes

This subprogram is usage-compatible with the standard 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
```

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

$$X^T AX = T.$$

Matrix Storage

Because the upper triangle of A may be obtained from the lower triangle, you need only provide the lower triangle of A, in a two-dimensional array large enough to hold the entire matrix. The upper triangle of the array is not referenced.

Usage

VECLIB:

```
INTEGER*4 ldax, n
REAL*8    a(ldax, n), d(n), e(n), x(ldax, n)
CALL TRED2 (ldax, n, a, d, e, x)
```

VECLIB8:

```
INTEGER*8 ldax, n
REAL*8    a(ldax, n), d(n), e(n), x(ldax, n)
CALL TRED2 (ldax, n, a, d, e, x)
```

Input

ldax The leading dimension of arrays **a** and **x** as declared in the calling program unit, with **ldax** ≥ max(n,1).

n The order of matrix A, **n** ≥ 0.

a Array containing the diagonal and lower triangle of the **n**-by-**n** matrix A. Elements in the strict upper triangle are not referenced.

Output

d Array containing diagonal elements of the tridiagonal matrix T.

e Array containing subdiagonal elements of T in elements **e**(2) through **e**(n). **e**(1) = 0.

x The transformation matrix X that reduces A to tridiagonal form.

Notes

This subprogram is usage-compatible with the standard 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
```

EISPACK Subprograms not in the *CONVEX VECLIB User's Guide*

Although VECLIB includes all EISPACK subprograms, the following nonoptimized subprograms are not documented in the *CONVEX VECLIB User's Guide*. The *EISPACK Guide* and the *EISPACK Guide Extension*, included in the VECLIB documentation set, document these subprograms.

Table 5-1: EISPACK Subprograms not in the *VECLIB User's Guide*

Name	Function
BAKVEC	Back Transform Eigenvectors following FIGI
BALANC	Balance a Real General Matrix
BALBAK	Back Transform Eigenvectors following BALANC
BANDR	Reduce a Real Symmetric Band Matrix to Real Symmetric Tridiagonal Form
BANDV	Determine Some Eigenvectors of a Real Symmetric Band Matrix
BISECT	Determine Some Eigenvectors of a Real Symmetric Tridiagonal Matrix
BQR	Determine Some Eigenvalues of a Real Symmetric Band Matrix
CBABK2	Back Transform Eigenvectors following CBAL
CBAL	Balance a Complex General Matrix
CG	Determine Eigenvalues/vectors of a Complex General Matrix
CH	Determine Eigenvalues/vectors of a Complex Hermitian Matrix
CINVIT	Determine Some Eigenvectors of a Complex Upper Hessenberg Matrix
COMBAK	Back Transform Eigenvectors following COMHES
COMHES	Reduce a Complex General Matrix to Complex Upper Hessenberg Form
COMLR	Determine the Eigenvalues of a Complex Upper Hessenberg Matrix
COMLR2	Determine the Eigenvalues/vectors of a Complex Hessenberg Matrix
COMQR	Determine the Eigenvalues of a Complex Upper Hessenberg Matrix
COMQR2	Determine the Eigenvalues/vectors of a Complex Upper Hessenberg Matrix
CORTB	Back Transform Eigenvectors following CORTH
CORTH	Reduce a Complex General Matrix to Complex Upper Hessenberg Form
ELMBAK	Back Transform Eigenvectors following ELMHES
ELMHES	Reduce a Real General Matrix to Real Upper Hessenberg Form
ELTRAN	Accumulate the Transformations in the Reduction by ELMHES
FIGI	Transform a Real Non-symmetric Tridiagonal Matrix to Real Symmetric Form
FIGI2	Transform a Real Non-symmetric Tridiagonal Matrix to Real Symmetric Form
HQR	Determine the Eigenvalues of a Real Upper Hessenberg Matrix
HQR2	Determine the Eigenvalues/vectors of a Real Upper Hessenberg Matrix
HTRIB3	Back Transform Eigenvectors following HTRID3
HTRIBK	Back Transform Eigenvectors following HTRIDI
HTRID3	Reduce a Complex Hermitian Matrix to Real Symmetric Tridiagonal Form
HTRIDI	Reduce a Complex Hermitian Matrix to Real Symmetric Tridiagonal Form
IMTQL1	Determine the Eigenvalues of a Real Symmetric Tridiagonal Matrix
IMTQL2	Determine the Eigenvalues/vectors of a Real Symmetric Tridiagonal Matrix
IMTQLV	Determine the Eigenvalues of a Real Symmetric Tridiagonal Matrix
INVIT	Determine Some Eigenvectors of a Real Upper Hessenberg Matrix
MINFIT	Solve a Least Squares Problem with a Real Rectangular Coefficient Matrix

Continued EISPACK Subprograms not in the *VECLIB User's Guide*

Name	Function
ORTBAK	Back Transform Eigenvectors following ORTHES
ORTHES	Reduce a Real General Matrix to Real Upper Hessenberg Form
ORTRAN	Accumulate the Transformations in the Reduction by ORTHES
QZHES	Partially Reduce a Real General Generalized Eigenproblem
QZIT	Complete the Reduction of a Real General Generalized Eigenproblem
QZVAL	Determine the Eigenvalues of a Reduced Real General Generalized Eigenproblem
QZVEC	Determine the Eigenvectors of a Reduced Real General Generalized Eigenproblem
RATQR	Determine Some Extreme Eigenvalues of a Real Symmetric Tridiagonal Matrix
REBAK	Back Transform Eigenvectors following REDUC or REDUC2
REBAKB	Back Transform Eigenvectors following REDUC2
REDUC	Reduce a Real Symmetric Generalized Eigenproblem to Standard Form
REDUC2	Reduce a Real Symmetric Generalized Eigenproblem to Standard Form
RG	Determine the Eigenvalues/vectors of a Real General Matrix
RGG	Determine the Eigenvalues/vectors of a Real General Generalized Eigenproblem
RSB	Determine the Eigenvalues/vectors of a Real Symmetric Band Matrix
RSG	Determine the Eigenvalues/vectors of a Real Symmetric Generalized Eigenproblem
RSGAB	Determine the Eigenvalues/vectors of a Real Symmetric Generalized Eigenproblem
RSGBA	Determine the Eigenvalues/vectors of a Real Symmetric Generalized Eigenproblem
RSM	Determine All Eigenvalues and Some Eigenvectors of a Real Symmetric Matrix
RSP	Determine the Eigenvalues/vectors of a Real Symmetric Packed Matrix
RST	Determine the Eigenvalues/vectors of a Real Symmetric Tridiagonal Matrix
RT	Determine the Eigenvalues/vectors of a Real Tridiagonal Matrix
SVD	Compute the Singular Value Decomposition of a Real Rectangular Matrix
TINVIT	Determine Some Eigenvectors of a Real Symmetric Tridiagonal Matrix
TQL1	Determine the Eigenvalues of a Real Symmetric Tridiagonal Matrix
TRBAK1	Back Transform Eigenvectors following TRED1
TRBAK3	Back Transform Eigenvectors following TRED3
TRED3	Reduce a Real Symmetric Matrix to Real Symmetric Tridiagonal Form
TRIDIB	Determine Some Eigenvalues of a Real Symmetric Tridiagonal Matrix
TSTURM	Determine Some Eigenvalues/vectors of a Real Symmetric Tridiagonal Matrix

6

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 CONVEX 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

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 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 6-2.

Figure 6–2: Column Pointer, Row Index Sparse Matrix Representation

```
COLSTR = 1  4  7  9  11 12 13

ROWIND = 1  3  4  2  3  5  3  5  4  5  5  6  -
MXVALU = 11 31 41 22 32 52 33 53 44 54 55 66
```

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.

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. 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, solution execution time increases 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.

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 users to avoid using this option routinely for indefinite problems unless they have already verified its effectiveness on their particular class of problems. We also recommend that matrix condition numbers be computed (refer to Chapter 4 for more details) to assure that sufficient accuracy can be obtained in the solution.

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, 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 $-1000 \leq$ **ier** ≤ -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. 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, the user 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.

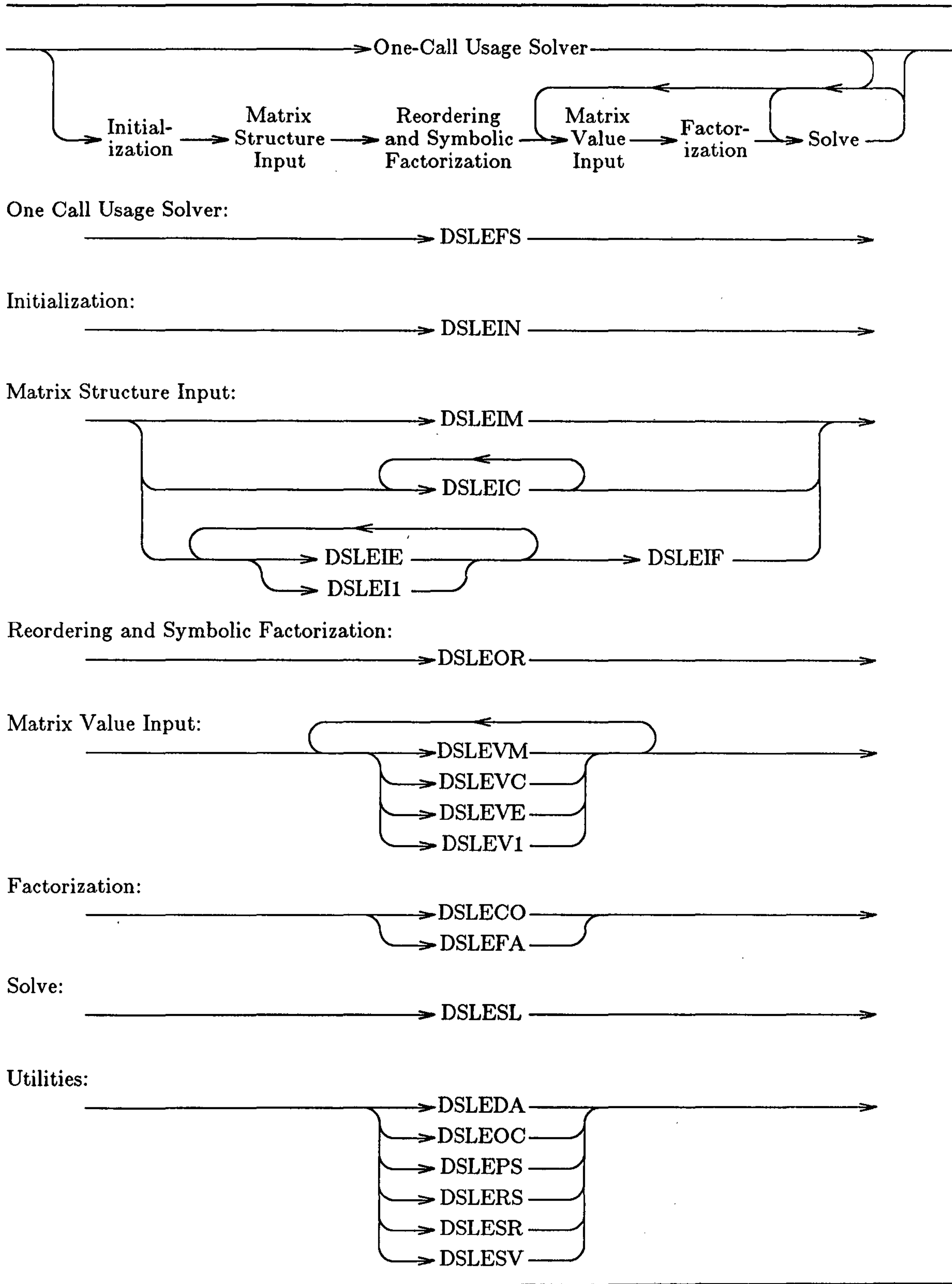
Paths of Control

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 Figure 6-3 and below:

```
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 6–3: Paths of Control



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),PVTTOL,VALUE

C     ----------------------------------------
C     ... INITIALIZE THE SPARSE MATRIX PACKAGE
C     ----------------------------------------

      CALL DSLEIN (NEQNS,1,6,GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000

C     ------------------------------
C     ... INPUT THE MATRIX STRUCTURE
C     ------------------------------

      DO 100 K = 1, NNZERO
          I = IROW(K)
          J = JCOL(K)
          CALL DSLEI1 (I,J,GLOBAL,IER)
          IF ( IER .NE. 0 ) GO TO 8000
  100 CONTINUE

      CALL DSLEIF (GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000

C     ---------------------
C     ... REORDER THE MATRIX
C     ---------------------

      CALL DSLEOR (0,GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000

C     ----------------------
C     ... INPUT MATRIX VALUES
C     ----------------------

      DO 200 K = 1, NNZERO
          I     = IROW(K)
          J     = JCOL(K)
          VALUE = MXVALU(K)
          CALL DSLEV1 (I,J,VALUE,GLOBAL,IER)
          IF ( IER .NE. 0 ) GO TO 8000
  200 CONTINUE
```

```
C     -----------------------------------------------------
C     ... FACTOR THE MATRIX AND ESTIMATE ITS CONDITION NUMBER
C     -----------------------------------------------------

      PVTTOL = 0.1D0
      CALL DSLECO (PVTTOL,COND,INRTIA,GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000

C     ------------------------------------
C     ... SOLVE FOR A GIVEN RIGHT HAND SIDE
C     ------------------------------------

      CALL DSLESL (1,RHS,NEQNS,GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000

C     ----------------------------------------
C     ... USE THE SOLUTION STORED IN ARRAY RHS
C     ----------------------------------------

       .
       .
       .

C     --------------
C     ... ERROR TRAP
C     --------------

 8000  ..........
```

Supplemental Reading

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.

Subprogram Descriptions

One-Call Usage
DSLEFS 6-10

Initialize Sparse Linear Equations
DSLEIN 6-13

Matrix Structure Input by Single Entry
DSLEI1 6-14

Matrix Structure Input by Column
DSLEIC 6-15

Matrix Structure Input by Finite Element
DSLEIE 6-16

Matrix Structure Input by Matrix
DSLEIM 6-17

End of Matrix Structure Input
DSLEIF 6-19

Reordering and Symbolic Factorization
DSLEOR 6-20

Matrix Value Input by Single Entry
DSLEV1 6-21

Matrix Value Input by Column
DSLEVC 6-22

Matrix Value Input by Finite Element
DSLEVE 6-24

Matrix Value Input by Matrix
DSLEVM 6-25

Numeric Factorization and Condition Number Estimation
DSLECO 6-27

Numeric Factorization
DSLEFA 6-28

Solve
DSLESL 6-29

Deallocate Working Storage
DSLEDA 6-30

Output Control
DSLEOC 6-31

Print Statistics
DSLEPS 6-32

Restore Problem State from a Savefile
DSLERS 6-33

Retrieve Runtime Statistics
DSLESR 6-34

Save Problem State to a Savefile
DSLESV 6-35

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 ≤ **pvttol** ≤ 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 j of the matrix. All of the nonzeros for column j are found, in ascending order, in **rowind(colstr**(j)**)**, **rowind(colstr**(j)+1), ..., **rowind(colstr**($j+1$)–1).

colstr(neqns+1) must be set to one greater than the total number of nonzeros, **nnzero**, in the lower triangular part of the matrix.

rowind List of row indices for all nonzeros, in ascending order within each column, in the lower triangle part of the matrix A.

value List of values corresponding in position to the indices in **rowind**.

nrhs Number of right-hand sides.

ldrhs The leading dimension of array **rhs** as declared in the calling program unit, with **ldrhs** ≥ **neqns**.

Updated

rhs On input, **rhs** contains the **nrhs** right-hand sides. On output, **rhs** has been overwritten with the computed solutions.

Output

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

$$A = \begin{matrix} 4.0 & 0.0 & 1.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 4.0 & 1.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 1.0 & 4.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 4.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 1.0 & 1.0 & 4.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 1.0 \end{matrix} \qquad b = \begin{matrix} 1.0 \\ 0.0 \\ 1.0 \\ 1.0 \\ 0.0 \\ 0.0 \end{matrix}$$

A supernode relaxation factor of 0 (no extra fill allowed) and a pivoting tolerance of 0.1 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
```

Initialize Sparse Linear Equations — DSLEIN

Purpose

This subprogram provides the necessary information to begin processing with the sparse linear equation solution package.

Usage

VECLIB:

INTEGER*4 neqns, msglvl, output, ier
REAL*8 global(150)
CALL DSLEIN (neqns, msglvl, output, global, ier)

Input

neqns Order of matrix (number of degrees of freedom); **neqns** > 0.

msglvl Message level for printable output:

msglvl ≤ 0	Suppress all output.
msglvl = 1	Error messages and summary statistics.
msglvl = 2	More complete statistics.
msglvl = 3	First stage of debugging output.
msglvl = 4	Complete debugging output.

output FORTRAN logical unit number to which all printable output will be written.

Output

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

ier Status response:

ier = 0	Normal return.
ier = -101	Error in dynamic storage allocation.
ier = -102	**neqns** ≤ 0.

Example

Prepare to solve a system of equations of order 10,000. Obtain error messages and standard minimal output on FORTRAN logical unit 6.

```
INTEGER*4 NEQNS,IER
REAL*8    GLOBAL(150)
NEQNS = 10000
CALL DSLEIN (NEQNS,1,6,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

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:

```
INTEGER*4 irow, jcol, ier
REAL*8    global(150)
CALL DSLEI1 (irow, jcol, global, ier)
```

Input

irow Row index of the nonzero entry; **jcol** ≤ **irow** ≤ **neqns**, where **neqns** is the number of equations specified in the call to DSLEIN. Input of diagonal entries is optional.

jcol Column index of the nonzero entry; 1 ≤ **jcol** ≤ **neqns**, where **neqns** is the number of equations specified in the call to DSLEIN.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = −100	Incorrect processing path; DSLEIN not called or matrix input already finished.
ier = −101	Error in dynamic storage allocation.
ier = −104	Illegal value for **jcol**.
ier = −105	Illegal value for **irow**.

Notes

Calls to DSLEI1 and DSLEIE can be intermixed. DSLEIC and DSLEIM cannot be used if DSLEI1 or DSLEIE are used.

Example

Add the entry in row 3035, column 1024 to the list of nonzeros in the matrix.

```
INTEGER*4 I,J,IER
REAL*8    GLOBAL(150)
I = 3035
J = 1024
CALL DSLEI1 (I,J,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

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:**

INTEGER*4 jcol, nzcol, jrowin(nzcol), ier
REAL*8 global(150)
CALL DSLEIC (jcol, nzcol, jrowin, global, ier)

Input

jcol Column index; 1 ≤ **jcol** ≤ **neqns**. Columns must be presented in order from 1 to **neqns**, except that columns with no entries can be skipped.

nzcol Number of nonzeros in column **jcol** of the matrix; **nzcol** ≥ 0.

jrowin List of row indices for all nonzeros, in ascending order, in the lower triangular part of column **jcol** of the matrix; **jcol** ≤ **jrowin**(1) < **jrowin**(2) < ... < **jrowin(nzcol)** ≤ **neqns**. Input of diagonal entries is optional

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = −100	Incorrect processing path; DSLEIN not called or matrix input already finished.
ier = −101	Error in dynamic storage allocation.
ier = −103	Illegal value for **nzcol.**
ier = −104	Illegal value for **jcol**, or out of order.
ier = −106	Illegal value for at least one entry in **jrowin** or entries out of order.

Notes The entire matrix structure must be input with DSLEIC if it is used. Its use is not compatible with DSELE1, DSLEIE, or DSLEIM.

If the matrix entries are available by column, using DSLEIC is more efficient than using DSLEI1.

Example 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
```

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:

```
INTEGER*4 nnode, nodlst(nnode), ier
REAL*8    global(150)
CALL DSLEIE (nnode, nodlst, global, ier)
```

Input

nnode Number of nodes in the finite element or clique.

nodlst List of nodes. All pairs (**nodlst**(i),**nodlst**(j)) in the lower triangle are added to the sparsity structure of the matrix.

Updated

nodlst The order of the values in **nodlst** may be changed.

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = –100	Incorrect processing path; DSLEIN not called or matrix input already finished.
ier = –101	Error in dynamic storage allocation.
ier = –107	Illegal value for **nnode**.
ier = –108	Illegal value for at least one entry in **nodlst**.

Notes

Calls to DSLEIE can be intermixed with calls to DSLEI1. DSLEIC and DSLEIM cannot be used if DSLEI1 or DSLEIE is used.

Example

Rows and columns 345, 346, 347 and 989 form a small dense submatrix of A. Add the positions consisting of all pairs of numbers from this set to the list of nonzeros in the matrix.

```
INTEGER*4 NNODE,NODLST(10),IER
REAL*8    GLOBAL(150)
NNODE = 4
NODLST(1) = 345
NODLST(2) = 346
NODLST(3) = 347
NODLST(4) = 989

CALL DSLEIE (NNODE,NODLST,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

Purpose

This subprogram specifies locations of all nonzeros in the lower triangle of the sparse matrix.

Usage

VECLIB:

```
INTEGER*4 neqns, nnzero, colstr(neqns+1), rowind(nnzero), ier
REAL*8    global(150)
CALL DSLEIM (colstr, rowind, global, ier)
```

Input

colstr — **colstr**(j) gives the index in **rowind** of the first nonzero in the lower triangular part of column j of the matrix. All of the nonzeros for column j are found, in ascending order, in **rowind(colstr(**j**))**, **rowind(colstr(**j**)**+1), ..., **rowind(colstr(**$j+1$**)–1)**.

colstr(neqns+1) must be set to one greater than the total number of nonzeros, **nnzero**, in the lower triangular part of the matrix.

rowind — List of row indices for all nonzeros, in ascending order within each column, in the lower triangle part of the matrix. Input of diagonal entries is optional.

Updated

global — Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier — Status response:

ier = 0	Normal return.
ier = –100	Incorrect processing path; DSLEIN not called or matrix input already finished.
ier = –101	Error in dynamic storage allocation.
ier = –106	Illegal value for at least one entry in **colstr** (not increasing or negative) or invalid entries in **rowind** (entries out of order within a column, or out of the lower triangle).

Notes

This is the most efficient mechanism for specifying the nonzero structure, but the entire matrix structure must be input with DSLEIM if it is used. Its use is not compatible with DSLEI1, DSLEIE, or DSLEIC.

Example 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
```

Purpose This subprogram specifies the end of structure input for the matrix. DSELIF is used only if routines DSLEI1 and/or DSLEIE were the mechanism by which the structure was input.

Usage **VECLIB:**

```
INTEGER*4 ier
REAL*8    global(150)
CALL DSLEIF (global, ier)
```

Updated **global** Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output **ier** Status response:

ier = 0	Normal return.
ier = –100	Incorrect processing path; DSLEIN not called or matrix input already finished.
ier = –101	Error in dynamic storage allocation.

Example The nonzero structure of a pair of finite element matrices was passed to the package using repeated calls to subroutine DSLEIE. Signal that no more nonzeros will be added to the matrix.

```
INTEGER*4  IER
REAL*8     GLOBAL(150)
CALL DSLEIF (GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

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:

```
INTEGER*4 maxzer, ier
REAL*8    global(150)
CALL DSLEOR (maxzer, global, ier)
```

Input

maxzer Supernodal relaxation parameter; **maxzer** ≥ 0. If **maxzer** > 0, the package allows additional fill for the purpose of forming columns with identical structure to increase the efficiency of the factorization. If **maxzer** $= 0$ no additional fill is allowed. Performance studies indicate that a value of 20 to 25 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:

ier = 0	Normal return.
ier = –200	Incorrect processing path; structure input not completed or value input subroutines already called.
ier = –201	Error in dynamic storage allocation.

Example

The sparse matrix has been communicated to the page using DSLEIN and DSLEI1, DSLEIE, and DSLEIF, and DSLEIC or DSLEIM. The next step is obtaining good reordering and performing symbolic factorization for the numeric factorization phase. The value of MAXZER = 0 is used so that no extra fill 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
```

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:

INTEGER*4 irow, jcol, ier
REAL*8 value, global(150)
CALL DSLEV1 (irow, jcol, value, global, ier)

Input

irow Row index of the nonzero entry, **jcol** ≤ **irow** ≤ **neqns**.

jcol Column index of the nonzero entry, 1 ≤ **jcol** ≤ **neqns**.

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; DSLEOR not called.
ier = –401	Error in dynamic storage allocation.
ier = –402	Subscript pair (**irow**, **jcol**) was not specified in structure input. No room for value.

Notes

Calls to DSLEV1, DSLEVC, DSLEVE, and DSLEVM can be intermixed.

Example

Store the value 4.523×10^{-5} as the nonzero entry in row 3035, column 1024 of the matrix.

```
INTEGER*4 I,J,IER
REAL*8    VALUE,GLOBAL(150)
I = 3035
J = 1024
VALUE = 4.523D-5
CALL DSLEV1 (I,J,VALUE,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

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 ≤ **jcol** ≤ **neqns**.

nzcol Number of nonzeros in the list of nonzeros for column **jcol** of the matrix; **nzcol** ≥ 0.

jrowin List of ascending order row indices for nonzeros in the lower triangular part of column **jcol** of the matrix; **jcol** ≤ **jrowin**(1) < **jrowin**(2) < ... < **jrowin**(**nzcol**) ≤ **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 values 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
```

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:**

```
INTEGER*4 nnode, ldelmx, nodlst(nnode), ier
REAL*8    elmmtx(ldelmx, nnode), global(150)
CALL DSLEVE (nnode, nodlst, elmmtx, ldelmx, global, ier)
```

Input

nnode Number of nodes in the finite element or clique.

nodlst List of nodes in element or clique. Values for all pairs (**nodlst**(i),**nodlst**(j)) are added to the values of the matrix.

elmmtx Array containing values to be added to the matrix. Only the lower triangle (including the diagonal) of **elmmtx** is referenced. The value in **elmmtx**(k,l) is added to the value in position (**nodlst**(k),**nodlst**(l)) in the sparse matrix.

ldelmx The leading dimension of array **elmmtx** as declared in the calling program unit, with **ldelmx** ≥ **nnode**.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = –400	Incorrect processing path; DSLEOR not called.
ier = –401	Error in dynamic storage allocation.
ier = –402	At least one subscript pair (**nodlst**(k),**nodlst**(l)) was not specified in structure input. No room for value.

Notes Calls to DSLEV1, DSLEVC, DSLEVE, and DSLEVM can be intermixed. Using DSLEVE is more efficient in finite element contexts, where the matrix is created as the sum of elemental matrices, if the user has stored the elemental matrices rather than the assembled matrix.

Example Rows and columns 345, 346, 347, and 989 form a small dense submatrix of A. Add the value 1.0 to the values in the positions consisting of all pairs of numbers from this set to the list of nonzeros in the matrix.

```
      INTEGER*4 NNODE,NODLST(10),IER
      REAL*8    ELMMTX(10,10),GLOBAL(150)
      NNODE = 4
      NODLST(1) = 345
      NODLST(2) = 346
      NODLST(3) = 347
      NODLST(4) = 989
      DO 200 K = 1, NNODE
        DO 100 L = K, NNODE
          ELMMTX(K,L) = 1.0D0
  100   CONTINUE
  200 CONTINUE
      CALL DSLEVE (NNODE,NODLST,ELMMTX,10,GLOBAL,IER)
      IF ( IER .NE. 0 ) THEN
        handle error condition
      ENDIF
```

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:

```
INTEGER*4 neqns, nnzero, colstr(neqns+1), rowind(nnzero), ier
REAL*8    values(nnzero), global(150)
CALL DSLEVM (colstr, rowind, values, global, ier)
```

Input

colstr — **colstr**(j) gives the index in **rowind** of the first nonzero in the lower triangular part of column j of the matrix. All of the nonzeros for column j are found, in ascending order, in **rowind(colstr(**j**))**, **rowind(colstr(**j**)+1)**, ..., **rowind(colstr(**j+1)–1). **colstr(neqns+1)** must be set to one greater than the total number of nonzeros 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:

ier = 0	Normal return.
ier = –400	Incorrect processing path; DSLEOR not called.
ier = –401	Error in dynamic storage allocation.
ier = –402	At least one subscript pair was not specified in structure input. No room for value.

Notes

This is the most efficient mechanism for specifying the nonzero values. Normally, DSLEVM is used in conjunction with DSLEIM. Additional entries or modifications can be entered with DSEVI1, DSEVIC, or DSEVIE.

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 would have avoided the calls to DSLEV1. Including the diagonal entries in COLSTR and ROWIND is also acceptable to DSLEIM.

Numeric Factorization and Condition Number Estimation — DSLECO

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; $0 \le$ **pvttol** ≤ 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
```

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 inrtia(3), ier
REAL*8 pvttol, global(150)
CALL DSLEFA (pvttol, inrtia, global, ier)

Input

pvttol Pivoting tolerance; 0 ≤ **pvttol** ≤ 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:

ier = 0	Normal return.
ier = –500	Incorrect processing path; value input not performed.
ier = –501	Error in dynamic storage allocation during factorization.
ier = –502	Error in dynamic storage allocation during factorization.
ier = –503	Exactly zero pivot encountered during factorization; matrix is singular.

Example Factor the matrix input to the package. Use a pivot tolerance of 0.1.

```
INTEGER*4 INRTIA(3),IER
REAL*8    GLOBAL(150)
CALL DSLEFA (0.1D0,COND,INRTIA,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
   handle error condition
ENDIF
```

Solve

DSLESL

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:

INTEGER*4 nrhs, ldrhs, ier
REAL*8 rhs(ldrhs, nrhs), global(150)
CALL DSLESL (nrhs, rhs, ldrhs, global, ier)

Input

nrhs Number of right-hand sides; **nrhs** > 0.

ldrhs The leading dimension of array **rhs** as declared in the calling program unit, with **ldrhs** $\geq$ **neqns**.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

rhs On input, **rhs** contains the **nrhs** right-hand sides. On output, **rhs** has been overwritten with computed solutions.

Output

ier Status response:

ier = 0	Normal return.
ier = –600	Incorrect processing path; numeric factorization has not been performed.
ier = –602	**nrhs** $\leq$ 0.
ier = –603	**ldrhs** $<$ **neqns**.

Example

Given the following right-hand side vector, compute the solution given the numerical factorization computed by DSLECO or DSLEFA.

$$b = \begin{matrix} 1.0 \\ 0.0 \\ 1.0 \\ 1.0 \\ 0.0 \\ 0.0 \end{matrix}$$

```
INTEGER*4 IER
REAL*8    RHS(6),GLOBAL(150)

RHS(1) = 1.0
RHS(2) = 0.0
RHS(3) = 1.0
RHS(4) = 1.0
RHS(5) = 0.0
RHS(6) = 0.0

CALL DSLESL (1,RHS,6,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

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 = –701	Error in dynamic storage deallocation.

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
```

Output Control — DSLEOC

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:**

```
INTEGER*4 msglvl, output
REAL*8    global(150)
CALL DSLEOC (msglvl, output, global)
```

Input

msglvl Message level for printable output:

msglvl ≤ 0	Suppress all output.
msglvl = 1	Error messages and summary statistics.
msglvl = 2	More complete statistics.
msglvl = 3	First stage of debugging output.
msglvl = 4	Complete debugging output.

output FORTRAN logical unit number to which all output 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 output unit the same at 6.

```
REAL*8 GLOBAL(150)
CALL DSLEOC (2,6,GLOBAL)
```

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 range 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)
```

Restore Problem State from a Savefile — DSLERS

Purpose

This subprogram restores the working problem from the state stored on the user-specified I/O file by subprogram DSLESV.

Usage

VECLIB:

INTEGER*4 svfile, ier
REAL*8 global(150)
CALL DSLERS (svfile, global, ier)

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 = –901	Error in dynamic storage allocation.
ier = –902	I/O error detected by FORTRAN.

Example

Restart the solution process form the save file on FORTRAN logical unit 42 created by subroutine DSLESV.

```
REAL*8 GLOBAL(150)
CALL DSLERS (42,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

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, 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 after DSLESL or DSLEFS).

```
INTEGER*4 INUSE,MXUSED,OPCNTS(2)
REAL*8    GLOBAL(150),TIME(6)
CALL DSLESR (GLOBAL,INUSE,MXUSED,TIME,OPCNTS)
```

Save Problem State to a Savefile — DSLESV

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:**

INTEGER*4 svfile, ier
REAL*8 global(150)
CALL DSLESV (svfile, global, ier)

Input

svfile FORTRAN logical unit number of the file onto which the state is to be saved. The file will be rewound before and after use.

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0 Normal return.
ier = –902 I/O error detected by FORTRAN.

Example Save the current state on FORTRAN logical unit 42.

```
REAL*8 GLOBAL(150)
CALL DSLESV (42,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

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 CONVEX 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.

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; $B=K_s$, 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:

$$\begin{matrix} 11 & 0 & 13 & 14 & 0 & 0 \\ 0 & 22 & 23 & 0 & 25 & 0 \\ 31 & 32 & 33 & 0 & 35 & 0 \\ 41 & 0 & 0 & 44 & 45 & 0 \\ 0 & 52 & 53 & 54 & 55 & 0 \\ 0 & 0 & 0 & 0 & 0 & 66 \end{matrix}$$

This matrix could be represented in the format described above by three arrays, IROW, JCOL, and MXVALU, as shown in Figure 7-1.

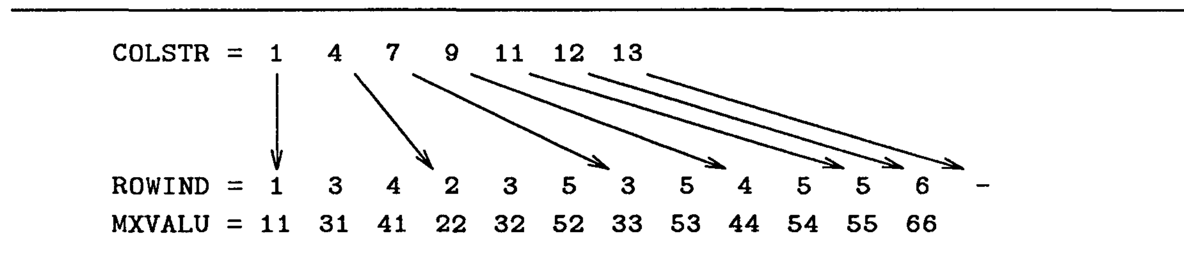
Figure 7–1: Row and Column Index Sparse Matrix Representation

```
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 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 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 $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$. The results from subroutines in this section normally form a contiguous set of eigenvalues, say, λ_i, λ_{i+1}, $\cdots$, λ_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 what is wanted are the algebraically least or greatest eigenvalues, the problem must be transformed by the user 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 is 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 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 $-1000 \le$ **ier** ≤ -100. The hundreds digit indicates the phase of processing in which the error occurred; the other digits specify the error itself. A positive error return indicates that some number of eigenvalues have been computed but a fatal error was encountered which caused early termination of the eigensolution phase.

In addition to the error return flag, the eigensolution phase has an additional warning return, **warnng**, to indicate when the computed results are not exactly what was requested by the user. A warning will occur, for example, when the user requested the lowest 20 eigenvalues in a finite interval which contains only 11 eigenvalues.

Output Controls

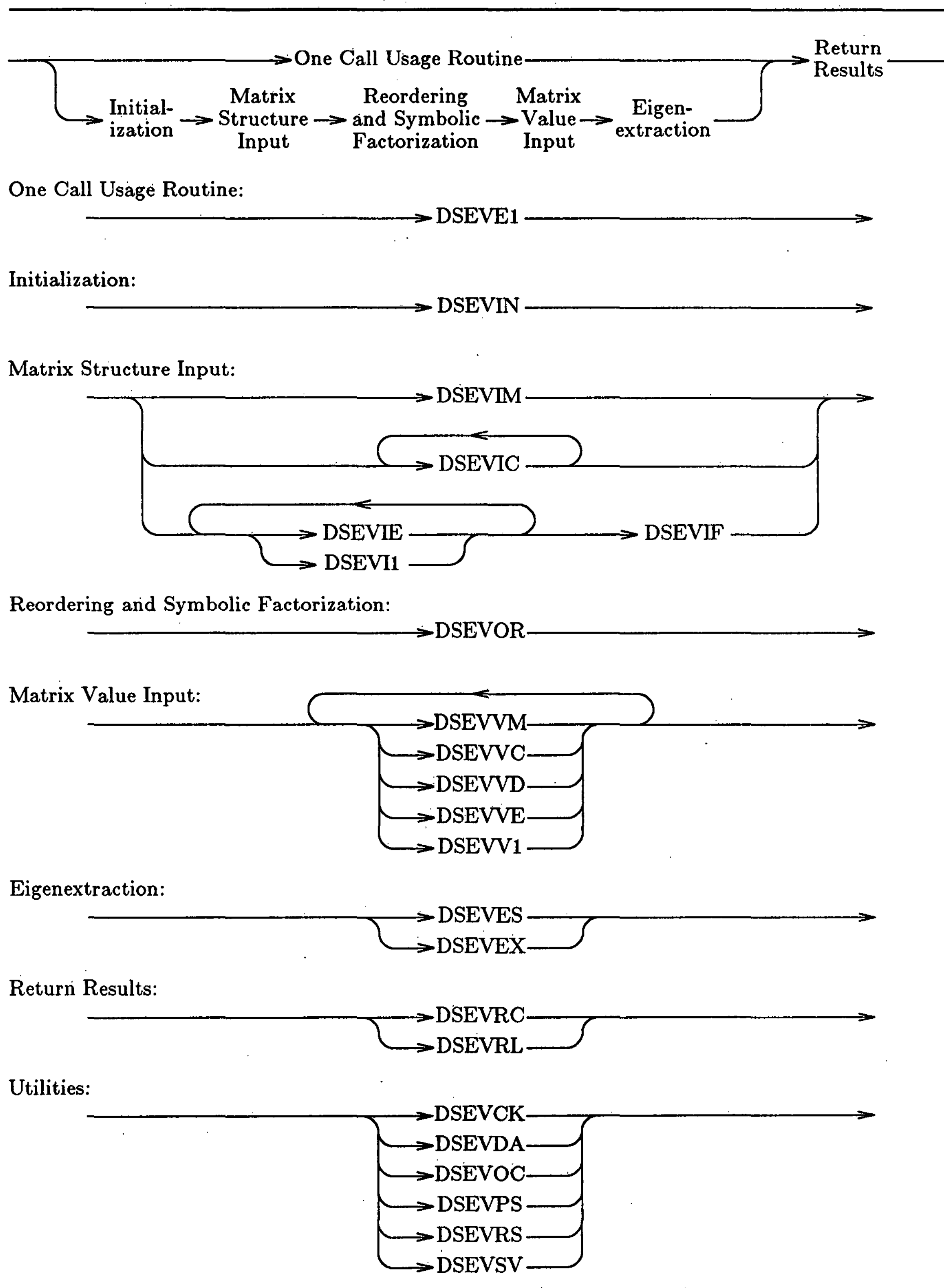
This package differs from most library subroutines in providing optional printed or printable output. The amount of output is controlled by an integer variable, **msglvl**, specifying the message level. Setting **msglvl** ≤ 0 suppresses all printed messages, including error messages, and thus should 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, the user 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.

Figure 7–3: Paths of Control



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.

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 following 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*4    DISCRP,DUMMY,VALUE
      REAL*4    MXVALU(NNZERO),GLOBAL(150),EVALUE(10),EVECTR(NEQNS,10)

C     ------------------------------
C     ... INPUT THE MATRIX STRUCTURE
C     ------------------------------

      CALL DSEVIN (NEQNS,'I',1,6,GLOBAL,IER )
      IF ( IER .NE. 0 ) GO TO 8000

      DO 100 K = 1, NNZERO
          I = IROW(K)
          J = JCOL(K)
          CALL DSEVI1 ('A',I,J,GLOBAL,IER)
          IF ( IER .NE. 0 ) GO TO 8000
  100 CONTINUE

      CALL DSEVIF (GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000
```

```
C     ----------------------
C     ... REORDER THE MATRIX
C     ----------------------

      CALL DSEVOR (GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000

C     -----------------------
C     ... INPUT MATRIX VALUES
C     -----------------------

      DO 200 K = 1, NNZERO
          I     = IROW(K)
          J     = JCOL(K)
          VALUE = MXVALU(K)
          CALL DSEVV1 ('A',I,J,VALUE,GLOBAL,IER)
          IF ( IER .NE. 0 ) GO TO 8000
  200 CONTINUE

C     --------------------------------------------
C     ... COMPUTE THE EIGENVALUES AND EIGENVECTORS
C     --------------------------------------------

      CALL DSEVES (10,'L','O',.FALSE.,DUMMY,.FALSE.,DUMMY,
     1              DUMMY,NFOUND,NDISCD,GLOBAL,IER,WARNNG)
      IF ( IER .NE. 0 ) GO TO 8000

C     ------------------
C     ... CHECK ACCURACY
C     ------------------

      CALL DSEVCK (.TRUE.,ORTWRN,DISCRP,GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000
      IF ( ORTWRN ) PRINT *,'EIGENVALUE/EIGENVECTOR DISCREPANCY =',DISCRP

C     -----------------------------------------
C     ... RETRIEVE EIGENVALUES AND EIGENVECTORS
C     -----------------------------------------

      CALL DSEVRC (20,EVALUE,1,NFOUND,EVECTR,NEQNS,GLOBAL,IER)
      IF ( IER .NE. 0 ) GO TO 8000

C     ----------------------------------------
C     ... USE THE EIGENVALUES AND EIGENVECTORS
C     ----------------------------------------

      .
      .
      .

C     --------------
C     ... ERROR TRAP
C     --------------

 8000  ..........
```

Supplemental Reading

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.

Subprogram Descriptions

One Call Usage
DSEVE1 7-12

Initialize Sparse Eigenvalues/Eigenvectors
DSEVIN 7-18

Matrix Structure Input by Single Entry
DSEVI1 7-20

Matrix Structure Input by Column
DSEVIC 7-21

Matrix Structure Input by Finite Element
DSEVIE 7-23

Matrix Structure Input by Matrix
DSEVIM 7-25

End of Matrix Structure Input
DSEVIF 7-27

Reordering and Symbolic Factorization
DSEVOR 7-28

Matrix Value Input by Single Entry
DSEVV1 7-29

Matrix Value Input by Column
DSEVVC 7-31

Matrix Value Input to Main Diagonal
DSEVVD 7-33

Matrix Value Input by Finite Element
DSEVVE 7-34

Matrix Value Input by Matrix
DSEVVM 7-36

Eigenextraction
DSEVES 7-38

Eigenextraction
DSEVEX 7-41

Return Eigenvalue/Eigenvector Results
DSEVRC 7-45

Return Eigenvalue Results
DSEVRL 7-47

Check Accuracy of Results
DSEVCK 7-49

Deallocate Working Storage
DSEVDA 7-50

Output Control
DSEVOC 7-51

Print Statistics
DSEVPS 7-52

Restore Problem State from a Savefile
DSEVRS 7-53

Save Problem State to a Savefile
DSEVSV 7-54

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     lfinit, rfinit
REAL*8        avalue(annzer), bvalue(bnnzer),
              lftend, rhtend, center, global(150)
CALL DSEVE1 (norder, msglvl, output, acolst, arowin, avalue,
             bmxtyp, bcolst, browin, bvalue, neigvl, which,
             pbtype, lfinit, lftend, rfinit, rhtend, center,
             nfound, ndiscd, global, ier, warnng)
```

Input

norder Order of matrices (number of degrees of freedom), **norder** > 0.

msglvl Message level for printable output:

msglvl $\leq$ 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.

pbtype 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 $Kx=\lambda K_s x$, with K positive semi-definite and K_s 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 pbtype.
ier = ±703	Illegal specification for **lftend** and **rhtend**.
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 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 encountered during Lanczos iteration.
ier = ±724	QL iteration did not converge.
ier = ±725	Number of eigenvalues computed exceeded storage limitations.
ier = ±726	Internal error involving access of Lanczos recurrence vectors.
ier = ±727	Singular Value Decomposition did not converge.
ier = ±728	Gram-Schmidt reorthogonalization did not converge.
ier = ±729	Three numeric factorizations in a row failed. Probable cause is that this is not a symmetric generalized eigenproblem.
ier = ±730	Number of trust regions formed exceeded storage limitations.
ier = ±740	Error in deallocation of storage for first set of Lanczos recurrence vectors.
ier = ±741	Error in deallocation of storage for second set of Lanczos recurrence vectors.

warnng Warning status. **warnng** includes several different warnings encoded in a three digit decimal integer:

low order digit:

warnng = xx0	Normal return.
warnng = xx1	Fewer modes computed than requested (interval does not include requested number).
warnng = xx2	Fewer modes computed than requested because eigenvalues are wildly different in size.
warnng = xx3	Both of the above conditions occurred.

ten's digit:

warnng = x0x	Normal return.
warnng = x1x	At some point in the process, an inconsistent inertia count was found.

hundred's digit:

warnng = 0xx	Normal return.
warnng = 1xx	Interval had to be expanded because one or both endpoints were very close to eigenvalues.

Notes

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **pbtype** argument as 'vibration' or 'buckling'.

Example

In a small (order 6) sparse generalized eigenvalue problem, the matrices A and B are as follows:

$$A = \begin{matrix} 4.0 & 0.0 & 1.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 4.0 & 1.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 1.0 & 4.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 4.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 1.0 & 1.0 & 4.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 1.0 \end{matrix}$$

$$B = \begin{matrix} 1.0 & 0.0 & 1.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 1.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 1.0 & 1.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 1.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 1.0 \end{matrix}$$

The eigenproblem is to compute the two eigenvalues nearest zero.

```
INTEGER*4 MSGLVL,OUTPUT,ACOLST(7),AROWIN(13),IER
REAL*8    GLOBAL(150),AVALUE(13),BVALUE(13)

MSGLVL = 1
OUTPUT = 6

ACOLST(1) = 1
ACOLST(2) = 4
ACOLST(3) = 7
ACOLST(4) = 9
ACOLST(5) = 12
ACOLST(6) = 13
ACOLST(7) = 14

AROWIN(1) = 1
AROWIN(2) = 3
AROWIN(3) = 4
AROWIN(4) = 2
AROWIN(5) = 3
AROWIN(6) = 5
AROWIN(7) = 3
AROWIN(8) = 5
AROWIN(9) = 4
AROWIN(10) = 5
AROWIN(11) = 6
AROWIN(12) = 5
AROWIN(13) = 6

AVALUE(1) = 4.0
AVALUE(2) = 1.0
AVALUE(3) = 1.0
AVALUE(4) = 4.0
AVALUE(5) = 1.0
AVALUE(6) = 1.0
AVALUE(7) = 4.0
AVALUE(8) = 1.0
AVALUE(9) = 4.0
AVALUE(10) = 1.0
AVALUE(11) = 1.0
```

```
AVALUE(12) = 4.0
AVALUE(13) = 1.0

BVALUE(1) = 1.0
BVALUE(2) = 1.0
BVALUE(3) = 1.0
BVALUE(4) = 1.0
BVALUE(5) = 1.0
BVALUE(6) = 1.0
BVALUE(7) = 1.0
BVALUE(8) = 1.0
BVALUE(9) = 1.0
BVALUE(10) = 1.0
BVALUE(11) = 1.0
BVALUE(12) = 1.0
BVALUE(13) = 1.0

CALL DSEVE1 (6,MSGLVL,OUTPUT,ACOLST,AROWIN,AVALUE,
         'A',IDUMMY,IDUMMY,BVALUE,2,'LOWEST',
         'VIBRATION',.FALSE.,-1.0D30,.FALSE.,1.0D30,0.0D0,
         NFOUND,NDISCD,GLOBAL,IER,WARNNG)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

DSEVIN — Initialize Sparse Eigenvalues/Eigenvectors

Purpose This subprogram provides information to start the sparse eigenvalue package.

Usage VECLIB:

```
CHARACTER*1 bmxtyp
INTEGER*4   norder, msglvl, output, ier
REAL*8      global(150)
CALL DSEVIN (norder, bmxtyp, msglvl, output, global, ier)
```

Input

norder Order of matrices (number of degrees of freedom), **norder** > 0.

bmxtyp A character string specifying the type of sparsity found in the right-hand-side B matrix in a generalized eigenvalue problem $Ax = \lambda Bx$.

'I' or 'i' or '1'	B is an identity matrix. Use this option for a standard (ordinary) eigenproblem $Ax = \lambda x$, where there is no second matrix.
'A' or 'a' or 'K' or 'k'	The sparsity structure of B is the same as that of A or K.
'D' or 'd'	B is a diagonal matrix, but not necessarily the identity.
Any other letter	B is a general sparse matrix whose structure is specified independently from A.

msglvl Message level for printable output:

msglvl ≤ 0	Suppress all output.
msglvl $= 1$	Error messages, summary statistics and inertia information.
msglvl $= 2$	More complete statistics.
msglvl $= 3$	First stage of debugging output.
msglvl $= 4$	Complete debugging output.

output FORTRAN logical unit number to which all printable output will be written.

Output

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

ier Status response:

ier $= 0$	Normal return.
ier $= -101$	Error in dynamic storage allocation.
ier $= -102$	**norder** not positive.

Notes Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **bmxtyp** argument as 'identity' or 'diagonal'.

Example Prepare to compute some of the eigenvalues and eigenvectors of a 10,000 by 10,000 matrix (ordinary symmetric for eigenproblem). Obtain error messages and standard minimal output on FORTRAN logical unit 6.

```
INTEGER*4 NORDER, IER
REAL*8    GLOBAL(150)
NORDER = 10000
CALL DSEVIN (NORDER,'IDENTITY',1,6,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

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:**

```
CHARACTER*1  matrix
INTEGER*4    irow, jcol, ier
REAL*8       global(150)
CALL DSEVI1 (matrix, irow, jcol, global, ier)
```

Input

matrix Character string denoting the matrix for which the nonzero location is being input:

'A' or 'a' or 'K' or 'k'	Add nonzero to the matrix on the left side.
'B' or 'b' or 'M' or 'm'	Add nonzero to the matrix on the right side.

irow Row index of the nonzero entry, **jcol** ≤ **irow** ≤ **norder**, where **norder** is the matrix order specified in the call to DSEVIN. Input of diagonal entries is optional.

jcol Column index of the nonzero entry, 1 ≤ **jcol** ≤ **norder**.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = –100	Incorrect processing path; DSEVIN not called or matrix input already finished.
ier = –101	Error in dynamic storage allocation.
ier = –104	Illegal value for **jcol**.
ier = –105	Illegal value for **irow**.
ier = –109	Illegal value for matrix designator.
ier = –110	Attempt to add nonzero location to B, which was not specified as having a general sparse structure (B is a diagonal matrix, an identity matrix, or has same structure as A).

Notes Calls to DSEVI1 and DSEVIE can be intermixed. DSEVIC and DSEVIM cannot be used if DSEVI1 or DSEVIE are used.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example Add the entry in row 3035, column 1024 to the list of nonzeros in the matrix A.

```
INTEGER*4 I,J,IER
REAL*8    GLOBAL(150)
I = 3035
J = 1024
CALL DSEVI1 ('A',I,J,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

Matrix Structure Input by Column — DSEVIC

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 ≤ **jcol** ≤ **norder**, where **norder** is the matrix order as specified in the call to DSEVIN. Columns must be presented in order from 1 to **norder**, except that columns with no entries can be skipped.

nzcol Number of nonzeros in column **jcol** of the matrix, **nzcol** ≥ 0.

jrowin List of row indices for all nonzeros, in ascending order, in the lower triangle part of column **jcol** of the matrix designated by **matrix**, **jcol** ≤**jrowin**(1) < **jrowin**(2) < · · · < **jrowin(nzcol)** ≤ **norder**.

Input of diagonal entries is optional.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = -100	Incorrect processing path; DSEVIN not called or matrix input already finished.
ier = -101	Error in dynamic storage allocation.
ier = -103	Illegal value for **nzcol**.
ier = -104	Illegal value for **jcol**; either out of range or out of order.
ier = -106	Illegal value for at least one entry in **jrowin** or entries out of order.
ier = -109	Illegal value for matrix designator.
ier = -110	Attempt to add nonzero location to B, which was not specified as having a general sparse structure (B is a diagonal matrix, an identity matrix, or has same structure as A).

Notes The entire matrix structure must be input with DSEVIC if it is used. Its use is not compatible with DSEVI1, DSEVIE, or DSEVIM.

If the matrix entries are available by column, using DSEVIC is more efficient than using DSEVI1.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example

Column 4519 has entries in rows 1, 2735, 4519, 4520, 4521, 6000, 6002, and 6004. Add all five entries in the lower triangular part to the list of nonzeros in the matrix A. Columns 1 to 4518 already have been input to the package using DSEVIC.

```
INTEGER*4 J,NZCOL,JROWIN(100),IER
REAL*8    GLOBAL(150)
J = 4519
NZCOL = 5
JROWIN(1) = 4520
JROWIN(2) = 4521
JROWIN(3) = 6000
JROWIN(4) = 6002
JROWIN(5) = 6004
CALL DSEVIC ('A',J,NZCOL,JROWIN,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

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 ≤ **nnode** ≤ **norder**, where **norder** is the matrix order specified in the call to DSEVIN.

nodlst List of nodes in element or clique, 1 ≤ **nodlst**(i) ≤ **norder** for all i, and **nodlst**(i) ≠ **nodlst**(j) for all i and j with $i \neq j$. All pairs (**nodlst**(i),**nodlst**(j)) in the lower triangle are added to the sparsity structure of the matrix designated by **matrix**.

Updated

nodlst The order of the values in **nodlst** may be changed.

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = –100	Incorrect processing path; DSEVIN not called or matrix input already finished.
ier = –101	Error in dynamic storage allocation.
ier = –107	Illegal value for **nnode**.
ier = –108	Illegal value for at least one entry in **nodlst**.
ier = –109	Illegal value for matrix designator.
ier = –110	Attempt to add nonzero location to B, which was not specified as having a general sparse structure (B is a diagonal matrix, an identity matrix, or has same structure as A).

Notes

Calls to DSEVIE can be intermixed with calls to DSEVI1. DSEVIC and DSEVIM cannot be used if DSEVI1 or DSEVIE are used.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example Rows and columns 345, 346, 347, and 989 form a small dense submatrix of A. Add positions consisting of all pairs of numbers from this set to the list of nonzeros in the matrix A.

```
INTEGER*4 NNODE,NODLST(10),IER
REAL*8    GLOBAL(150)
NNODE = 4
NODLST(1) = 345
NODLST(2) = 346
NODLST(3) = 347
NODLST(4) = 989
CALL DSEVIE ('A',NNODE,NODLST,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

Purpose This subprogram specifies the locations of all of the nonzeros in the lower triangle of one of the matrices.

Usage VECLIB:

```
CHARACTER*1 matrix
INTEGER*4   norder, nnzero, colstr(norder+1), rowind(nnzero),
            ier
REAL*8      global(150)
CALL DSEVIM (matrix, colstr, rowind, global, ier)
```

Input

matrix Character string denoting the matrix for which the nonzero location is being input:

'A' or 'a' or 'K' or 'k'	Add nonzero to the matrix on the left side.
'B' or 'b' or 'M' or 'm'	Add nonzero to the matrix on the right side.

colstr **colstr**(j) gives the address in **rowind** of the first nonzero in the lower triangular part of column j of the matrix specified by **matrix**. All of the nonzeros for column j are found, in ascending order, in **rowind(colstr(**j**))**, **rowind(colstr(**j**)+1)**, ..., **rowind(colstr(**$j+1$**)–1)**.

colstr(norder+1) must be set to one greater than the total number of nonzeros, **nnzero** in the lower triangular part of the matrix, where **norder** is the matrix order as specified in the call to DSEVIN.

rowind List of row indices for all nonzeros, in ascending order within each column, in the lower triangle part of the matrix designated by **matrix**. Input of diagonal entries is optional.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = –100	Incorrect processing path; DSEVIN not called or matrix input already finished.
ier = –101	Error in dynamic storage allocation.
ier = –106	Illegal value for at least one entry in **colstr** (not increasing or negative) or invalid entries in **rowind** (entries out of order within a column, or out of the lower triangle)
ier = –109	Illegal value for matrix designator.
ier = –110	Attempt to add nonzero location to B, which was not specified as having a general sparse structure (B is a diagonal matrix, an identity matrix, or has same structure as A).

Notes This is the most efficient mechanism for specifying the nonzero structure, but the entire matrix structure for both matrices must be input with DSEVIM if it is used. Its use is not compatible with DSEVI1, DSEVIE, or DSEVIC.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example In a small (order 6) sparse generalized eigenvalue problem, B is a mass matrix with the following structure:

```
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
```

End of Matrix Structure Input — DSEVIF

Purpose

This subprogram indicates the end of structure input for matrices in the sparse eigenvalue problem. DSEVIF is used only if subprograms DSEVI1 and/or DSEVIE were the mechanism by which the structure was input.

Usage

VECLIB:

```
INTEGER*4 ier
REAL*8    global(150)
CALL DSEVIF (global, ier)
```

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = –100	Incorrect processing path; DSEVIN not called or matrix input already finished.
ier = –101	Error in dynamic storage allocation.

Example

The nonzero structure of a pair of finite element matrices was passed to the sparse eigenvalue package using repeated calls to subroutine DSEVIE. Signal that no more nonzeros will be added to either matrix.

```
INTEGER*4 IER
REAL*8    GLOBAL(150)
CALL DSEVIF (GLOBAL,IER )
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

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, so 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	Internal 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
```

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:

```
CHARACTER*1  matrix
INTEGER*4    irow, jcol, ier
REAL*8       value, global(150)
CALL DSEVV1 (matrix, irow, jcol, value, global, ier)
```

Input

matrix Character string denoting the matrix for which the nonzero location is being input:

'A' or 'a' or 'K' or 'k'	Add to the matrix on the left side.
'B' or 'b' or 'M' or 'm'	Add to the matrix on the right side.

irow Row index of the nonzero entry, **jcol** ≤ **irow** ≤ **norder**, where **norder** is the matrix order as specified in the call to DSEVIN.

jcol Column index of the nonzero entry, 1 ≤ **jcol** ≤ **norder**.

value Numeric value that will be added to any previous values input for this location.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = –400	Incorrect processing path; DSEVOR not called or eigenextraction already begun.
ier = –401	Error in dynamic storage allocation.
ier = –402	Subscript pair (**irow**, **jcol**) is not in lower triangle of matrix.
ier = –403	Illegal value for matrix designator.
ier = –404	Subscript pair (**irow**, **jcol**) was not specified in structure input. No room for value.
ier = –405	The *B* matrix was specified as diagonal. Cannot add off-diagonal nonzero value.
ier = –406	The *B* matrix was specified as an identity matrix. Cannot add nonzero value.

Notes

Calls to DSEVV1, DSEVVC, DSEVVD, DSEVVE, and DSEVVM can be intermixed.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example Store the value 4.523×10^{-5} as the nonzero entry in row 3035, column 1024 of the matrix A.

```
INTEGER*4 I,J,IER
REAL*8    VALUE,GLOBAL(150)
I = 3035
J = 1024
VALUE = 4.523D-5
CALL DSEVV1 ('A',I,J,VALUE,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

Matrix Value Input by Column

DSEVVC

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:

```
CHARACTER*1  matrix
INTEGER*4    jcol, nzcol, jrowin(nzcol), ier
REAL*8       values(nzcol), global(150)
CALL DSEVVC (matrix, jcol, nzcol, jrowin, values, global, ier)
```

Input

matrix Character string denoting the matrix for which the nonzero location is being input:

'A' or 'a' or 'K' or 'k'	Add to the matrix on the left side.
'B' or 'b' or 'M' or 'm'	Add to the matrix on the right side.

jcol Column index, 1 ≤ **jcol** ≤ **norder**, where **norder** is the matrix order as specified in the call to DSEVIN.

nzcol Number of values in the list to be added to column **jcol** of the matrix, **nzcol** > 0.

jrowin List of row indices, in ascending order, for values to be added to the lower triangle part of column **jcol** of the matrix designated by **matrix**, **jcol** ≤ **jrowin(1)** < **jrowin(2)** < · · · < **jrowin(nzcol)** ≤ **norder**. It is not necessary that all entries in column **jcol** be included in **jrowin**.

values List of values corresponding to positions specified by **jcol** and **jrowin**.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = –400	Incorrect processing path; DSEVOR not called or eigenextraction already begun.
ier = –401	Error in dynamic storage allocation.
ier = –402	Illegal value for either **nzcol** or **jcol**.
ier = –403	Illegal value for matrix designator.
ier = –404	At least one subscript pair, (**jrowin**(j),**jcol**) was not specified in structure input. No room for value.
ier = –405	The B matrix was specified as diagonal. Cannot add off-diagonal nonzero value.
ier = –406	The B matrix was specified as an identity matrix. Cannot add nonzero value.

Notes

Calls to DSEVV1, DSEVVC, DSEVVD, DSEVVE and DSEVVM can be intermixed. If the matrix entries are available by column, using DSEVVC is more efficient than using DSEVV1.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the CALL statement may be improved by coding the **matrix** argument as 'addleft'.

Example Column 4519 has entries in rows 1, 2735, 4519, 4520, 4521, 6000, 6002, and 6004. Add values 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
```

Matrix Value Input to Main Diagonal — DSEVVD

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 B matrix was specified as an identity matrix. Cannot add nonzero value.

Notes Calls to DSEVV1, DSEVVC, DSEVVD, DSEVVE and DSEVVM can be intermixed. If the matrix entries of the main diagonal are available as a vector, using DSEVVD is more efficient than using DSEVV1.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example Create a main diagonal equal to 4.0 for B.

```
INTEGER*4 IER
REAL*8    VALUES(100),GLOBAL(150)

VALUES(1) = 4.0D0
VALUES(2) = 4.0D0
VALUES(3) = 4.0D0
VALUES(4) = 4.0D0
VALUES(5) = 4.0D0
VALUES(6) = 4.0D0

CALL DSEVVD ('B',VALUES,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

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:**

```
CHARACTER*1  matrix
INTEGER*4    nnode, ldelmx, nodlst(nnode), ier
REAL*8       elmmtx(ldelmx, nnode), global(150)
CALL DSEVVE (matrix, nnode, nodlst, elmmtx, ldelmx, global, ier)
```

Input

matrix Character string denoting the matrix for which the nonzero location is being input:

'A' or 'a' or 'K' or 'k'	Add to the matrix on the left side.
'B' or 'b' or 'M' or 'm'	Add to the matrix on the right side.

nnode Number of nodes in the finite element or clique, 1 ≤ **nnode** ≤ **norder**, where **norder** is the matrix order as specified in the call to DSEVIN.

nodlst List of nodes in element or clique, 1 ≤ **nodlst**(i) ≤ **norder** for all i, and **nodlst**(i) ≠ **nodlst**(j) for all i and j with $i \neq j$. Values for all pairs (**nodlst**(i),**nodlst**(j)) in the lower triangle are added to the values of the matrix designated by **matrix**.

elmmtx A two-dimensional array containing values to be added to the matrix. Only the lower triangle (including the diagonal) of **elmmtx** is referenced. The value in **elmmtx**(k,l) is added to the value in position (**nodlst**(k),**nodlst**(l)) in the sparse matrix.

ldelmx The leading dimension of array **elmmtx** as declared in the calling program unit, with **ldelmx** ≥ **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** (values out of range).
ier = –403	Illegal value for matrix designator.
ier = –404	At least one subscript pair, (**nodlst**(k),**nodlst**(l)) was not specified in structure input. No room for value.
ier = –405	The B matrix was specified as diagonal. Cannot add off-diagonal nonzero value.
ier = –406	The B matrix was specified as an identity matrix. Cannot add nonzero value.

Notes Calls to DSEVV1, DSEVVC, DSEVVD, DSEVVE, and DSEVVM can be intermixed. If the matrix entries are available as a sum of elemental matrices, using DSEVVE is more efficient than using DSEVV1.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example Rows and columns 345, 346, 347, and 989 form a small dense submatrix of A. Add the value 1.0 to the values in the positions consisting of all pairs of numbers from this set to the list of nonzeros in matrix A.

```
      INTEGER*4 NNODE,NODLST(10),IER
      REAL*8    ELMMTX(10,10),GLOBAL(150)

      NNODE = 4

      NODLST(1) = 345
      NODLST(2) = 346
      NODLST(3) = 347
      NODLST(4) = 989

      DO 200 K = 1, NNODE
        DO 100 L = K, NNODE
          ELMMTX(K,L) = 1.0D0
100     CONTINUE
200   CONTINUE
      CALL DSEVVE ('A',NNODE,NODLST,ELMMTX,10,GLOBAL,IER)
      IF ( IER .NE. 0 ) THEN
        handle error condition
      ENDIF
```

Purpose This subprogram adds values to all of the entries in the lower triangle of one of the sparse matrices.

Usage **VECLIB:**

```
CHARACTER*1 matrix
INTEGER*4   norder, nnzero, colstr(norder+1), rowind(nnzero), ier
REAL*8      values(nnzero), global(150)
CALL DSEVVM (matrix, colstr, rowind, values, global, ier)
```

Input

matrix Character string denoting the matrix for which the nonzero location is being input:

'A' or 'a' or 'K' or 'k'	Add to the matrix on the left side.
'B' or 'b' or 'M' or 'm'	Add to the matrix on the right side.

colstr **colstr**(j) gives the address in **rowind** of the first nonzero in the lower triangular part of column j of the matrix specified by **matrix**. All of the nonzeros for column j are found, in ascending order, in **rowind(colstr(**j**))**, **rowind(colstr(**j**)+1)**, ..., **rowind(colstr(**$j+1$**)–1)**.

colstr(norder+1) must be set to one greater than the total number of nonzeros, **nnzero**, in the lower triangular part of the matrix, where **norder** is the matrix order as specified in the call to DSEVIN.

rowind List of row indices for all nonzeros, in ascending order within each column, in the lower triangle part of the matrix designated by **matrix**. Diagonal entries may be present, but are not required.

values List of values corresponding in position to the indices in **rowind**. These values will be added to any values already present in the matrix specified by **matrix**.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = –400	Incorrect processing path; DSEVOR not called or eigenextraction already begun.
ier = –401	Error in dynamic storage allocation.
ier = –402	Illegal value for either **nnode** or in **nodlst**.
ier = –403	Illegal value for matrix designator.
ier = –404	At least one subscript pair was not specified in structure input. No room for value.
ier = –405	The B matrix was specified as diagonal. Cannot add off-diagonal nonzero value.
ier = –406	The B matrix was specified as an identity matrix. Cannot add nonzero value.

Notes This is the most efficient mechanism for specifying nonzero values. Normally, DSEVVM is used in conjunction with DSEVIM. The matrix structure passed to DSEVVM must be identical to the structure passed to DSEVIM.

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **matrix** argument as 'addleft'.

Example In a small (order 6) sparse generalized eigenvalue problem, B is the following mass matrix:

4.0	0.0	1.0	1.0	0.0	0.0
0.0	4.0	1.0	0.0	1.0	0.0
1.0	1.0	4.0	0.0	1.0	0.0
1.0	0.0	0.0	4.0	1.0	0.0
0.0	1.0	1.0	1.0	4.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0

Use DSEVVM and DSEVVD to input this as B.

```
INTEGER*4 COLSTR(7),ROWIND(6),IER
REAL*8    VALUES(6),DIAGNL(6),GLOBAL(150)

COLSTR(1) = 1
COLSTR(2) = 3
COLSTR(3) = 5
COLSTR(4) = 6
COLSTR(5) = 7
COLSTR(6) = 7
COLSTR(7) = 7

ROWIND(1) = 3
ROWIND(2) = 4
ROWIND(3) = 3
ROWIND(4) = 5
ROWIND(5) = 6
ROWIND(6) = 6

VALUES(1) = 1.0
VALUES(2) = 1.0
VALUES(3) = 1.0
VALUES(4) = 1.0
VALUES(5) = 1.0
VALUES(6) = 1.0

CALL DSEVVM ('B',COLSTR,ROWIND,VALUES,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF

DIAGNL(1) = 4.0
DIAGNL(2) = 4.0
DIAGNL(3) = 4.0
DIAGNL(4) = 4.0
DIAGNL(5) = 4.0
DIAGNL(6) = 0.0

CALL DSEVVD ('B',DIAGNL,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

Note: it would not have been necessary to use DSEVVD if the diagonal values had been held in the same value array as the rest of the sparse matrix,

Purpose Compute 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, pbtype
INTEGER*4   neigvl, nfound, ndiscd, ier, warnng
LOGICAL*4   lfinit, rfinit
REAL*8      lftend, rhtend, center, global(150)
CALL DSEVES (neigvl, which, pbtype, lfinit, lftend, rfinit, rhtend,
             center, nfound, ndiscd, global, ier, warnng)
```

Input

neigvl Number of eigenvalues to be found, **neigvl** > 0.

which Character string indicating which eigenvalues are to be computed.

'L' or 'l'	The lowest (smallest magnitude) eigenvalues.
'H' or 'h'	The highest (greatest magnitude) eigenvalues.
'C' or 'c' or 'N' or 'n'	The eigenvalues nearest **center**.
'A' or 'a'	All eigenvalues in the specified interval.

pbtype Character string indicating type of problem.

'V' or 'v'	Generalized symmetric (vibration) problem $Kx=\lambda Mx$, with M positive semi-definite.
'B' or 'b'	Generalized symmetric (buckling) problem $Kx=\lambda K_s x$, with K positive semi-definite and K_s possibly indefinite.
'O' or 'o'	Ordinary symmetric eigenproblem $Kx=\lambda x$.

lfinit If .TRUE., the value of the argument **lftend** is to be used as a restriction on the location of computed eigenvalues. If .FALSE., no lower bound is placed on the value of computed eigenvalues; equivalently, the left endpoint is negative infinity.

lftend Left endpoint of an interval in which the desired eigenvalues lie, not used if **lfinit** is .FALSE.

rfinit If .TRUE., the value of the argument **rhtend** is to be used as a restriction on the location of computed eigenvalues. If .FALSE., no upper bound is placed on the value of computed eigenvalues; equivalently, the right endpoint is positive infinity.

rhtend Right endpoint of an interval in which the desired eigenvalues lie, not used if **rfinit** is .FALSE.

center A center or critical value for use when eigenvalues nearest some particular value are desired. Referenced only when **which** specifies 'centered' or 'nearest'.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

nfound The number of eigenvalues computed and confirmed to meet the problem description constraints by inertia (Sturm sequence) computations.

ndiscd The number of other eigenvalues computed during the course of the computation, but discarded because they were outside specifications for eigenvalues or because they were not confirmed by inertia computations.

ier Status response:

ier = 0	Normal return.
ier < 0	Fatal error, no useful results returned.
ier > 0	Fatal error, but some eigenvalues and vectors computed.
ier = –700	DSEVES called without first successfully calling earlier stages.
ier = ±701	Error in storage allocation.
ier = ±702	Illegal specification for **pbtype**.
ier = ±703	Illegal specification for **lftend** and **rhtend**.
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 encountered during Lanczos iteration.
ier = ±724	QL iteration did not converge.
ier = ±725	Number of eigenvalues computed exceeded storage limitations.
ier = ±726	Internal error involving access of Lanczos recurrence vectors.
ier = ±727	Singular Value Decomposition did not converge.
ier = ±728	Gram-Schmidt reorthogonalization did not converge.
ier = ±729	Three numeric factorizations in a row failed. Probable cause is that this is not a symmetric generalized eigenproblem.
ier = ±730	Number of trust regions formed exceeded storage limitations.
ier = ±731	Error during restoration of a numeric factorization.
ier = ±740	Error in deallocation of storage for first set of Lanczos recurrence vectors.
ier = ±741	Error in deallocation of storage for second set of Lanczos recurrence vectors.

warnng Warning status. **warnng** includes several different warnings encoded in a three-digit decimal integer:

low order digit:

warnng = xx0	Normal return.
warnng = xx1	Fewer modes computed than requested (interval does not include requested number).
warnng = xx2	Fewer modes computed than requested because eigenvalues are different in size.
warnng = xx3	Both of the above conditions occurred.

ten's digit:

warnng = x0x	Normal return.
warnng = x1x	At some point in the process, an inconsistent inertia count was found.

hundred's digit:

warnng = 0xx	Normal return.
warnng = 1xx	Interval had to be expanded because one or both endpoints were very close to eigenvalues.

Notes

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **pbtype** argument as 'vibration' or 'buckling'.

Example 1

Compute the lowest 10 eigenvalues of a structural engineering vibration analysis, where matrix A is the stiffness matrix K, matrix B is the mass matrix M, and matrices K and M are positive definite or semi-definite.

```
INTEGER*4 NFOUND,NDISCD,IER,WARNNG
CALL DSEVES (10,'LOWEST','VIBRATION',.FALSE.,-1.0D30,
        .FALSE.,.1.0D30,1.0D30,NFOUND,NDISCD,GLOBAL,IER,
        WARNNG)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
IF ( IER .GE. 0 ) THEN
  retrieve eigenvalues and/or eigenvectors
ENDIF
```

Example 2

Compute the lowest negative eigenvalue and corresponding eigenvector for a structural engineering buckling analysis, where the matrix A is the stiffness matrix K and the matrix B is the geometric stiffness matrix K_s. 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
```

Purpose

Compute 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, nusrvc, nfound, ndiscd, ier,
             warnng
LOGICAL*4    lfinit, rfinit
REAL*8       lftend, rhtend, center, tolact, shfscl, global(150),
             usrsvc(norder, nusrvc)
CALL DSEVEX (neigvl, which, pbtype, lfinit, lftend, rfinit,
             rhtend, center, mxbksz, tolact, shfscl, nusrvc,
             usrsvc, nfound, ndiscd, global, ier, warnng)
```

Input

neigvl Number of eigenvalues to be found.

which Character string indicating which eigenvalues are to be computed.

'L' or 'l'	The lowest (smallest magnitude) eigenvalues.
'H' or 'h'	The highest (greatest magnitude) eigenvalues.
'C' or 'c' or 'N' or 'n'	The eigenvalues nearest **center**.
'A' or 'a'	All eigenvalues in the specified interval.

pbtype Character string indicating type of problem.

'V' or 'v'	Generalized symmetric (vibration) problem $Kx = \lambda Mx$, with M positive semi-definite.
'B' or 'b'	Generalized symmetric (buckling) problem $Kx = \lambda K_s x$, with K positive semi-definite and K_s possibly indefinite.
'O' or 'o'	Ordinary symmetric eigenproblem $Kx = \lambda x$.

lfinit If .TRUE., the value of the argument **lftend** is to be used as a restriction on the location of computed eigenvalues. If .FALSE., no lower bound is placed on the value of computed eigenvalues; equivalently, the left endpoint is negative infinity.

lftend Left endpoint of an interval in which the desired eigenvalues lie, not used if **lfinit** is .FALSE.

rfinit If .TRUE., the value of the argument **rhtend** is to be used as a restriction on the location of computed eigenvalues. If .FALSE., no upper bound is placed on the value of computed eigenvalues; equivalently, the right endpoint is positive infinity.

rhtend Right endpoint of an interval in which the desired eigenvalues lie, not used if **rfinit** is .FALSE.

center A center or critical value for use when eigenvalues nearest some particular value are desired. Referenced only when **which** specifies 'centered' or 'nearest'.

mxbksz A limit on the block size used in the block Lanczos recurrence. In general, it is best to choose a block size as large, or slightly larger than, the largest multiplicity of the eigenvalues desired. The following constraints also apply: a block size of 1 is usually less efficient than a block size of 2. Block sizes larger than 10 are usually less efficient than smaller block sizes. If **mxbksz** is zero or negative, a default block size optimized to the computer system will be chosen. In all cases the Lanczos algorithm may find it necessary to reduce the block size to fit the problem into the storage available.

tolact An accuracy tolerance for the eigenvalues, which must be in the range from $\sqrt{\epsilon} \leq$ **tolact** $\leq \epsilon$, where ϵ is the machine precision. If a negative or zero value is supplied, a default value of $\epsilon^{2/3}$ will be used. Special care should be taken if the value of **tolact** is reduced below its default value. In particular, it is important to be sure that the computed eigenvectors remain adequately orthogonal. Subroutine DSEVCK can facilitate such monitoring.

shfscl An estimate of the magnitude of the smallest nonzero eigenvalues. The package will compute a heuristic estimate if the user does not supply an estimate (signaled by **shfscl** $\leq$ 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 **pbtype**.
ier = ±703	Illegal specification for **lftend** and **rhtend**.
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 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 wildly different in size.
warnng = xx3	Both of the above conditions occurred.

ten's digit:

warnng = x0x	Normal return.
warnng = x1x	At some point in the process, an inconsistent inertia count was found.

hundred's digit:

warnng = 0xx	Normal return.
warnng = 1xx	Interval had to be expanded because one or both endpoints were very close to eigenvalues.

Notes

Actual character arguments in a subroutine call may be longer than corresponding dummy arguments. Therefore, readability of the **CALL** statement may be improved by coding the **pbtype** argument as 'vibration' or 'buckling'.

Example 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_s. 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
```

Return Eigenvalue/Eigenvector Results — DSEVRC

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 the discardable set. The following pairs illustrate the meaning of **fstval** and **lstval**:

fstval	lstval	
1	nfound	All final trust-region eigenvalues.
–1	–ndiscd	All other eigenvalues.
1	–ndiscd	All computed eigenvalues.
i	i	i-th final trust-region eigenvalue.
$-i$	$-i$	i-th discardable eigenvalue.
i	j	i-th through j-th final trust-region eigenvalues.
i	$-j$	i-th through 'nfound'-th final trust-region eigenvalues and 1st through j-th discardable eigenvalues.

ldevct The leading dimension of array **evectr** as declared in the calling program unit, with **ldevct** ≥ **norder**, where **norder** is the matrix order as specified in the call to DSEVIN.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

evalue List of selected eigenvalues.

evectr Corresponding array of eigenvectors. Each eigenvector is normalized so that

$$x^T Bx = x^T Mx = 1 \quad \text{(vibration)}$$
$$x^T Ax = x^T Kx = 1 \quad \text{(buckling)}$$
$$x^T x = 1 \quad \text{(ordinary)}$$

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
```

Return Eigenvalue Results **DSEVRL**

Purpose This subprogram retrieves all or selected eigenvalues after eigenextraction.

Usage **VECLIB:**

INTEGER*4 levalu, fstval, lstval, ier
REAL*8 evalue(levalu), global(150)
CALL DSEVRL (levalu, evalue, fstval, lstval, global, ier)

Input

levalu Length of vector for storing eigenvalues.

fstval Ordinal position of first eigenvalue to be returned in **evalue**.

lstval Ordinal position of last eigenvalue to be returned in **evalue**.

The sign of **fstval** and **lstval** determine whether eigenvalues returned are from the confirmed set or the discarded set. Positive indices indicate eigenvalues in the final confirmed trust region, while negative indices the discardable set. The following pairs illustrate the meaning of **fstval** and **lstval**:

fstval	**lstval**	
1	nfound	All final trust-region eigenvalues.
–1	–ndiscd	All other eigenvalues.
1	–ndiscd	All computed eigenvalues.
i	i	i-th final trust-region eigenvalue.
$-i$	$-i$	i-th discardable eigenvalue.
i	j	i-th through j-th final trust-region eigenvalues.
i	$-j$	i-th through 'nfound'-th final trust-region eigenvalues and 1st through j-th discardable eigenvalues.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

evalue List of eigenvalues.

ier Status response:

ier = 0	Normal return.
ier = –800	Incorrect processing path.
ier = –801	**levalu** not large enough.
ier = –803	Input error, indices out of range.

Example 1 Retrieve all of the final trust region eigenvalues.

```
INTEGER*4 NEVALS,NFOUND,IER
REAL*8    EVALUE(NEVALS),GLOBAL(150)
NEVALS = 100
LDEVCT = 1000
CALL DSEVRL (NEVALS,EVALUE,1,NFOUND,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

Example 2 Retrieve all of the computed eigenvalues.

```
INTEGER*4 NEVALS,NDISCD,NFOUND,IER
REAL*8    EVALUE(NEVALS),GLOBAL(150)
IF ( NDISCD .GT. 0 ) THEN
  CALL DSEVRL (NEVALS,EVALUE,1,-NDISCD,GLOBAL,IER)
ELSE
  CALL DSEVRL (NEVALS,EVALUE,1,NFOUND,GLOBAL,IER)
ENDIF
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

Check Accuracy of Results

DSEVCK

Purpose

This subprogram performs an *a posteriori* check on the accuracy of the computed eigenvalues and eigenvectors. The eigenvectors should satisfy the following pairs of equations

$$X^T KX = \mathrm{diag}(\{\lambda_i\}), \qquad X^T MX = I \quad \text{(vibration)}$$

$$X^T K_s X = \mathrm{diag}(\{1/\lambda_i\}), \qquad X^T KX = I \quad \text{(buckling)}$$

$$X^T AX = \mathrm{diag}(\{\lambda_i\}), \qquad X^T X = I \quad \text{(ordinary)}$$

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
```

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
```

Output Control **DSEVOC**

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:**

```
INTEGER*4 msglvl, output
REAL*8    global(150)
CALL DSEVOC (msglvl, output, global)
```

Input

msglvl Message level for printable output:

msglvl $\leq$ 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 the same at 6.

```
REAL*8 GLOBAL(150)
CALL DSEVOC (2,6,GLOBAL)
```

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 range 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)
```

Restore Problem State from a Savefile — DSEVRS

Purpose

This subprogram restores the working problem from the state stored on the user-specified I/O file by subprogram DSEVSV.

Usage

VECLIB:

```
INTEGER*4 svfile, ier
REAL*8    global(150)
CALL DSEVRS (svfile, global, ier)
```

Input

svfile FORTRAN logical unit number of a file containing the saved problem state as created by DSEVSV.

Output

global Global communications array restored from **svfile**.

ier Status response:

ier = 0	Normal return.
ier = –901	Error in storage allocation.
ier = –902	I/O error detected by FORTRAN.

Example

Restart the eigenanalysis from the save file on FORTRAN logical unit 42 created by subroutine DSEVSV.

```
REAL*8 GLOBAL(150)
CALL DSEVRS (42,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

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:**

INTEGER*4 svfile, ier
REAL*8 global(150)
CALL DSEVSV (svfile, global, ier)

Input

svfile FORTRAN logical unit number of the file onto which the state is to be saved. The file will be rewound before and after use.

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = –902	I/O error detected by FORTRAN.

Example Save the current state on FORTRAN logical unit 42.

```
REAL*8 GLOBAL(150)
CALL DSEVSV (42,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

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 CONVEX 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.

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:

$$\begin{matrix} 11 & 0 & 13 & 14 & 0 & 0 \\ 0 & 22 & 23 & 0 & 25 & 0 \\ 31 & 32 & 33 & 0 & 35 & 0 \\ 41 & 0 & 0 & 44 & 45 & 0 \\ 0 & 52 & 53 & 54 & 55 & 0 \\ 0 & 0 & 0 & 0 & 0 & 66 \end{matrix}$$

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 ith 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 ith 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

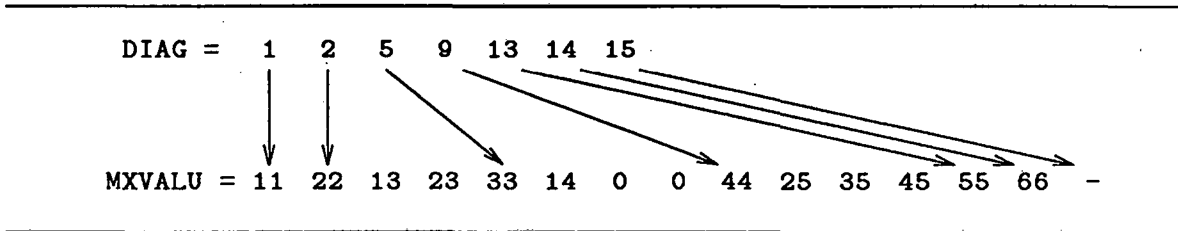
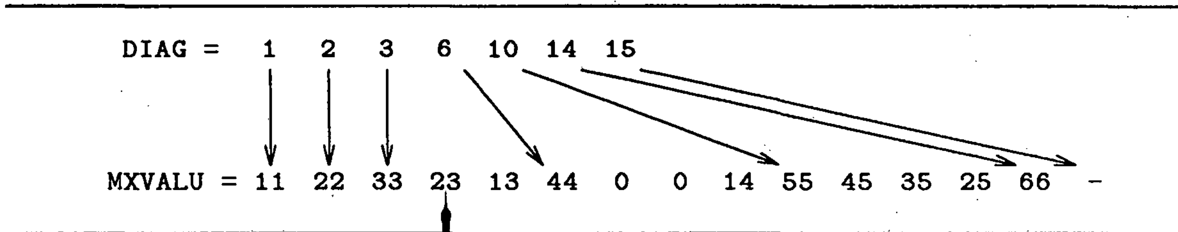
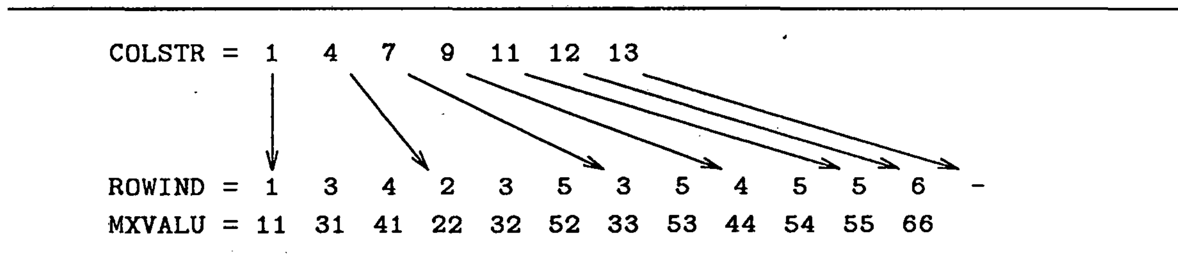


Figure 8–3: Reverse Skyline Matrix Representation



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 an 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 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 $-1000 \leq$ **ier** ≤ -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** $\leq$ 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** $\geq$ 3, various arrays whose length is on the order of the number of equations will be printed. If **msglvl** $\geq$ 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 ($\geq$ 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.

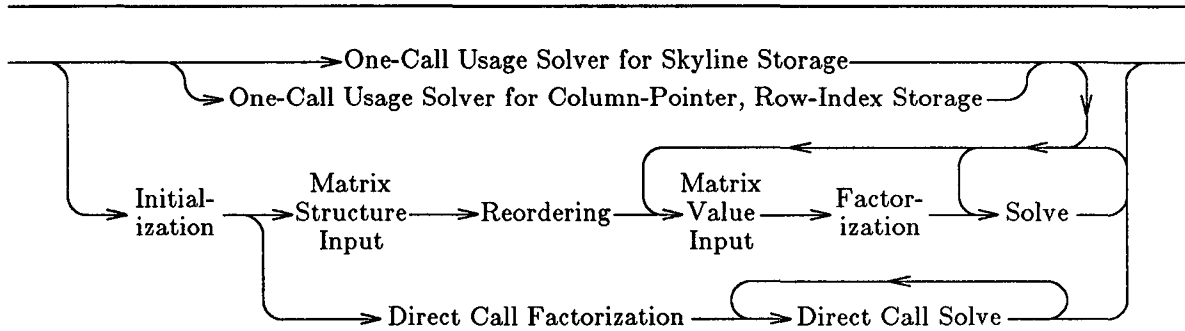
Paths of Control

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:

```
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

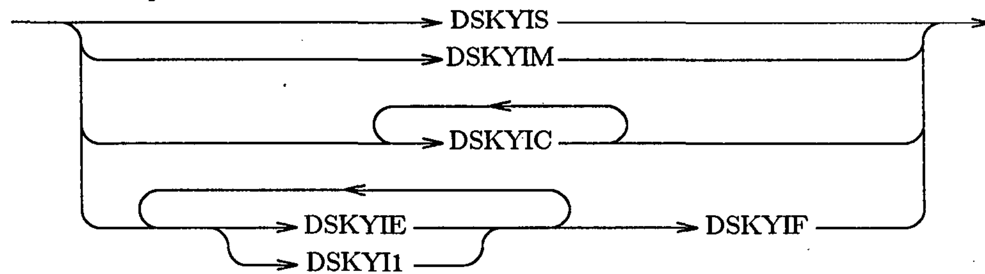


One Call Usage Solver for Skyline Storage:

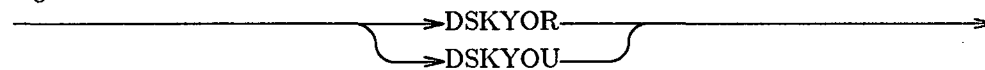
One Call Usage Solver for Column-Pointer, Row-Index Storage:

Initialization:

Matrix Structure Input:



Reordering:



Matrix Value Input:

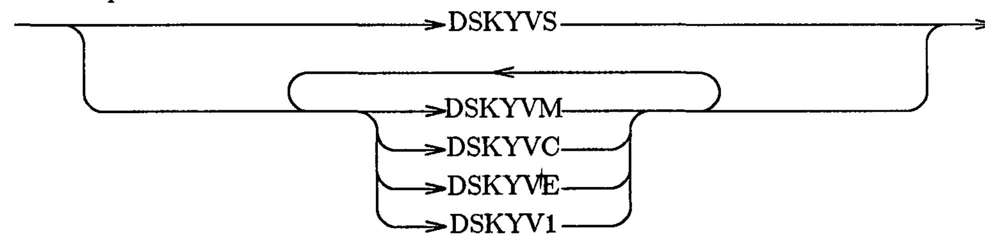
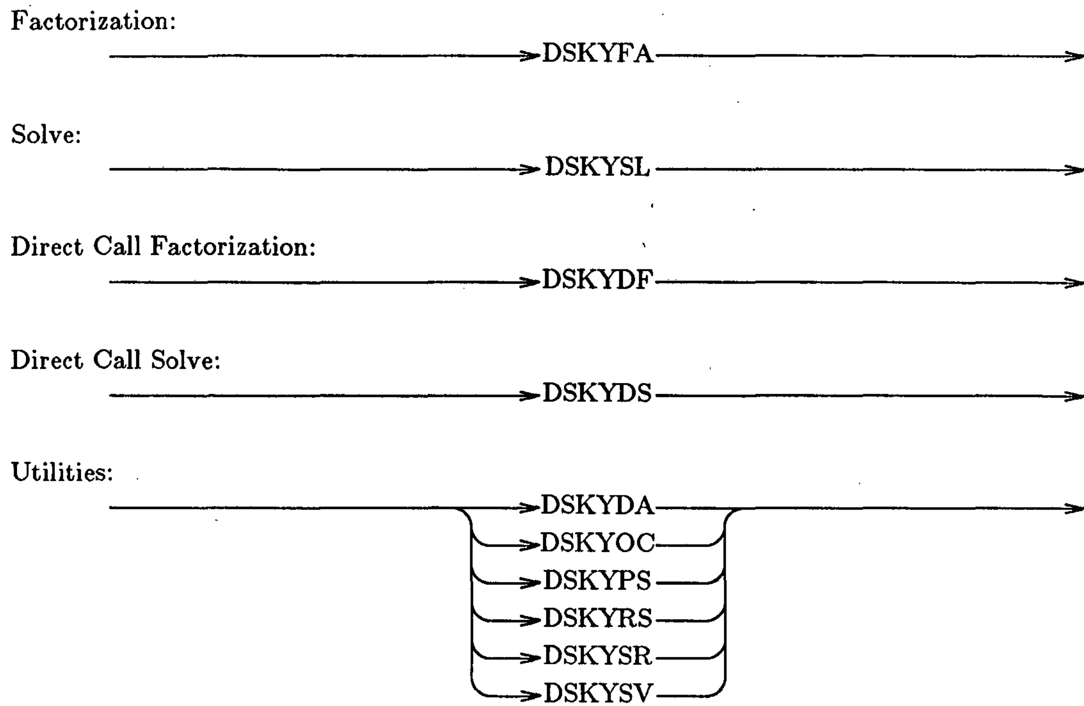


Figure 8-5: Paths of Control (Continued)



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. 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 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
          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       ----------------------------------------
C       ... USE THE SOLUTION STORED IN ARRAY RHS
C       ----------------------------------------

         .
         .
         .

C       --------------
C       ... ERROR TRAP
C       --------------

 8000   ..........
```

Supplemental Reading

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.

Subprogram Descriptions

One-Call Usage
DSKYFS 8-13

One-Call Usage
DSKYFX 8-15

Initialize Skyline Linear Equations
DSKYIN 8-18

Matrix Structure Input by Single Entry
DSKYI1 8-19

Matrix Structure Input by Column
DSKYIC 8-20

Matrix Structure Input by Finite Element
DSKYIE 8-21

Matrix Structure Input by Matrix
DSKYIM 8-22

Matrix Structure Input by Skyline Matrix
DSKYIS 8-24

End of Matrix Structure Input
DSKYIF 8-26

Automatic Reordering
DSKYOR 8-27

Automatic Reordering
DSKYOU 8-28

Matrix Value Input by Single Entry
DSKYV1 8-29

Matrix Value Input by Column
DSKYVC 8-30

Matrix Value Input by Finite Element
DSKYVE 8-32

Matrix Value Input by Matrix
DSKYVM 8-34

Matrix Value Input by Skyline Matrix
DSKYVS 8-36

Numeric Factorization
DSKYFA 8-38

Solve Right-Hand Side Vector
DSKYSL 8-39

Numeric Factorization
DSKYDF 8-40

Solve Right-Hand Side Vector
DSKYDS 8-42

Deallocate Working Storage
DSKYDA 8-44

Output Control
DSKYOC 8-45

Print Statistics
DSKYPS 8-46

Restore Problem State from a Savefile
DSKYRS 8-47

Retrieve Runtime Statistics
DSKYSR 8-48

Save Problem State to a Savefile
DSKYSV 8-49

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:

```
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)
```

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) indicates the position of the ith 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.

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

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.

Output

ier Status response:

ier = 0	Normal return.
ier = –200	Error in dynamic storage allocation during matrix structure input.
ier = –601	Error from DSKYIN.
ier = –602	Error from DSKYIS.
ier = –603	Error from DSKYOR.
ier = –604	Error from DSKYVS.
ier = –605	Error from DSKYFA.
ier = –606	Error from DSKYSL.

Notes

Calling DSKYFS is equivalent to calling DSKYIN, DSKYIS, DSKYOR, DSKYVS, DSKYFA, and DSKYSL in sequence. You may follow the call to DSKYFS with other calls to subprograms in the package to solve additional right-hand sides, to input new matrix values, to input a new matrix structure, or to perform utility functions.

Example

Solve the small (order 6) sparse system of linear equations where the matrix A and right-hand side b are

$$A = \begin{matrix} 4.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 4.0 & 1.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 1.0 & 4.0 & 1.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 4.0 & 1.0 & 1.0 \\ 0.0 & 1.0 & 1.0 & 1.0 & 4.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 4.0 \end{matrix} \qquad b = \begin{matrix} 1.0 \\ 0.0 \\ 1.0 \\ 1.0 \\ 0.0 \\ 0.0 \end{matrix}$$

Printed output will be to FORTRAN logical unit 6 and only printed error messages and some runtime statistics will be printed. Usage of DSKYFS is shown below.

```
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
```

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:

```
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)
```

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 j of the matrix specified by **matrix**. All of the nonzeros for column j are found, in ascending order, in **rowind(colstr(**j**))**, **rowind(colstr(**j**)+1)**, ..., **rowind(colstr(**$j+1$**)–1)**.

colstr(norder+1) must be set to one greater than the total number of nonzeros 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 A and right-hand side b are

$$A = \begin{matrix} 4.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 4.0 & 1.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 1.0 & 4.0 & 1.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 4.0 & 1.0 & 1.0 \\ 0.0 & 1.0 & 1.0 & 1.0 & 4.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 4.0 \end{matrix} \qquad b = \begin{matrix} 1.0 \\ 0.0 \\ 2.0 \\ 1.0 \\ 0.0 \\ 3.0 \end{matrix}$$

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)
MSGLVL = 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,MSGLVL,
&        OUTPUT,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

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
```

Matrix Structure Input by Single Entry — DSKYI1

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:**

```
INTEGER*4 irow, jcol, ier
REAL*8    global(150)
CALL DSKYI1 (irow, jcol, global, ier)
```

Input

irow — Row index of the nonzero entry; **jcol** ≤ **irow** ≤ **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 ≤ **jcol** ≤ **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:

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**.

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.

```
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
```

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:**

INTEGER*4 jcol, nzcol, jrowin(nzcol), ier
REAL*8 global(150)
CALL DSKYIC (jcol, nzcol, jrowin, global, ier)

Input

jcol Column index; 1 ≤ **jcol** ≤ **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; **jcol** ≤ **jrowin**(1) < **jrowin**(2) < ... < **jrowin(nzcol)** ≤ **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:

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.

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
```

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:**

```
INTEGER*4 nnode, nodlst(nnode), ier
REAL*8    global(150)
CALL DSKYIE (nnode, nodlst, global, ier)
```

Input

nnode Number of nodes in the finite element or clique.

nodlst List of nodes. All pairs **(nodlst(i),nodlst(j))** in the lower triangle are added to the sparsity structure of the matrix.

Updated

nodlst The order of the values in **nodlst** may be changed.

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = –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**.

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
```

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 Number of equations; **neqns** > 0.

colstr **colstr**(j) gives the index in **rowind** of the first nonzero in the lower triangular part of column j of the matrix specified by **matrix**. All of the nonzeros for column j are found, in ascending order, in **rowind(colstr(**j**))**, **rowind(colstr(**j**)+1)**, ..., **rowind(colstr(**$j+1$**)–1)**.

colstr(norder+1) must be set to one greater than the total number of nonzeros 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.

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
```

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 ith 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
```

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:**

```
INTEGER*4 ier
REAL*8    global(150)
CALL DSKYIF (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 = –210	Incorrect processing path; DSKYIN not called or matrix input already finished.
ier = –211	Error in dynamic storage allocation.

Example The nonzero structure of a pair of finite element matrices was passed to the package using repeated calls to subroutine DSKYIE. Signal that no more nonzeros will be added to the matrix.

```
INTEGER*4  IER
REAL*8     GLOBAL(150)
CALL DSKYIF (GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

Automatic Reordering — DSKYOR

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:**

INTEGER*4 ier
REAL*8 global(150)
CALL DSKYOR (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 = –300 Incorrect processing path; structure input not completed or value input subroutines already called.
ier = –301 Error in dynamic storage allocation.

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.

```
INTEGER*4 IER
REAL*8    GLOBAL(150)
CALL DSKYOR (GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

Purpose This subprogram accepts a reordering permutation provided by the user. It then constructs data structures that represent the factorization.

Usage **VECLIB:**

INTEGER*4 neqns, permut(neqns), ier
REAL*8 global(150)
CALL DSKYOU (permut, global, ier)

Input

permut Permutation vector: **permut(i)** gives the index of the row and column to which the ith row and column are to be permuted, i.e., matrix element $A_{i,j}$ is reordered to $(PAP)_{\mathbf{permut}(i),\mathbf{permut}(j)}$.

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:

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.

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
```

Matrix Value Input by Single Entry — DSKYV1

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:**

```
INTEGER*4 irow, jcol, ier
REAL*8    value, global(150)
CALL DSKYV1 (irow, jcol, value, global, ier)
```

Input

irow Row index of the nonzero entry, **jcol** ≤ **irow** ≤ **neqns**, where **neqns** is the number of equations specified in the call to DSKYIN.

jcol Column index of the nonzero entry, 1 ≤ **jcol** ≤ **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
```

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:

```
INTEGER*4 jcol, nzcol, jrowin(nzcol), ier
REAL*8    values(nzcol), global(150)
CALL DSKYVC (jcol, nzcol, jrowin, values, global, ier)
```

Input

jcol Column index; 1 ≤ **jcol** ≤ **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; **nzcol** ≥ 0.

jrowin List of ascending order row indices for nonzeros in the lower triangular part of column **jcol** of the matrix; **jcol** ≤ **jrowin**(1) < **jrowin**(2) < ... < **jrowin**(**nzcol**) ≤ **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:

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.

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
```

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:

```
INTEGER*4 nnode, ldelmx, nodlst(nnode), ier
REAL*8    elmmtx(ldelmx, nnode), global(150)
CALL DSKYVE (nnode, nodlst, elmmtx, ldelmx, global, ier)
```

Input

nnode Number of nodes in the finite element or clique.

nodlst List of nodes in element or clique. Values for all pairs (**nodlst(i)**, **nodlst(j)**) are added to the values of the matrix.

elmmtx Array containing values to be added to the matrix. Only the lower triangle (including the diagonal) of **elmmtx** is referenced. The value in **elmmtx(k, l)** is added to the value in position (**nodlst(k)**, **nodlst(l)**) in the sparse matrix.

ldelmx The leading dimension of array **elmmtx** as declared in the calling program unit, with **ldelmx** ≥ **nnode**.

Updated

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = –400	Incorrect processing path; 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
```

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:

```
INTEGER*4 colstr(neqns+1), rowind(nnzero), ier
REAL*8    values(nnzero), global(150)
CALL DSKYVM (colstr, rowind, values, global, ier)
```

Input

colstr — **colstr**(*j*) gives the index in **rowind** of the first nonzero in the lower triangular part of column j of the matrix specified by **matrix**. All of the nonzeros for column j are found, in ascending order, in **rowind(colstr(*j*))**, **rowind(colstr(*j*)+1)**, ..., **rowind(colstr(*j*+1)–1)**.

colstr(norder+1) must be set to one greater than the total number of nonzeros 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:

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. Normally, DSKYVM is used in conjunction with DSKYIM. Additional entries or modifications can be entered with DSKYVI1, 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(7),ROWIND(12),VALUES,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

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:

```
INTEGER*4 diag(neqns+1), idata, ier
REAL*8    values(nnzero), global(150)
CALL DSKYVS (values, diag, idata, global, ier)
```

Input

values Array of matrix values in skyline or reverse skyline order.

diag Array of diagonal positions. **diag**(i) indicates the position of the ith 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
```

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. Use a pivot tolerance of 0.1.

```
INTEGER*4 IER
REAL*8    GLOBAL(150)
CALL DSKYFA (GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

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.

$$b = \begin{matrix} 1.0 \\ 0.0 \\ 1.0 \\ 1.0 \\ 0.0 \\ 0.0 \end{matrix}$$

```
INTEGER*4 IER
REAL*8    RHS(6),GLOBAL(150)

RHS(1) = 1.0
RHS(2) = 0.0
RHS(3) = 1.0
RHS(4) = 1.0
RHS(5) = 0.0
RHS(6) = 0.0

CALL DSKYSL (RHS,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
 handle error condition
ENDIF
```

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 ith 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 order.
idata = 2	Array **values** is in reverse skyline order.

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

$$A = \begin{matrix} 4.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 4.0 & 1.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 1.0 & 4.0 & 1.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 4.0 & 1.0 & 1.0 \\ 0.0 & 1.0 & 1.0 & 1.0 & 4.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 4.0 \end{matrix} \qquad b = \begin{matrix} 1.0 \\ 0.0 \\ 1.0 \\ 1.0 \\ 0.0 \\ 0.0 \end{matrix}$$

```
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
```

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:**

```
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)
```

Input

values Array of matrix values from DSKYDF.

diag Array of diagonal positions. **diag**(i) indicates the position of the ith 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 order.
idata = 2	Array **values** is in reverse skyline order.

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.

$$b = \begin{matrix} 1.0 \\ 0.0 \\ 1.0 \\ 1.0 \\ 0.0 \\ 0.0 \end{matrix}$$

```
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
```

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:

```
INTEGER*4 ier
REAL*8    global(150)
CALL DSKYDA (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 = –701	Error in dynamic storage deallocation.

Example

Deallocate working storage after use of package.

```
REAL*8 GLOBAL(150)
CALL DSKYDA (GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
   handle error condition
ENDIF
```

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:

```
INTEGER*4 msglvl, output
REAL*8    global(150)
CALL DSKYOC (msglvl, output, global)
```

Input

msglvl Message level for printable output:

msglvl ≤ 0	Suppress all output.
msglvl = 1	Error messages and summary statistics.
msglvl = 2	More complete statistics.
msglvl = 3	First stage of debugging output.
msglvl = 4	Complete debugging output.

output FORTRAN logical unit number to which all output 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 output unit the same at 6.

```
REAL*8 GLOBAL(150)
CALL DSKYOC (2,6,GLOBAL)
```

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 range 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 after DSKYSL or DSKYFS).

```
REAL*8 GLOBAL(150)
CALL DSKYPS (GLOBAL)
```

Restore Problem State from a Savefile

DSKYRS

Purpose

This subprogram restores the working problem from the state stored on the user-specified I/O file by subprogram DSKYSV.

Usage

VECLIB:

```
INTEGER*4 svfile, ier
REAL*8    global(150)
CALL DSKYRS (svfile, global, ier)
```

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.

```
REAL*8 GLOBAL(150)
CALL DSKYRS (42,GLOBAL,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
ENDIF
```

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 after DSKYSL or DSKYFS).

```
INTEGER*4 INUSE,MXUSED,OPCNTS(2)
REAL*8    GLOBAL(150),TIME(6)
CALL DSKYSR (GLOBAL,INUSE,MXUSED,TIME,OPCNTS)
```

Save Problem State to a Savefile — DSKYSV

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:**

```
INTEGER*4 svfile, ier
REAL*8    global(150)
CALL DSKYSV (svfile, global, ier)
```

Input

svfile FORTRAN logical unit number of the file onto which the state is to be saved. The file will be rewound before and after use.

global Global communications array for this problem. This array must be passed, untouched by the user, to successive subroutines in this package.

Output

ier Status response:

ier = 0	Normal return.
ier = –902	I/O error detected by FORTRAN.

Example Save the current state on FORTRAN logical unit 42.

```
REAL*8 GLOBAL(150)
CALL 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

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. For one-dimensional transforms, the number of points must be a power of two:

$$l = 2^p, \quad p \geq 0.$$

For simultaneous and multidimensional transforms, 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 these restrictions 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.

Supplemental Reading

Brigham, E.O. *The Fast Fourier Transform.* Englewood Cliffs, NJ: Prentice-Hall, Inc. 1974.

Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, Inc. 1975.

Subprogram Descriptions

One-Dimensional FFT — Complex Storage Mode
C1DFFT, Z1DFFT 9-3

One-Dimensional FFT — Real Storage Mode
S1DFFT, D1DFFT 9-5

Two-Dimensional FFT — Complex Storage Mode
C2DFFT, Z2DFFT 9-8

Two-Dimensional FFT — Real Storage Mode
S2DFFT, D2DFFT 9-10

Three-Dimensional FFT — Complex Storage Mode
C3DFFT, Z3DFFT 9-12

Three-Dimensional FFT — Real Storage Mode
S3DFFT, D3DFFT 9-14

Simultaneous One-Dimensional FFT — Complex Storage Mode
CFFTS, ZFFTS 9-16

Simultaneous One-Dimensional FFT — Real Storage Mode
SFFTS, DFFTS 9-20

Real-to-Complex One-Dimensional FFT — Full Storage Mode
CRC1FT, ZRC1FT 9-24

Real-to-Complex One-Dimensional FFT — Storage Conserving Mode
SRC1FT, DRC1FT 9-26

Real-to-Complex Two-Dimensional FFT — Full Storage Mode
CRC2FT, ZRC2FT 9-29

Real-to-Complex Two-Dimensional FFT — Storage Conserving Mode
SRC2FT, DRC2FT 9-31

Real-to-Complex Three-Dimensional FFT — Full Storage Mode
CRC3FT, ZRC3FT 9-34

Real-to-Complex Three-Dimensional FFT — Storage Conserving Mode
SRC3FT, DRC3FT 9-37

Simultaneous Real-to-Complex 1-D FFT — Full Storage Mode
CRCFTS, ZRCFTS 9-40

Simultaneous Real-to-Complex 1-D FFT — Storage Conserving Mode
SRCFTS, DRCFTS 9-44

Purpose

Given an array of complex data, these subprograms compute the one-dimensional forward or inverse discrete Fourier transform using a radix 2 fast Fourier transform (FFT) algorithm. A pair of companion subprograms, S1DFFT and D1DFFT, performs the same operation, but with the complex data presented with real and imaginary parts in separate real arrays.

The one-dimensional forward discrete Fourier transform of $z(n)$, for $n = 1, 2, \ldots, l$, is defined by

$$Z(m) = \sum_{n=1}^{l} z(n)\, e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, \ldots, l$ and $i = \sqrt{-1}$.

Alternatively, the one-dimensional scaled inverse discrete Fourier transform of $Z(m)$, for $m = 1, 2, \ldots, l$, is defined by

$$z(n) = \frac{1}{l} \sum_{m=1}^{l} Z(m)\, e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, \ldots, l$.

Finally, the one-dimensional unscaled inverse discrete Fourier transform of $Z(m)$, for $m = 1, 2, \ldots, l$, is defined by

$$z(n) = \sum_{m=1}^{l} Z(m)\, e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, \ldots, l$.

These subprograms require that l be a power of 2, i.e., of the form $l = 2^p$, where $p \geq 0$.

Usage

Because it is common to use one data set length repetitively, these subprograms have a separate initialization call so that the setup can be performed only once for each different transform size. You will, therefore, always have at least two CALL statements to the FFT subprogram, using the same working storage array. Refer to "Example."

VECLIB:

```
INTEGER*4   l, iopt, ier
COMPLEX*8   z(l)
REAL*4      work(5*l/2)
CALL C1DFFT (z, l, work, iopt, ier)

INTEGER*4   l, iopt, ier
COMPLEX*16  z(l)
REAL*8      work(5*l/2)
CALL Z1DFFT (z, l, work, iopt, ier)
```

VECLIB8:

```
INTEGER*8   l, iopt, ier
COMPLEX*16  z(l)
REAL*8      work(5*l/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, 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** ≠ –3, **work** must have been initialized by a previous call with this value of **l** in which **iopt** was –3.

Output

z If **iopt** ≠ –3, transformed data replaces the input if **ier** = 0 is returned. Not used as output if **iopt** = –3.

ier Status response:

ier = 0 Normal return—transform or initialization successful.
ier = –1 **l** < 0.
ier = –2 **l** not a power of 2.
ier = –3 invalid value of **iopt**.

Example

Compute the forward discrete Fourier transform of two COMPLEX*8 data sets of length 1024. Here, length of working storage is 5 × 1024/2 = 2560.

```
INTEGER*4 L,IOPT,IER
COMPLEX*8 Z1(1024),Z2(1024)
REAL*4    WORK(5*1024/2)
L = 1024
IOPT = -3
CALL C1DFFT (Z1,L,WORK,IOPT,IER)       ! INITIALIZE
IF ( IER .NE. 0 ) THEN
  PRINT *,'IER =',IER,' FROM C1DFFT.'
  STOP
END IF
CALL C1DFFT (Z1,L,WORK,IOPT,IER)       ! FIRST TRANSFORM
IF ( IER .NE. 0 ) THEN
  PRINT *,'IER =',IER,' FROM C1DFFT.'
  STOP
END IF
CALL C1DFFT (Z2,L,WORK,IOPT,IER)       ! SECOND TRANSFORM
IF ( IER .NE. 0 ) THEN
  PRINT *,'IER =',IER,' FROM C1DFFT.'
  STOP
END IF
```

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 discrete Fourier transform using a radix 2 fast Fourier transform (FFT) algorithm. A pair of companion subprograms, C1DFFT and Z1DFFT, performs the same operation, but with the complex data presented in a complex array.

The one-dimensional forward discrete Fourier transform of $z(n)$, for $n = 1, 2, \ldots, l$, is defined by

$$Z(m) = \sum_{n=1}^{l} z(n)\, e^{-2\pi i (m-1)(n-1)/l}$$

for $m = 1, 2, \ldots, l$ and $i = \sqrt{-1}$.

Alternatively, the one-dimensional scaled inverse discrete Fourier transform of $Z(m)$, for $m = 1, 2, \ldots, l$, is defined by

$$z(n) = \frac{1}{l} \sum_{m=1}^{l} Z(m)\, e^{+2\pi i (m-1)(n-1)/l}$$

for $n = 1, 2, \ldots, l$.

Finally, the one-dimensional unscaled inverse discrete Fourier transform of $Z(m)$, for $m = 1, 2, \ldots, l$, is defined by

$$z(n) = \sum_{m=1}^{l} Z(m)\, e^{+2\pi i (m-1)(n-1)/l}$$

for $n = 1, 2, \ldots, l$.

These subprograms require that l be a power of 2, i.e., of the form $l = 2^p$, where $p \geq 0$.

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 so that the setup can be performed only once for each different transform size. You will, therefore, always have at least two **CALL** statements to the FFT subprogram, using the same working storage array. Refer to "Example."

VECLIB:

```
INTEGER*4 l, iopt, ier
REAL*4    x(l), y(l), work(5*l/2)
CALL S1DFFT (x, y, l, work, iopt, ier)

INTEGER*4 l, iopt, ier
REAL*8    x(l), y(l), work(5*l/2)
CALL D1DFFT (x, y, l, work, iopt, ier)
```

VECLIB8:

```
INTEGER*8 l, iopt, ier
REAL*8    x(l), y(l), work(5*l/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, 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** ≠ –3, **work** must have been initialized by a previous call with this value of **l** in which **iopt** was –3.

Output

x and **y** If **iopt** ≠ –3, the transformed data replaces the input if **ier** = 0 is returned. Not used as output if **iopt** = –3.

ier Status response:

ier = 0	Normal return—transform or initialization successful.
ier = –1	**l** < 0.
ier = –2	**l** not a power of 2.
ier = –3	invalid value of **iopt**.

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
```

Purpose

Given an array of complex data, these subprograms compute the two-dimensional forward or inverse discrete Fourier transform using a radix 2-3-5 fast Fourier transform (FFT) algorithm. A pair of companion subprograms, S2DFFT and D2DFFT, performs the same operation, but with the complex data presented with real and imaginary parts in separate real arrays.

The two-dimensional forward discrete Fourier transform of $z(n_1,n_2)$, for $n_1 = 1, 2, \ldots, l_1$ and $n_2 = 1, 2, \ldots, l_2$, is defined by

$$Z(m_1,m_2) = \sum_{n_1=1}^{l_1} \sum_{n_2=1}^{l_2} z(n_1,n_2)\, e^{-2\pi i(m_1-1)(n_1-1)/l_1}\, e^{-2\pi i(m_2-1)(n_2-1)/l_2}$$

for $m_1 = 1, 2, \ldots, l_1$, $m_2 = 1, 2, \ldots, l_2$, and $i = \sqrt{-1}$.

Alternatively, the two-dimensional inverse discrete Fourier transform of $Z(m_1,m_2)$, for $m_1 = 1, 2, \ldots, l_1$ and $m_2 = 1, 2, \ldots, l_2$, is defined by

$$z(n_1,n_2) = \frac{1}{l_1 l_2} \sum_{m_1=1}^{l_1} \sum_{m_2=1}^{l_2} Z(m_1,m_2)\, e^{+2\pi i(m_1-1)(n_1-1)/l_1}\, e^{+2\pi i(m_2-1)(n_2-1)/l_2}$$

for $n_1 = 1, 2, \ldots, l_1$ and $n_2 = 1, 2, \ldots, l_2$.

These subprograms require that l_1 and l_2 be products of powers of 2, 3, and 5, i.e., of the form

$$l_k = 2^{p_k} 3^{q_k} 5^{r_k},$$

where $p_k, q_k, r_k \geq 0$, $k=1,2$. Refer to "Notes" for a partial list of permissible values of l_1 and l_2.

Usage

VECLIB:

```
INTEGER*4   l1, l2, ldz, iopt, ier
COMPLEX*8   z(ldz, l2)
CALL C2DFFT (z, l1, l2, ldz, iopt, ier)

INTEGER*4   l1, l2, ldz, iopt, ier
COMPLEX*16  z(ldz, l2)
CALL Z2DFFT (z, l1, l2, ldz, iopt, ier)
```

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** ≥ **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**.

Notes

The following list indicates some of the permissible values of **l1** and **l2**. Although **l1** and **l2** can be any value of the form $2^p\ 3^q\ 5^r$ where $p,q,r \geq 0$, this list only shows the 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.

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 discrete Fourier transform using a radix 2-3-5 fast Fourier transform (FFT) algorithm. A pair of companion subprograms, C2DFFT and Z2DFFT, performs the same operation, but with the complex data presented in a complex array.

The two-dimensional forward discrete Fourier transform of $z(n_1,n_2)$, for $n_1 = 1, 2, ..., l_1$ and $n_2 = 1, 2, ..., l_2$, 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 $m_1 = 1, 2, ..., l_1$, $m_2 = 1, 2, ..., l_2$, and $i = \sqrt{-1}$.

Alternatively, the two-dimensional inverse discrete Fourier transform of $Z(m_1,m_2)$, for $m_1 = 1, 2, ..., l_1$ and $m_2 = 1, 2, ..., l_2$, 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 $n_1 = 1, 2, ..., l_1$ and $n_2 = 1, 2, ..., l_2$.

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. Refer to "Notes" for a partial list of permissible values of l_1 and l_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, ldxy, iopt, ier
REAL*8    x(ldxy, l2), y(ldxy, l2)
CALL S2DFFT (x, y, l1, l2, ldxy, iopt, ier)
```

Input

x Array of real parts of the data to be transformed.

y Array of imaginary parts of the data to be transformed.

l1 Number of rows of data, of the form $l1 = 2^{p_1} 3^{q_1} 5^{r_1}$, 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 $p_2, q_2, r_2 \geq 0$.

ldxy The leading dimension of arrays **x** and **y**, with **ldxy** ≥ **l1**.

iopt Option flag:

- **iopt** ≥ 0 Compute forward transform.
- **iopt** < 0 Compute inverse transform.

Output

x and **y** The transformed data replaces the input if **ier** = 0 is returned.

ier Status response:

- **ier** = 0 Normal return—transform successful.
- **ier** = −1 **l1** not of the required form.
- **ier** = −2 **l2** not of the required form.
- **ier** = −3 **ldxy** < **l1**.
- **ier** ≤ −4 Probable error in **ldxy** or dimensions of **x** and **y**.

Notes

The following list indicates some of the permissible values of **l1** and **l2**. Although **l1** and **l2** can be any value of the form $2^p\ 3^q\ 5^r$ where $p, q, r \geq 0$, this list only shows the 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.

Purpose

Given an array of complex data, these subprograms compute the three-dimensional forward or inverse discrete Fourier transform using a radix 2-3-5 fast Fourier transform (FFT) algorithm. A pair of companion subprograms, S3DFFT and D3DFFT, performs the same operation, but with the complex data presented with real and imaginary parts in separate real arrays.

The three-dimensional forward discrete Fourier transform of $z(n_1,n_2,n_3)$, for $n_1 = 1, 2, \ldots, l_1$, $n_2 = 1, 2, \ldots, l_2$, and $n_3 = 1, 2, \ldots, l_3$, is defined by

$$Z(m_1,m_2,m_3) = \sum_{n_1=1}^{l_1} \sum_{n_2=1}^{l_2} \sum_{n_3=1}^{l_3} z(n_1,n_2,n_3) \times e^{-2\pi i(m_1-1)(n_1-1)/l_1} e^{-2\pi i(m_2-1)(n_2-1)/l_2} e^{-2\pi i(m_3-1)(n_3-1)/l_3}$$

for $m_1 = 1, 2, \ldots, l_1$, $m_2 = 1, 2, \ldots, l_2$, $m_3 = 1, 2, \ldots, l_3$, and $i = \sqrt{-1}$.

Alternatively, the three-dimensional inverse discrete Fourier transform of $Z(m_1,m_2,m_3)$, for $m_1 = 1, 2, \ldots, l_1$, $m_2 = 1, 2, \ldots, l_2$, and $m_3 = 1, 2, \ldots, l_3$, is defined by

$$z(n_1,n_2,n_3) = \frac{1}{l_1 l_2 l_3} \sum_{m_1=1}^{l_1} \sum_{m_2=1}^{l_2} \sum_{m_3=1}^{l_3} Z(m_1,m_2,m_3) \times e^{+2\pi i(m_1-1)(n_1-1)/l_1} e^{+2\pi i(m_2-1)(n_2-1)/l_2} e^{+2\pi i(m_3-1)(n_3-1)/l_3}$$

for $n_1 = 1, 2, \ldots, l_1$, $n_2 = 1, 2, \ldots, l_2$, and $n_3 = 1, 2, \ldots, l_3$.

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 $p_k, q_k, r_k \geq 0$, $k = 1,2,3$. Refer to "Notes" for a partial list of permissible values of l_1, l_2, and l_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** ≥ **l1**.
	mdz	The middle dimension of array **z**, with **mdz** ≥ **l2**.
	iopt	Option flag: **iopt** ≥ 0 Compute forward transform. **iopt** < 0 Compute inverse transform.
Output	**z**	The transformed data replaces the input if **ier** = 0 is returned.
	ier	Status response: **ier** = 0 Normal return—transform successful. **ier** = –1 **l1** not of the required form. **ier** = –2 **l2** not of the required form. **ier** = –3 **l3** not of the required form. **ier** = –4 **ldz** < **l1**. **ier** = –5 **mdz** < **l2**. **ier** ≤ –6 Probable error in **ldz** or **mdz**.

Notes

The following list indicates some of the permissible values of **l1**, **l2**, and **l3**. Although **l1**, **l2**, and **l3** can be any value of the form $2^p\ 3^q\ 5^r$ where $p, q, r \geq 0$, this list only shows the 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.

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 discrete Fourier transform using a radix 2-3-5 fast Fourier transform (FFT) algorithm. A pair of companion subprograms, C3DFFT and Z3DFFT, performs the same operation, but with the complex data presented in a complex array.

The three-dimensional forward discrete Fourier transform of $z(n_1,n_2,n_3)$, for $n_1 = 1, 2, ..., l_1$, $n_2 = 1, 2, ..., l_2$, and $n_3 = 1, 2, ..., l_3$, is defined by

$$Z(m_1,m_2,m_3) = \sum_{n_1=1}^{l_1} \sum_{n_2=1}^{l_2} \sum_{n_3=1}^{l_3} z(n_1,n_2,n_3) \times e^{-2\pi i(m_1-1)(n_1-1)/l_1} e^{-2\pi i(m_2-1)(n_2-1)/l_2} e^{-2\pi i(m_3-1)(n_3-1)/l_3}$$

for $m_1 = 1, 2, ..., l_1$, $m_2 = 1, 2, ..., l_2$, $m_3 = 1, 2, ..., l_3$, and $i = \sqrt{-1}$.

Alternatively, the three-dimensional inverse discrete Fourier transform of $Z(m_1,m_2,m_3)$, for $m_1 = 1, 2, ..., l_1$, $m_2 = 1, 2, ..., l_2$, and $m_3 = 1, 2, ..., l_3$, is defined by

$$z(n_1,n_2,n_3) = \frac{1}{l_1 l_2 l_3} \sum_{m_1=1}^{l_1} \sum_{m_2=1}^{l_2} \sum_{m_3=1}^{l_3} Z(m_1,m_2,m_3) \times e^{+2\pi i(m_1-1)(n_1-1)/l_1} e^{+2\pi i(m_2-1)(n_2-1)/l_2} e^{+2\pi i(m_3-1)(n_3-1)/l_3}$$

for $n_1 = 1, 2, ..., l_1$, $n_2 = 1, 2, ..., l_2$, and $n_3 = 1, 2, ..., l_3$.

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 $p_k, q_k, r_k \geq 0$, $k = 1,2,3$. Refer to "Notes" for a partial list of permissible values of l_1, l_2, and l_3.

Usage

VECLIB:

```
INTEGER*4 l1, l2, l3, ldxy, mdxy, iopt, ier
REAL*4    x(ldxy, mdxy, l3), y(ldxy, mdxy, l3)
CALL S3DFFT (x, y, l1, l2, l3, ldxy, mdxy, iopt, ier)

INTEGER*4 l1, l2, l3, ldxy, mdxy, iopt, ier
REAL*8    x(ldxy, mdxy, l3), y(ldxy, mdxy, l3)
CALL D3DFFT (x, y, l1, l2, l3, ldxy, mdxy, iopt, ier)
```

VECLIB8:

```
INTEGER*8 l1, l2, l3, ldxy, mdxy, iopt, ier
REAL*8    x(ldxy, mdxy, l3), y(ldxy, mdxy, l3)
CALL S3DFFT (x, y, l1, l2, l3, ldxy, mdxy, iopt, ier)
```

Input

x	Array of real parts of the data to be transformed.
y	Array of imaginary parts of the data to be transformed.
l1	Number of rows of data, of the form **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 $p_3, q_3, r_3 \geq 0$.
ldxy	The leading dimension of arrays **x** and **y**, with **ldxy** ≥ **l1**.
mdxy	The middle dimension of arrays **x** and **y**, with **mdxy** ≥ **l2**.
iopt	Option flag: **iopt** ≥ 0 Compute forward transform. **iopt** < 0 Compute inverse transform.

Output

x and y	The transformed data replaces the input if **ier** = 0 is returned.
ier	Status response: **ier** = 0 Normal return—transform successful. **ier** = –1 **l1** not of the required form. **ier** = –2 **l2** not of the required form. **ier** = –3 **l3** not of the required form. **ier** = –4 **ldxy** < **l1**. **ier** = –5 **mdxy** < **l2**. **ier** ≤ –6 Probable error in **ldxy** or **mdxy**.

Notes

The following list indicates some of the permissible values of **l1**, **l2**, and **l3**. Although **l1**, **l2**, and **l3** can be any value of the form $2^p 3^q 5^r$ where $p,q,r \geq 0$, this list only shows the 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.

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 discrete Fourier transforms using a radix 2-3-5 fast Fourier transform (FFT) algorithm. A pair of companion subprograms, SFFTS and DFFTS, performs 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 discrete Fourier transform of a complex data set $z(n)$, for $n = 1, 2, ..., l$, is defined by

$$Z(m) = \sum_{n=1}^{l} z(n)\, e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, ..., l$ and $i = \sqrt{-1}$.

Alternatively, the one-dimensional scaled inverse discrete Fourier transform of a data set $Z(m)$, for $m = 1, 2, ..., l$, is defined by

$$z(n) = \frac{1}{l} \sum_{m=1}^{l} Z(m)\, e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, ..., l$.

These subprograms perform forward or inverse transform operations simultaneously on a number of data sets. They require that the length l of the data sets be a product of powers of 2, 3, and 5, i.e., of the form

$$l = 2^p\, 3^q\, 5^r,$$

where $p, q, r \geq 0$. Refer to "Notes" for a partial list of permissible values of l.

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

$$\textbf{lenz} = (\textbf{l}-1)\times\textbf{incl} + (\textbf{n}-1)\times\textbf{incn} + 1.$$

The i-th data point of the j-th data set, $1 \le i \le \textbf{l}$, $1 \le j \le \textbf{n}$, is stored in

$$\textbf{z}((i-1)\times\textbf{incl} + (j-1)\times\textbf{incn} + 1).$$

l Number of data points in each data set, of the form $\textbf{l} = 2^p\, 3^q\, 5^r$, with $p,q,r \ge 0$.

incl Storage increment between successive elements of the same data set, **incl** > 0. Use **incl** = 1 if each data set is stored contiguously in array **z**.

n The number of data sets, **n** > 0.

incn Storage increment between corresponding data points of successive data sets, **incn** > 0.

iopt Option flag:

iopt ≥ 0 Compute forward transform.
iopt < 0 Compute inverse transform.

Output

z The transformed data replaces the input if **ier** = 0 is returned. Unchanged if **ier** < 0.

ier Status response:

ier = 0 Normal return—transform successful.
ier = –1 **l** not of the form $2^p\, 3^q\, 5^r$ for $p,q,r \ge 0$.
ier = –2 **incl** ≤ 0.
ier = –3 **n** ≤ 0.
ier = –4 **incn** ≤ 0.
ier = –5 **l**, **incl**, **n**, and **incn** are incompatible. Refer to "Notes."

Notes

Typically, **z** will be a two- or three-dimensional array with each set of data being a one-dimensional section of the array, 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**, **l** ≤ **ldz**, **n** ≤ **mdz**, **incl** = 1, and **incn** = **ldz**. For example,

CALL CFFTS (z, l, 1, n, ldz, iopt, ier)

If **z** is a two-dimensional array as above and data sets are stored in rows of **z**, **l** ≤ **mdz**, **n** ≤ **ldz**, **incl** = **ldz**, and **incn** = 1. For example,

CALL CFFTS (z, l, ldz, n, 1, iopt, ier)

The subprograms generally are faster if the data sets are the rows of the array, so that **incn** = 1, rather than the columns. Lacking that, it is generally better to have an odd value of **incn**.

If **z** is a three-dimensional array of dimension **ldz** by **mdz** by **ndz**, then **incl** and **incn** will usually be 1, **ldz**, or **ldz**×**mdz**, depending on which of the subscripts of the three-dimensional array varies within a data set, which subscript varies between data

sets, and which remains constant. Specifically, if the subscript that varies within a data set is the

1st subscript, use **incl** = 1.
2nd subscript, use **incl** = **ldz**.
3rd subscript, use **incl** = **ldz**× **mdz**.

Similarly, if the subscript that varies between data sets is the

1st subscript, use **incn** = 1.
2nd subscript, use **incn** = **ldz**.
3rd subscript, use **incn** = **ldz**× **mdz**.

l, **incl**, **n**, and **incn** must be such that no two points of any data sets occupy the same element of **z**. These subprograms detect this situation and return **ier** = –5 if

$$\textbf{incl} < \textbf{n} \times \gcd(\textbf{incl},\textbf{incn})$$

and

$$\textbf{incn} < \textbf{l} \times \gcd(\textbf{incl},\textbf{incn})$$

where gcd(·,·) is the greatest common divisor.

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*8 data sets of length 1024. The data sets are stored as the columns of array Z whose dimensions are 1025 by 256.

```
INTEGER*4 L,INCL,N,INCN,IOPT,IER
COMPLEX*8 Z(1025,256)
L = 1024
INCL = 1
N = 256
INCN = 1025
IOPT = 1
CALL CFFTS (Z,L,INCL,N,INCN,IOPT,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
END IF
```

Example 2 Compute the inverse discrete Fourier transform of 1024 COMPLEX*8 data sets of length 256. The data sets are stored as the rows of array Z whose dimensions are 1025 by 256.

```
INTEGER*4 L,INCL,N,INCN,IOPT,IER
COMPLEX*8 Z(1025,256)
L = 256
INCL = 1025
N = 1024
INCN = 1
IOPT = -1
CALL CFFTS (Z,L,INCL,N,INCN,IOPT,IER)
IF ( IER .NE. 0 ) THEN
  handle error condition
END IF
```

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 discrete Fourier transforms using a radix 2-3-5 fast Fourier transform (FFT) algorithm. A pair of companion subprograms, CFFTS and ZFFTS, performs 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 $n = 1, 2, ..., l$, is defined by

$$Z(m) = \sum_{n=1}^{l} z(n)\, e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, ..., l$ and $i = \sqrt{-1}$.

Alternatively, the one-dimensional scaled inverse discrete Fourier transform of $Z(m)$, for $m = 1, 2, ..., l$, is defined by

$$z(n) = \frac{1}{l} \sum_{m=1}^{l} Z(m)\, e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, ..., l$.

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 $p, q, r \geq 0$. Refer to "Notes" for a partial list of permissible values of l.

The complex data, z or Z, are stored with real and imaginary parts in separate real arrays, **x** and **y**, respectively.

Usage

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

$$\mathbf{lenxy} = (\mathbf{l}-1)\times\mathbf{incl} + (\mathbf{n}-1)\times\mathbf{incn} + 1.$$

The real and imaginary parts of the i-th data point of the j-th data set, $1 \le i \le \mathbf{l}$, $1 \le j \le \mathbf{n}$, are stored in

$$\mathbf{x}((i-1)\times\mathbf{incl} + (j-1)\times\mathbf{incn} + 1)$$

and

$$\mathbf{y}((i-1)\times\mathbf{incl} + (j-1)\times\mathbf{incn} + 1),$$

respectively.

l Number of data points in each data set, of the form $\mathbf{l} = 2^p\ 3^q\ 5^r$, with $p,q,r \ge 0$.

incl Storage increment between successive elements of the same data set, **incl** > 0. Use **incl** $= 1$ if each data set is stored contiguously in arrays **x** and **y**.

n The number of data sets, **n** > 0.

incn Storage increment between corresponding data points of successive data sets, **incn** > 0.

iopt Option flag:

iopt ≥ 0 Compute forward transform.
iopt < 0 Compute inverse transform.

Output

x and y The transformed data replaces the input if **ier** $= 0$ is returned. Unchanged if **ier** < 0.

ier Status response:

ier $= 0$ Normal return—transform successful.
ier $= -1$ **l** not of the form $2^p\ 3^q\ 5^r$ for $p,q,r \ge 0$.
ier $= -2$ **incl** ≤ 0.
ier $= -3$ **n** ≤ 0.
ier $= -4$ **incn** ≤ 0.
ier $= -5$ **l**, **incl**, **n**, and **incn** are incompatible. Refer to "Notes."

Notes

Typically, **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**, **l** $\le$ **ldxy**, **n** $\le$ **mdxy**, **incl** $= 1$, and **incn** $=$ **ldxy**. For example:

```
CALL SFFTS (x, y, l, 1, n, ldxy, iopt, ier)
```

If **x** and **y** are two-dimensional arrays as above and data sets are stored in rows of **x** and **y**, **l** ≤ **mdxy**, **n** ≤ **ldxy**, **incl** = **ldxy**, and **incn** = 1. For example:

CALL SFFTS (x, y, l, ldxy, n, 1, iopt, ier)

The subprograms generally are faster if the data sets are the rows of the arrays, so that **incn** = 1, rather than the columns. Lacking that, it is generally better to have an odd value of **incn**.

If **x** and **y** are three-dimensional arrays of dimension **ldxy** by **mdxy** by **ndxy**, then **incl** and **incn** will usually be 1, **ldxy**, or **ldxy**×**mdxy**, depending on which of the subscripts of the three-dimensional array varies within a data set, which subscript varies between data sets, and which remains constant. Specifically, if the subscript that varies within a data set is the

1st subscript, use **incl** = 1.
2nd subscript, use **incl** = **ldxy**.
3rd subscript, use **incl** = **ldxy**×**mdxy**.

Similarly, if the subscript that varies between data sets is the

1st subscript, use **incn** = 1.
2nd subscript, use **incn** = **ldxy**.
3rd subscript, use **incn** = **ldxy**×**mdxy**.

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

incl < **n**× gcd(**incl**,**incn**)

and

incn < **l**× gcd(**incl**,**incn**)

where gcd(·,·) 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
```

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, discrete Fourier transform using a radix 2 fast Fourier transform (FFT) algorithm optimized for real input or output. A pair of companion subprograms, SRC1FT and DRC1FT, performs a similar operation, but in a space-conserving manner.

The one-dimensional forward discrete Fourier transform of a real data set, $z(n)$, for $n = 1, 2, \ldots, l$, is defined by

$$Z(m) = \sum_{n=1}^{l} z(n)\, e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, \ldots, l$ and $i = \sqrt{-1}$.

The sequence $Z(m)$ is conjugate-symmetric about $Z(l/2+1)$, i.e.,

$$Im(Z(1)) = Im(Z(l/2+1)) = 0$$

and

$$Z(l/2+1+m) = \overline{Z}(l/2+1-m), \quad m = 1, 2, \ldots, l/2-1,$$

where $\overline{Z}$ is the complex conjugate of Z.

Alternatively, if $Z(m)$, for $m = 1, 2, \ldots, l$, 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 $n = 1, 2, \ldots, l$.

Finally, the one-dimensional, real, unscaled, inverse discrete Fourier transform of the conjugate-symmetric complex data set $Z(m)$, for $m = 1, 2, \ldots, l$, is defined by

$$z(n) = \sum_{m=1}^{l} Z(m)\, e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, \ldots, l$.

These subprograms require that l be a power of 2, i.e., of the form $l = 2^p$, where $p \geq 0$.

Usage

Because it is common to use one data set length repetitively, these subprograms have a separate initialization call so that the setup can be performed only once for each different transform size. Therefore, you will always have at least two CALL statements to the FFT subprogram, using the same working storage array.

VECLIB:

```
        INTEGER*4  l, iopt, ier
        COMPLEX*8  z(l)
        REAL*4     work(3*l/2)
        CALL CRC1FT (z, l, work, iopt, ier)
```

```
INTEGER*4    l, iopt, ier
COMPLEX*16   z(l)
REAL*8       work(3*l/2)
CALL ZRC1FT (z, l, work, iopt, ier)
```

VECLIB8:

```
INTEGER*8    l, iopt, ier
COMPLEX*16   z(l)
REAL*8       work(3*l/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:

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**.

Purpose

Given a set of real data in a real array, these subprograms compute the nonredundant part of the complex one-dimensional forward discrete Fourier transform using a radix 2 fast Fourier transform (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 discrete Fourier transform of a real data set, $z(n)$, for $n = 1, 2, ..., l$, is defined by

$$Z(m) = \sum_{n=1}^{l} z(n)\, e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, ..., l$ and $i = \sqrt{-1}$.

The sequence $Z(m)$ is conjugate-symmetric about $Z(l/2+1)$, i.e.,

$$Im(Z(1)) = Im(Z(l/2+1)) = 0$$

and

$$Z(l/2+1+m) = \bar{Z}(l/2+1-m), \quad m = 1, 2, ..., l/2-1,$$

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

Alternatively, if $Z(m)$, for $m = 1, 2, ..., l$, is a conjugate-symmetric complex data set, the one-dimensional real scaled inverse discrete Fourier transform of $Z(m)$ is defined by

$$z(n) = \frac{1}{l} \sum_{m=1}^{l} Z(m)\, e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, ..., l$.

Finally, the one-dimensional real unscaled inverse discrete Fourier transform of the conjugate-symmetric complex data set $Z(m)$, for $m = 1, 2, ..., l$, is defined by

$$z(n) = \sum_{m=1}^{l} Z(m)\, e^{+2\pi i(m-1)(n-1)/l}$$

for $n = 1, 2, ..., l$.

These subprograms require that l be a power of 2, i.e., of the form $l = 2^p$, where $p \geq 0$.

Usage

Since it is common to use one data set length repetitively, these subprograms have a separate initialization call so that the setup can be performed only once for each different transform size. You will, therefore, always have at least two call statements to the FFT subprogram, using the same working storage array.

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, ..., l.

For an inverse complex-to-real transform, the real part of the i-th complex data point is stored in **x**(2×i–1) and the imaginary part of the i-th data point is stored in **x**(2×i), i = 1, 2, ..., 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** ≠ –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**$(2\times i)$, $i = 1, 2, ..., l/2+1$. If needed, the remaining $l/2 - 1$ complex output values may be formed by using the conjugate symmetry condition.

For an inverse complex-to-real transform, the i-th real output point is stored in **x**(i), $i = 1, 2, ..., l$.

Not used as output if **iopt** $= -3$.

ier Status response:

ier $= 0$ Normal return—transform or initialization successful.
ier $= -1$ $l < 0$.
ier $= -2$ l not a power of 2.
ier $= -3$ invalid value of **iopt**.

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 $n_1 = 1, 2, ..., l_1$ and $n_2 = 1, 2, ..., l_2$, 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 $m_1 = 1, 2, ..., l_1$, $m_2 = 1, 2, ..., l_2$, and $i = \sqrt{-1}$.

The $Z(m_1,m_2)$ satisfy the two-dimensional, conjugate-symmetry conditions:

$$Im(Z(1,1)) = 0,$$

$$Z(m_1,1) = \bar{Z}(l_1+2-m_1,1), \quad m_1 = 2, 3, ..., l_1,$$

$$Z(1,m_2) = \bar{Z}(1,l_2+2-m_1), \quad m_2 = 2, 3, ..., l_2,$$

and

$$Z(m_1,m_2) = \bar{Z}(l_1+2-m_1,l_2+2-m_2), \quad m_k = 2, 3, ..., l_k, \; k = 1,2,$$

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

Alternatively, if $Z(m_1,m_2)$, for $m_1 = 1, 2, ..., l_1$ and $m_2 = 1, 2, ..., l_2$, 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

$$z(n_1,n_2) = \frac{1}{l_1 l_2} \sum_{m_1=1}^{l_1} \sum_{m_2=1}^{l_2} Z(m_1,m_2)\, e^{+2\pi i(m_1-1)(n_1-1)/l_1}\, e^{+2\pi i(m_2-1)(n_2-1)/l_2}$$

for $n_1 = 1, 2, ..., l_1$ and $n_2 = 1, 2, ..., l_2$.

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, 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** ≥ **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**.

Notes

The following list indicates some of the permissible values of **l1**. Although **l1** can be any even value of the form $2^p\ 3^q\ 5^r$ where $q, r \geq 0$ and either **l1** = 1 or $p \geq 1$, this list only shows values not exceeding 1000:

1, 2, 4, 6, 8, 10, 12, 16, 18, 20, 24, 30, 32, 36, 40, 48, 50, 54, 60, 64, 72, 80, 90, 96, 100, 108, 120, 128, 144, 150, 160, 162, 180, 192, 200, 216, 240, 250, 256, 270, 288, 300, 320, 324, 360, 384, 400, 432, 450, 480, 486, 500, 512, 540, 576, 600, 640, 648, 720, 750, 768, 800, 810, 864, 900, 960, 972, and 1000.

The following list indicates some of the permissible values of **l2**. Although **l2** 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.

Purpose

Given a set of two-dimensional real data, these subprograms compute the nonredundant portion of the complex, two-dimensional, forward discrete Fourier transform using a radix 2-3-5 fast Fourier transform (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 discrete Fourier transform of a real data set, $z(n_1,n_2)$, for $n_1 = 1, 2, ..., l_1$ and $n_2 = 1, 2, ..., l_2$, 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 $m_1 = 1, 2, ..., l_1$, $m_2 = 1, 2, ..., l_2$, and $i = \sqrt{-1}$.

The $Z(m_1,m_2)$ satisfy the two-dimensional conjugate-symmetry conditions:

$$Im(Z(1,1)) = 0,$$

$$Z(m_1,1) = \overline{Z}(l_1+2-m_1,1), \quad m_1 = 2, 3, ..., l_1,$$

$$Z(1,m_2) = \overline{Z}(1,l_2+2-m_1), \quad m_2 = 2, 3, ..., l_2,$$

and

$$Z(m_1,m_2) = \overline{Z}(l_1+2-m_1,l_2+2-m_2), \quad m_k = 2, 3, ..., l_k, \; k = 1,2,$$

where $\overline{Z}$ is the complex conjugate of Z.

Alternatively, the two-dimensional, real, inverse discrete Fourier transform of $Z(m_1,m_2)$, for $m_1 = 1, 2, ..., l_1$ and $m_2 = 1, 2, ..., l_2$, 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 $n_1 = 1, 2, ..., l_1$ and $n_2 = 1, 2, ..., l_2$.

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, ..., $ **l1**, $n_2 = 1, 2, ..., $ **l2**.

For an inverse complex-to-real transform, the real part of $Z(m_1,m_2)$ is stored in **x**$(2\times m_1\text{-}1,m_2)$ and the imaginary part is stored in **x**$(2\times m_1,m_2)$, $m_1 = 1, 2, ...,$ **l1**$/2+1$, $m_2 = 1, 2, ...,$ **l2**.

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 $p_2,q_2,r_2 \geq 0$.

ldx The leading dimension of array **x**, with **ldx** ≥ **l1**+2.

iopt Option flag:

iopt ≥ 0 Compute forward transform.
iopt < 0 Compute inverse transform.

Output

x The transformed data replaces the input if **ier** = 0 is returned.

For a forward real-to-complex transform, the real part of $Z(m_1,m_2)$ is stored in **x**$(2\times m_1\text{-}1,m_2)$ and the imaginary part is stored in **x**$(2\times m_1,m_2)$, $m_1 = 1, 2, ...,$ **l1**$/2+1$, $m_2 = 1, 2, ...,$ **l2**. If needed, the remaining (**l1**/2 - 1) × **l2** complex output values may be formed by using the conjugate-symmetry condition.

For an inverse complex-to-real transform, the real output point $z(n_1,n_2)$ is stored in **x**(n_1,n_2), $n_1 = 1, 2, ...,$ **l1**, $n_2 = 1, 2, ...,$ **l2**.

ier Status response:

ier = 0 Normal return—transform successful.
ier = -1 **l1** not of the required form.
ier = -2 **l2** not of the required form.
ier = -3 **ldx** < **l1**+2.
ier ≤ -4 Probable error in **ldx** or dimensions of **x**.

Notes

The following list indicates some of the permissible values of **l1**. Although **l1** can be any even value of the form $2^p\ 3^q\ 5^r$ where $q,r \geq 0$ and either **l1** = 1 or $p \geq 1$, this list only shows values not exceeding 1000:

1, 2, 4, 6, 8, 10, 12, 16, 18, 20, 24, 30, 32, 36, 40, 48, 50, 54, 60, 64, 72, 80, 90, 96, 100, 108, 120, 128, 144, 150, 160, 162, 180, 192, 200, 216, 240, 250, 256, 270, 288, 300, 320, 324, 360, 384, 400, 432, 450, 480, 486, 500, 512, 540, 576, 600, 640, 648, 720, 750, 768, 800, 810, 864, 900, 960, 972, and 1000.

The following list indicates some of the permissible values of **l2**. Although **l2** 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.

Purpose

These subprograms compute either the forward real-to-complex or the inverse complex-to-real, three-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, SRC3FT and DRC3FT, performs a similar operation, but in a space-conserving manner.

The three-dimensional complex forward discrete Fourier transform of a real data set $z(n_1,n_2,n_3)$, for $n_1 = 1, 2, ..., l_1$, $n_2 = 1, 2, ..., l_2$, and $n_3 = 1, 2, ..., l_3$, is defined by

$$Z(m_1,m_2,m_3) = \sum_{n_1=1}^{l_1} \sum_{n_2=1}^{l_2} \sum_{n_3=1}^{l_3} z(n_1,n_2,n_3) \times e^{-2\pi i(m_1-1)(n_1-1)/l_1} e^{-2\pi i(m_2-1)(n_2-1)/l_2} e^{-2\pi i(m_3-1)(n_3-1)/l_3}$$

for $m_1 = 1, 2, ..., l_1$, $m_2 = 1, 2, ..., l_2$, $m_3 = 1, 2, ..., l_3$, and $i = \sqrt{-1}$.

The $Z(m_1,m_2,m_3)$ satisfy the three-dimensional conjugate-symmetry conditions:

$$Im(Z(1,1,1)) = 0,$$

$$Z(m_1,1,1) = \bar{Z}(l_1+2-m_1,1,1), \quad m_1 = 2, 3, ..., l_1,$$

$$Z(1,m_2,1) = \bar{Z}(1,l_2+2-m_2,1), \quad m_2 = 2, 3, ..., l_2,$$

$$Z(1,1,m_3) = \bar{Z}(1,1,l_3+2-m_3), \quad m_3 = 2, 3, ..., l_3,$$

$$Z(m_1,m_2,1) = \bar{Z}(l_1+2-m_1,l_2+2-m_2,1), \quad m_k = 2, 3, ..., l_k, \quad k = 1,2,$$

$$Z(m_1,1,m_3) = \bar{Z}(l_1+2-m_1,1,l_3+2-m_3), \quad m_k = 2, 3, ..., l_k, \quad k = 1,3,$$

$$Z(1,m_2,m_3) = \bar{Z}(1,l_2+2-m_2,l_3+2-m_3), \quad m_k = 2, 3, ..., l_k, \quad k = 2,3,$$

and

$$Z(m_1,m_2,m_3) = \bar{Z}(l_1+2-m_1,l_2+2-m_2,l_3+2-m_3), \quad m_k = 2, 3, ..., l_k, \quad k = 1,2,3,$$

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

Alternatively, if $Z(m_1,m_2,m_3)$, for $m_1 = 1, 2, ..., l_1$, $m_2 = 1, 2, ..., l_2$, and $m_3 = 1, 2, ..., l_3$, is a conjugate-symmetric complex data set, the three-dimensional, real, scaled, inverse discrete Fourier transform of $Z(m_1,m_2,m_3)$ is defined by

$$z(n_1,n_2,n_3) = \frac{1}{l_1 l_2 l_3} \sum_{m_1=1}^{l_1} \sum_{m_2=1}^{l_2} \sum_{m_3=1}^{l_3} Z(m_1,m_2,m_3) \times e^{+2\pi i(m_1-1)(n_1-1)/l_1} e^{+2\pi i(m_2-1)(n_2-1)/l_2} e^{+2\pi i(m_3-1)(n_3-1)/l_3}$$

for $n_1 = 1, 2, ..., l_1$, $n_2 = 1, 2, ..., l_2$, and $n_3 = 1, 2, ..., l_3$.

These subprograms require that l_1, l_2, and l_3 be products of powers of 2, 3, and 5, 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. Refer to "Notes" for a partial list of permissible values of l_1, l_2, and l_3.

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 **l1**/2+1 rows of **z** are used as input.

l1 Number of rows of data, of the form **l1** $= 2^{p_1} 3^{q_1} 5^{r_1}$, with $q_1, r_1 \geq 0$ and either **l1** $= 1$ or $p_1 \geq 1$.

l2 Number of columns of data, of the form **l2** $= 2^{p_2} 3^{q_2} 5^{r_2}$, with $p_2, q_2, r_2 \geq 0$.

l3 Number of planes of data, of the form **l3** $= 2^{p_3} 3^{q_3} 5^{r_3}$, with $p_3, q_3, r_3 \geq 0$.

ldz The leading dimension of array **z**, with **ldz** ≥ **l1**.

mdz The middle dimension of array **z**, with **mdz** ≥ **l2**.

iopt Option flag:

iopt ≥ 0 Compute forward transform.
iopt < 0 Compute inverse transform.

Output

z The **l1**-by-**l2**-by-**l3** 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 **l3** not of the required form.
ier = –4 **ldz** < **l1**.
ier = –5 **mdz** < **l2**.
ier ≤ –6 Probable error in **ldz** or **mdz**.

Notes

The following list indicates some of the permissible values of **l1**. Although **l1** can be any even value of the form $2^p\ 3^q\ 5^r$ where $q,r \geq 0$ and either **l1** $= 1$ or $p \geq 1$, this list only shows values not exceeding 1000:

1, 2, 4, 6, 8, 10, 12, 16, 18, 20, 24, 30, 32, 36, 40, 48, 50, 54, 60, 64, 72, 80, 90, 96, 100, 108, 120, 128, 144, 150, 160, 162, 180, 192, 200, 216, 240, 250, 256, 270, 288, 300, 320, 324, 360, 384, 400, 432, 450, 480, 486, 500, 512, 540, 576, 600, 640, 648, 720, 750, 768, 800, 810, 864, 900, 960, 972, and 1000.

The following list indicates some of the permissible values of **l2** and **l3**. Although **l2** and **l3** 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.

Purpose

Given a set of three-dimensional real data, these subprograms compute the nonredundant portion of the complex, three-dimensional forward discrete Fourier transform using a radix 2-3-5 fast Fourier transform (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 discrete Fourier transform 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 discrete Fourier transform of a real data set $z(n_1,n_2,n_3)$, for $n_1 = 1, 2, \ldots, l_1$, $n_2 = 1, 2, \ldots, l_2$, and $n_3 = 1, 2, \ldots, l_3$, is defined by

$$Z(m_1,m_2,m_3) = \sum_{n_1=1}^{l_1} \sum_{n_2=1}^{l_2} \sum_{n_3=1}^{l_3} z(n_1,n_2,n_3) \times e^{-2\pi i(m_1-1)(n_1-1)/l_1} e^{-2\pi i(m_2-1)(n_2-1)/l_2} e^{-2\pi i(m_3-1)(n_3-1)/l_3}$$

for $m_1 = 1, 2, \ldots, l_1$, $m_2 = 1, 2, \ldots, l_2$, $m_3 = 1, 2, \ldots, l_3$, and $i = \sqrt{-1}$.

The $Z(m_1,m_2,m_3)$ satisfy the three-dimensional conjugate-symmetry conditions:

$$Im(Z(1,1,1)) = 0,$$

$$Z(m_1,1,1) = \overline{Z}(l_1+2-m_1,1,1), \quad m_1 = 2, 3, \ldots, l_1,$$

$$Z(1,m_2,1) = \overline{Z}(1,l_2+2-m_1,1), \quad m_2 = 2, 3, \ldots, l_2,$$

$$Z(1,1,m_3) = \overline{Z}(1,1,l_3+2-m_1), \quad m_3 = 2, 3, \ldots, l_3,$$

$$Z(m_1,m_2,1) = \overline{Z}(l_1+2-m_1,l_2+2-m_2,1), \quad m_k = 2, 3, \ldots, l_k, \ k = 1,2,$$

$$Z(m_1,1,m_3) = \overline{Z}(l_1+2-m_1,1,l_3+2-m_3), \quad m_k = 2, 3, \ldots, l_k, \ k = 1,3,$$

$$Z(1,m_2,m_3) = \overline{Z}(1,l_2+2-m_2,l_3+2-m_3), \quad m_k = 2, 3, \ldots, l_k, \ k = 2,3,$$

and

$$Z(m_1,m_2,m_3) = \overline{Z}(l_1+2-m_1,l_2+2-m_2,l_3+2-m_3), \ m_k = 2, 3, \ldots, l_k, \ k = 1,2,3,$$

where $\overline{Z}$ is the complex conjugate of Z.

Alternatively, the three-dimensional, real, inverse discrete Fourier transform of $Z(m_1,m_2,m_3)$, for $m_1 = 1, 2, \ldots, l_1$, $m_2 = 1, 2, \ldots, l_2$, and $m_3 = 1, 2, \ldots, l_3$, is defined by

$$z(n_1,n_2,n_3) = \frac{1}{l_1 l_2 l_3} \sum_{m_1=1}^{l_1} \sum_{m_2=1}^{l_2} \sum_{m_3=1}^{l_3} Z(m_1,m_2,m_3) \times e^{+2\pi i(m_1-1)(n_1-1)/l_1} e^{+2\pi i(m_2-1)(n_2-1)/l_2} e^{+2\pi i(m_3-1)(n_3-1)/l_3}$$

for $n_1 = 1, 2, \ldots, l_1$, $n_2 = 1, 2, \ldots, l_2$, and $n_3 = 1, 2, \ldots, l_3$.

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. Refer to "Notes" for a partial list of permissible values of l_1, l_2, and l_3.

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, ..., $ **l1**, $n_2 = 1, 2, ...,$ **l2**, $n_3 = 1, 2, ...,2, ..., l3$. If needed, the remaining $(boldl/2 - 1) \times$ **l2** $\times$ **l3** 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 **x**$(2\times m_1 - 1, m_2, m_3)$ and the imaginary part is stored in **x**$(2\times m_1, m_2, m_3)$, $m_1 = 1, 2, ...,$ **l1**$/2+1$, $m_2 = 1, 2, ...,$ **l2**, $m_3 = 1, 2, ...,$ **l3**.

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 **x**, with **ldx** $\geq$ **l1**+2.

mdx The middle dimension of array **x**, with **mdx** $\geq$ **l2**.

iopt Option flag:

iopt ≥ 0 Compute forward transform.
iopt < 0 Compute inverse transform.

Output

x The transformed data replaces the input if **ier** = 0 is returned.

For a forward real-to-complex transform, the real part of $Z(m_1,m_2,m_3)$ is stored in $\mathbf{x}(2\times m_1\text{-}1,m_2,m_3)$ and the imaginary part is stored in $\mathbf{x}(2\times m_1,m_2,m_3)$, $m_1 = 1, 2, \ldots, \mathbf{l1}/2+1$, $m_2 = 1, 2, \ldots, \mathbf{l2}$, $m_3 = 1, 2, \ldots, \mathbf{l3}$.

For an inverse complex-to-real transform, the real output point $z(n_1,n_2,n_3)$ is stored in $\mathbf{x}(n_1,n_2,n_3)$, $n_1 = 1, 2, \ldots, \mathbf{l1}$, $n_2 = 1, 2, \ldots, \mathbf{l2}$, $n_3 = 1, 2, \ldots, \mathbf{l3}$.

ier Status response:

ier = 0 Normal return—transform successful.
ier = –1 **l1** not of the required form.
ier = –2 **l2** not of the required form.
ier = –3 **l3** not of the required form.
ier = –4 **ldx** < **l1**+2.
ier = –5 **mdx** < **l2**.
ier ≤ –6 Probable error in **ldx** or **mdx**.

Notes

The following list indicates some of the permissible values of **l1**. Although **l1** can be any even value of the form $2^p\ 3^q\ 5^r$ where $q,r \geq 0$ and either **l1** = 1 or $p \geq 1$, this list only shows values not exceeding 1000:

1, 2, 4, 6, 8, 10, 12, 16, 18, 20, 24, 30, 32, 36, 40, 48, 50, 54, 60, 64, 72, 80, 90, 96, 100, 108, 120, 128, 144, 150, 160, 162, 180, 192, 200, 216, 240, 250, 256, 270, 288, 300, 320, 324, 360, 384, 400, 432, 450, 480, 486, 500, 512, 540, 576, 600, 640, 648, 720, 750, 768, 800, 810, 864, 900, 960, 972, and 1000.

The following list indicates some of the permissible values of **l2** and **l3**. Although **l2** and **l3** 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.

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 fast Fourier transform (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 discrete Fourier transform of a real data set $z(n)$, for $n = 1, 2, \ldots, l$, is defined by

$$Z(m) = \sum_{n=1}^{l} z(n)\, e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, \ldots, l$ and $i = \sqrt{-1}$.

The sequence $Z(m)$ is conjugate-symmetric about $Z(l/2+1)$, i.e.,

$$Im(Z(1)) = Im(Z(l/2+1)) = 0$$

and

$$Z(l/2+1+m) = \bar{Z}(l/2+1-m), \quad m = 1, 2, \ldots, l/2-1,$$

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

Alternatively, if $Z(m)$, for $m = 1, 2, \ldots, l$, 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 $n = 1, 2, \ldots, l$.

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. Refer to "Notes" for a partial list of permissible values of l.

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

$$\mathbf{lenz} = (\mathbf{l}-1)\times \mathbf{incl} + (\mathbf{n}-1)\times \mathbf{incn} + 1.$$

The i-th data point of the j-th data set, $1 \le i \le \mathbf{l}$, $1 \le j \le \mathbf{n}$, is stored in

$$\mathbf{z}((i-1)\times \mathbf{incl} + (j-1)\times \mathbf{incn} + 1).$$

For a forward real-to-complex transform, only real parts of **z** are used as input.

For an inverse complex-to-real transform, only the first **l**/2+1 complex data points of each data set are used as input.

l Number of data points in each data set, of the form $\mathbf{l} = 2^p\,3^q\,5^r$, with $q, r \ge 0$ and either $\mathbf{l} = 1$ or $p \ge 1$.

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 **l** not of the required form.
ier = –2 **incl** ≤ 0.
ier = –3 **n** ≤ 0.
ier = –4 **incn** ≤ 0.
ier = –5 **l**, **incl**, **n**, and **incn** are incompatible. Refer to "Notes."

Notes

Typically, **z** is a two- or three-dimensional array with each set of data being a one-dimensional section of the array, that is, all but one subscript is constant within a data set.

If **z** is a two-dimensional array of dimension **ldz** by **mdz**, and if the data sets are stored in the columns of **z**, then **l** ≤ **ldz**, **n** ≤ **mdz**, **incl** = 1, and **incn** = **ldz**. For example, use

CALL CRCFTS (z, l, 1, n, ldz, iopt, ier)

If **z** is a two-dimensional array as above and the data sets are stored in the rows of **z**, then **l** ≤ **mdz**, **n** ≤ **ldz**, **incl** = **ldz**, and **incn** = 1. For example, use

CALL CRCFTS (z, l, ldz, n, 1, iopt, ier)

The subprograms generally are faster if the data sets are the rows of the array, so that **incn** = 1, rather than the columns. Lacking that, it is generally better to have an odd value of **incn**.

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

incl < **n**× gcd(**incl**,**incn**)

and

incn < **l**× gcd(**incl**,**incn**),

where gcd(·,·) is the greatest common divisor.

The following list indicates some of the permissible values of **l**. Although **l** can be any even value of the form $2^p\ 3^q\ 5^r$ where $q,r \geq 0$ and either **l** = 1 or $p \geq 1$, this list only shows the values not exceeding 1000:

1, 2, 4, 6, 8, 10, 12, 16, 18, 20, 24, 30, 32, 36, 40, 48, 50, 54, 60, 64, 72, 80, 90, 96, 100, 108, 120, 128, 144, 150, 160, 162, 180, 192, 200, 216, 240, 250, 256, 270, 288, 300, 320, 324, 360, 384, 400, 432, 450, 480, 486, 500, 512, 540, 576, 600, 640, 648, 720, 750, 768, 800, 810, 864, 900, 960, 972, and 1000.

Example 1 Compute the forward discrete Fourier transform of 256 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
```

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 $n = 1, 2, \ldots, l$, is defined by

$$Z(m) = \sum_{n=1}^{l} z(n)\, e^{-2\pi i(m-1)(n-1)/l}$$

for $m = 1, 2, \ldots, l$ and $i = \sqrt{-1}$.

The sequence $Z(m)$ is conjugate-symmetric about $Z(l/2+1)$, i.e.,

$$Im(Z(1)) = Im(Z(l/2 + 1)) = 0$$

and

$$Z(l/2+1 + m) = \bar{Z}(l/2+1 - m), \quad m = 1, 2, \ldots, l/2-1,$$

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 $m = 1, 2, \ldots, l$, 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 $n = 1, 2, \ldots, l$. 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. Refer to "Notes" for a partial list of permissible values of l.

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
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 **l**/2+1 complex data points of a conjugate-symmetric complex data set of length **l**, to be transformed. Typically, **x** is a two- or three-dimensional array with each set of data being a one-dimensional array section. Refer to "Notes" for suggested usages.

Treating **x** as a one-dimensional array, results in

$$\mathbf{lenx} = (\mathbf{l}+1)\times \mathbf{incl} + (\mathbf{n}-1)\times \mathbf{incn} + 1.$$

For a forward real-to-complex transform, the i-th real data point of the j-th data set, $1 \le i \le \mathbf{l}$, $1 \le j \le \mathbf{n}$, is stored in

$$\mathbf{x}((i-1)\times \mathbf{incl} + (j-1)\times \mathbf{incn} + 1).$$

For an inverse complex-to-real transform, the real part of the i-th data point of the j-th data set, $1 \le i \le \mathbf{l}/2+1$, $1 \le j \le \mathbf{n}$, is stored in

$$\mathbf{x}((2\times i-2)\times \mathbf{incl} + (j-1)\times \mathbf{incn} + 1)$$

and the imaginary part is stored in

$$\mathbf{x}((2\times i-1)\times \mathbf{incl} + (j-1)\times \mathbf{incn} + 1),$$

respectively.

l Number of data points in each complete data set, of the form $\mathbf{l} = 2^p\ 3^q\ 5^r$, with $q,r \ge 0$ and either $\mathbf{l} = 1$ or $p \ge 1$.

incl Storage increment between successive elements of the same data set, **incl** > 0. **incl** = 1 means that the data points of a real data set are stored contiguously in array **x**, or that the real and imaginary parts of the data points of a complex data set are stored alternatingly in contiguous elements of **x**.

n The number of data sets, **n** > 0.

incn Storage increment between corresponding data points of successive data sets, **incn** > 0.

iopt Option flag:

iopt ≥ 0 Compute forward transform.
iopt < 0 Compute inverse transform.

Output

x The transformed data replaces the input if **ier** = 0 is returned. Unchanged if **ier** < 0.

For a forward real-to-complex transform, the real part of the i-th output point of the j-th data set, $1 \le i \le \mathbf{l}/2+1$, $1 \le j \le \mathbf{n}$, is stored in

$$\mathbf{x}((2\times i-2)\times \mathbf{incl} + (j-1)\times \mathbf{incn} + 1)$$

and the imaginary part is stored in

$$\mathbf{x}((2\times i-1)\times \mathbf{incl} + (j-1)\times \mathbf{incn} + 1),$$

respectively. If needed, the remaining $(\mathbf{l}/2 - 1) \times \mathbf{n}$ complex output values may be formed by using the conjugate-symmetry condition.

For an inverse complex-to-real transform, the i-th real output point of the j-th data set, $1 \le i \le \mathbf{l}$, $1 \le j \le \mathbf{n}$, is stored in

$$\mathbf{x}((i-1)\times \mathbf{incl} + (j-1)\times \mathbf{incn} + 1).$$

ier Status response:

ier = 0	Normal return—transform successful.
ier = -1	**l** not of the required form.
ier = -2	**incl** ≤ 0.
ier = -3	**n** ≤ 0.
ier = -4	**incn** ≤ 0.
ier = -5	**l**, **incl**, **n**, and **incn** are incompatible. Refer to "Notes."

Notes

Typically, **x** will be a two- or three-dimensional array with each set of data being a one-dimensional section of the array, 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** ≤ **ldx**, **n** ≤ **mdx**, **incl** = 1, and **incn** = **ldx**. For example,

CALL SRCFTS (x, l, 1, n, ldxy, iopt, ier)

If **x** is a two-dimensional array as above and the data sets are stored in the rows of **x**, then **l+2** ≤ **mdx**, **n** ≤ **ldx**, **incl** = **ldx**, and **incn** = 1. For example,

CALL SRCFTS (x, l, ldxy, n, 1, iopt, ier)

The subprograms generally are faster if data sets are rows of the arrays, so that **incn** = 1, rather than the columns. Lacking that, it is generally better to have an odd value of **incn**.

If **x** is a three-dimensional array of dimension **ldx** by **mdx** by **ndx**, then **incl** and **incn** will usually be 1, **ldx**, or **ldx**×**mdx**, depending on which of the subscripts of the three-dimensional array varies within a data set, which subscript varies between data sets, and which remains constant. Specifically, if the subscript that varies within a data set is the

1st subscript, use **incl** = 1.
2nd subscript, use **incl** = **ldx**.
3rd subscript, use **incl** = **ldx**×**mdx**.

Similarly, if the subscript that varies between data sets is the

1st subscript, use **incn** = 1.
2nd subscript, use **incn** = **ldx**.
3rd subscript, use **incn** = **ldx**×**mdx**.

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

incl < **n**× gcd(**incl**,**incn**)

and

incn < (**l**+ 2)× gcd(**incl**,**incn**),

where gcd(·,·) is the greatest common divisor.

The following list indicates some of the permissible values of **l**. Although **l** can be any even value of the form $2^p\ 3^q\ 5^r$ where $q,r \geq 0$ and either **l** = 1 or $p \geq 1$, this list only shows values not exceeding 1000:

1, 2, 4, 6, 8, 10, 12, 16, 18, 20, 24, 30, 32, 36, 40, 48, 50, 54, 60, 64, 72, 80, 90, 96, 100, 108, 120, 128, 144, 150, 160, 162, 180, 192, 200, 216, 240, 250, 256, 270, 288, 300, 320, 324, 360, 384, 400, 432, 450, 480, 486, 500, 512, 540, 576, 600, 640, 648, 720, 750, 768, 800, 810, 864, 900, 960, 972, and 1000.

Example 1

Compute the forward discrete Fourier transform of 256 real data sets of length 1024. 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 conjugate-symmetric complex data sets of length 256. The real and imaginary parts of the first 513 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.

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.

Supplemental Reading

Rabiner, L.R., and B. Gold. *Theory and Application of Digital Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, Inc. 1975.

Subprogram Descriptions

Discrete Correlation and Convolution
SCONV, DCONV 10-2

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, $i = 1, 2, \ldots, l$ and w_j, $j = 1, 2, \ldots, n$ are two data sequences, their discrete correlation y_k is defined by

$$y_k = \sum_i w_{i-k} x_i,$$

for $k = -n+1, -n+2, \ldots, l-1$, where the sum is taken over all indices i for which both w_{i-k} and x_i are defined.

Again, if x_i, $i = 1, 2, \ldots, l$ and w_j, $j = 1, 2, \ldots, n$ are two data sequences, their discrete convolution z_k is defined by

$$z_k = \sum_i w_{k-i+1} x_i,$$

for $k = 1, 2, \ldots, l+n-1$, 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 $k = 1, 2, \ldots, m$, 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 $m+n-1$. **lenx** = (m+n−2)×|incx|+1.

incx Increment for the array **x**, **incx** ≠ 0. x_i is stored in **x**((i-1)×**incx**+1). **incx** is normally positive whether computing the correlation or the convolution. Use **incx** = 1 if the vector x is stored contiguously in array **x**, 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. **lenw** = (**n**–1)× |**incw**|+1. Refer to the description of **incw** for alternate usage of **w**.

incw Increment for the array **w**, **incw** ≠ 0. w_j is stored in **w**((j-1)×**incw**+1). **incw** is normally positive to compute the correlation and negative to compute the convolution.

Use **incw** = 1 if computing the correlation and vector w is stored contiguously in forward order in array **w**, i.e., if w_i is stored in **w**(i). Refer to "Example 1." Also, use **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 **incw** = –1 if computing the convolution and vector w is stored contiguously in forward order 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**, **incy** ≠ 0. y_k is stored in **y**((k-1)×**incy**+1). **incy** will normally be positive whether computing the correlation or the convolution. Use **incy** = 1 if the vector y is stored contiguously in array **y**, i.e., if y_i is stored in **y**(i). Refer to "Notes."

m The length of the y vector.

n The length of the w vector.

Output

y The convolution or correlation vector y of length m. **leny** = (**m**–1)× |**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 $\bar{x} = (0, 1, 3, 5, 0)$. Then, you have $n = 2$ and $m + n - 1 = 5$, so $m = 4$.

```
INTEGER*4 INCX,INCW,INCY,M,N
REAL*4    X(5),W(2),Y(4)
DATA      X / 0.0 , 1.0 , 3.0 , 5.0, 0.0 /
DATA      W / 2.0 , 1.0 /
M = 4
N = 2
INCX = 1
INCW = -1
INCY = 1
CALL SCONV (X,INCX,W(N),INCW,Y,INCY,M,N)
```

The result is $y = (2, 7, 13, 5)$.

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

A recurrence is a loop-carried data dependency between a value calculated during one iteration of a loop and used during a subsequent iteration. When the FORTRAN compiler detects a recurrence, it cannot vectorize the loop. Therefore, these subprograms, which use special, vectorizable algorithms, can be used to optimize FORTRAN loops containing certain linear recurrences.

Subprogram Descriptions

Solve a First Order Linear Recurrence
SFLR1M, SFLR1P, DFLR1M, DFLR1P 11-2

Solve a First Order Linear Recurrence with Constant Coefficient
SFLR1C, DFLR1C 11-5

Solve a First Order Linear Recurrence
SFLR2M, SFLR2P, DFLR2M, DFLR2P 11-7

Solve a First Order Linear Recurrence with Constant Coefficient
SFLR2C, DFLR2C 11-10

Solve for the Last Term of a First Order Linear Recurrence
SFLRLM, SFLRLP, DFLRLM, DFLRLP 11-13

Compute the Vector of Partial Products of a Vector
SPPROD, DPPROD 11-16

Compute the Vector of Partial Sums of a Vector
SPSUM, DPSUM, IPSUM 11-18

Solve for the Last Term of a Second Order Linear Recurrence
SSLRL, DSLRL 11-20

Solve a Second Order Linear Recurrence
SSLR2, DSLR2 11-23

Solve a Second Order Linear Recurrence
SSLR3, DSLR3 11-26

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

$$y_1 = x_1$$
$$y_i = x_i \pm a_i y_{i-1}, \quad i = 2, 3, \ldots, n,$$

overwriting the input x vector with the resulting y vector.

The operation indicated by "±" above is specified by the subprogram name used:

SFLR1M or DFLR1M "±" represents "–": $y_i = x_i - a_i y_{i-1}$
SFLR1P or DFLR1P "±" 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** ≥ 0. If **n** = 0, the subprograms do not reference **a** or **x**.

a Array of length **lena** = (**n**–1)×|**inca**|+1 containing the n-vector a.

inca Increment for the array **a**, **inca** ≠ 0:

inca > 0 a is stored forward in array **a**, i.e., a_i is stored in **a**((i–1)×**inca**+1).
inca < 0 a is stored backward in array **a**, i.e., a_i is stored in **a**((i–**n**)×**inca**+1).

Use **inca** = 1 if the vector a is stored contiguously in array **a**, i.e., if a_i is stored in **a**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

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

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

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

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

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

$$y_1 = x_1$$
$$y_i = x_i - a_i y_{i-1}, \quad i = 2, 3, \ldots, n,$$

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

```
INTEGER*4 N,INCA,INCX
REAL*8    A(20),X(20)
N = 10
INCA = 1
INCX = 1
CALL DFLR1M (N,A,INCA,X,INCX)
```

Purpose

Given a real coefficient α and a real vector x of length n, these subprograms solve the first order linear recurrence

$$y_1 = x_1$$
$$y_i = x_i + \alpha y_{i-1}, \quad i = 2, 3, \ldots, n.$$

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** ≥ 0. If **n** = 0, the subprogram does not reference **x**.

alpha The scalar α.

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

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

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

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

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

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
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

$$x_1 = a_1$$
$$x_i = a_i + 4x_{i-1}, \quad i = 2, 3, \ldots, n,$$

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)
```

Purpose

Given real vectors a and x of length n, these subprograms solve the first order linear recurrence

$$y_1 = x_1$$
$$y_i = x_i \pm a_i y_{i-1}, \quad i = 2, 3, \ldots, n,$$

for the y vector.

The operation indicated by ± above is specified by the subprogram name used:

SFLR2M or DLFR2M ± represents −: $y_i = x_i - a_i y_{i-1}$
SFLR2P or DLFR2P ± 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** ≥ 0. If **n** = 0, the subprograms do not reference **a**, **x**, or **y**.

a Array of length **lena** = (n−1)×|**inca**|+1 containing the n-vector a.

inca Increment for the array **a**, **inca** ≠ 0:

inca > 0 a is stored forward in array **a**, i.e., a_i is stored in **a**((i−1)×**inca**+1).
inca < 0 a is stored backward in array **a**, i.e., a_i is stored in **a**((i−n)×**inca**+1).

Use **inca** = 1 if the vector a is stored contiguously in array **a**, i.e., if a_i is stored in **a**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

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

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

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

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

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

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

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

Output

y Array of length **leny** = (**n**–1)× |**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

$$y_1 = x_1$$
$$y_i = x_i - a_i y_{i-1}, \quad i = 2, 3, \ldots, n,$$

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

```
      INTEGER*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)
```

Purpose Given a real coefficient α and a real vector x of length n, these subprograms solve the first order linear recurrence

$$y_1 = x_1$$
$$y_i = x_i + \alpha y_{i-1}, \quad i = 2, 3, \ldots, n.$$

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

Usage

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** ≥ 0. If **n** $= 0$, the subprogram does not reference **x** or **y**.

alpha The scalar α.

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

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

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

incx < 0 x is stored backward in array **x**, i.e., x_i is stored in **x**$((i-\mathbf{n})\times$ **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.

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**$((i-\mathbf{n})\times$ **incy**$+1)$.

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

Output

y Array of length **leny** = (**n**−1)× |**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

$$x_1 = a_1$$
$$x_i = a_i + 4x_{i-1}, \quad i = 2, 3, \ldots, n,$$

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)
```

Last Term of First Order Linear Recurrence — SFLRLM/.../DFLRLP

Purpose

Given real vectors a and x of length n, these subprograms solve for the last term of the first order linear recurrence

$$y_1 = x_1$$
$$y_i = x_i \pm a_i y_{i-1}, \quad i = 2, 3, \ldots, n,$$

i.e., returning y_n.

The operation indicated by ± above is specified by the subprogram name used:

SFLRLM or DFLRLM	± represents −:	$y_i = x_i - a_i y_{i-1}$
SFLRLP or DFLRLP	± 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** ≥ 0. If **n** = 0, the subprograms do not reference **a** or **x**.

a Array of length **lena** = (**n**−1)×|**inca**|+1 containing the n-vector a.

inca Increment for the array **a**, **inca** ≠ 0:

- **inca** > 0: a is stored forward in array **a**, i.e., a_i is stored in **a**((i−1)×**inca**+1).
- **inca** < 0: a is stored backward in array **a**, i.e., a_i is stored in **a**((i−**n**)×**inca**+1).

Use **inca** = 1 if the vector a is stored contiguously in array **a**, i.e., if a_i is stored in **a**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

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

incx Increment for the array **x**:

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

If **incx** = 0, then x_i = **x**(1) for all i. Refer to "Notes" to see how to use SFLRLP with **incx** = 0 to evaluate a polynomial. Use **incx** = 1 if the vector x is stored contiguously in array **x**, i.e., if x_i is stored in **x**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

Output **yn** If **n** = 0, then **yn** = 0. Otherwise, the last term of the recurrence's solution is returned.

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

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

$$y_0 = a_n$$
$$y_i = a_{n-i} + y_{i-1}x, \quad i = 1, 2, \ldots, n.$$

Refer to "Example 2".

If an error in the arguments is detected, the subprograms call error handler 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
inca = 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
      END IF
      IA = 1 + INCA
      IX = 1 + INCX
      IF ( INCA .LT. 0 ) IA = 1 - (N-2) * INCA
      IF ( INCX .LT. 0 ) IX = 1 - (N-2) * INCX
      FOLRN = X(IX-INCX)
      DO 10 I = 2, N
         FOLRN = X(IX) - A(IA) * FOLRN
         IA = IA + INCA
         IX = IX + INCX
   10 CONTINUE
      RETURN
      END
```

Example 1 Solve for the last term of the REAL*8 first order linear recurrence

$$y_1 = x_1$$
$$y_i = x_i - a_i y_{i-1}, \quad i = 2, 3, \ldots, n,$$

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

```
      INTEGER*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)
```

Purpose Given real vector x of length n, these subprograms compute the n-vector y of partial products

$$x_i = \prod_{j=1}^{i} a_j, \quad i = 1, 2, \ldots, n.$$

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

Usage

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** ≥ 0. If **n** = 0, the subprograms do not reference **x** or **y**.

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

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

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

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

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

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

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

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

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

Output

y Array of length **leny** = (**n**–1)×|**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

$$\mathbf{n} < 0,$$
$$\mathbf{incx} = 0, \text{ and}$$
$$\mathbf{incy} = 0.$$

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)
```

Purpose

Given real vector x of length n, these subprograms computes the n-vector y of partial sums

$$y_i = \sum_{j=1}^{i} x_j, \quad i = 1, 2, \ldots, n.$$

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

Usage

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, $\mathbf{n} \geq 0$. If $\mathbf{n} = 0$, the subprograms do not reference **x** or **y**.

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

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

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

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

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

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

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

Output

y Array of length **leny** = (**n**–1)×|**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,
incx = 0, and
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)
```

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

$$x_i = a_{i-2}x_{i-1} + b_{i-2}x_{i-2}, \quad i = 3, 4, \ldots, n,$$

i.e., returning x_n.

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

Usage

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** ≥ 0. If **n** = 0, the subprogram does not reference **a**, **b**, or **x**.

a Array of length **lena** = (**n**−1)×|**inca**|+1 containing the n-vector a.

inca Increment for the array **a**, **inca** ≠ 0:

inca > 0 a is stored forward in array **a**, i.e., a_i is stored in **a**((i−1)×**inca**+1).

inca < 0 a is stored backward in array **a**, i.e., a_i is stored in **a**((i−**n**)×**inca**+1).

Use **inca** = 1 if the vector a is stored contiguously in array **a**, i.e., if a_i is stored in **a**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

b Array of length **lenb** = (**n**−1)×|**incb**|+1 containing the n-vector b.

incb Increment for the array **b**, **incb** ≠ 0:

incb > 0 b is stored forward in array **b**, i.e., b_i is stored in **b**((i−1)×**incb**+1).

incb < 0 b is stored backward in array **b**, i.e., b_i is stored in **b**((i−**n**)×**incb**+1).

Use **incb** = 1 if the vector b is stored contiguously in array **b**, i.e., if b_i is stored in **b**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the first two elements, x_1 and x_2 of the n-vector x.

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

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

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

Output

xn If **n** $= 0$, then **xn** $= 0$. Otherwise, the last term of the recurrence's solution is returned.

x If **n** $= 0$, then **x** is unchanged. Otherwise, **x** is overwritten with intermediate results. These intermediate results are not necessarily what is shown in "FORTRAN Equivalent".

Notes

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

If an error in the arguments is detected, the subprograms call error handler 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

$$x_i = a_{i-2}x_{i-1} + b_{i-2}x_{i-2}, \quad i = 3, 4, \ldots, n,$$

where a, b, and x are vectors 10 elements long stored in one-dimensional arrays A, B, and X of dimension 20.

```
      INTEGER*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)
```

Second Order Linear Recurrence — SSLR2/DSLR2

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

$$x_i = a_{i-2}x_{i-1} + b_{i-2}x_{i-2}, \quad i = 3, 4, \ldots, n.$$

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

Usage

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** ≥ 0. If **n** = 0, the subprogram does not reference **a**, **b**, or **x**.

a Array of length **lena** = (**n**−1)×|**inca**|+1 containing the n-vector a.

inca Increment for the array **a**, **inca** ≠ 0:

inca > 0 — a is stored forward in array **a**, i.e., a_i is stored in **a**((i−1)×**inca**+1).
inca < 0 — a is stored backward in array **a**, i.e., a_i is stored in **a**((i−**n**)×**inca**+1).

Use **inca** = 1 if the vector a is stored contiguously in array **a**, i.e., if a_i is stored in **a**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

b Array of length **lenb** = (**n**−1)×|**incb**|+1 containing the n-vector b.

incb Increment for the array **b**, **incb** ≠ 0:

incb > 0 — b is stored forward in array **b**, i.e., b_i is stored in **b**((i−1)×**incb**+1).
incb < 0 — b is stored backward in array **b**, i.e., b_i is stored in **b**((i−**n**)×**incb**+1).

Use **incb** = 1 if the vector b is stored contiguously in array **b**, i.e., if b_i is stored in **b**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

x Array of length **lenx** = (**n**−1)×|**incx**|+1 containing the first two elements, x_1 and x_2 of the n-vector x.

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

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

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

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

Output

x If **n** = 0, then **x** is unchanged. Otherwise, the recurrence's solution vector replaces the input.

Notes

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

If an error in the arguments is detected, the subprograms call error handler 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

$$x_i = a_{i-2}x_{i-1} + b_{i-2}x_{i-2}, \quad i = 3, 4, \ldots, n,$$

where a, b, and x are vectors 10 elements long stored in one-dimensional arrays A, B, and X of dimension 20.

```
      INTEGER*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)
```

Purpose

Given real vectors a, b, and x of length n, these subprograms solves the second order linear recurrence

$$y_1 = x_1$$
$$y_2 = x_2$$
$$y_i = x_i + a_{i-2}y_{i-1} + b_{i-2}y_{i-2}, \quad i = 3, 4, \ldots, n$$

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** ≥ 0. If **n** = 0, the subprogram does not reference **a**, **b**, or **x**.

a Array of length **lena** = (**n**−1)×|**inca**|+1 containing the n-vector a.

inca Increment for the array **a**, **inca** ≠ 0:

inca > 0 — a is stored forward in array **a**, i.e., a_i is stored in **a**((i−1)×**inca**+1).

inca < 0 — a is stored backward in array **a**, i.e., a_i is stored in **a**((i−**n**)×**inca**+1).

Use **inca** = 1 if the vector a is stored contiguously in array **a**, i.e., if a_i is stored in **a**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

b Array of length **lenb** = (**n**−1)×|**incb**|+1 containing the n-vector b.

incb Increment for the array **b**, **incb** ≠ 0:

incb > 0 — b is stored forward in array **b**, i.e., b_i is stored in **b**((i−1)×**incb**+1).

incb < 0 — b is stored backward in array **b**, i.e., b_i is stored in **b**((i−**n**)×**incb**+1).

Use **incb** = 1 if the vector b is stored contiguously in array **b**, i.e., if b_i is stored in **b**(i). Refer to "BLAS Indexing Conventions" in the introduction to Chapter 2.

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

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

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

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

Output **x** If **n** = 0, then **x** is unchanged. Otherwise, the recurrence's solution vector replaces the input.

Notes The result is unspecified if **a**, **b**, or **x** overlap such that any element of a, b, or x shares a memory location with any other element of a, b, or 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,
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

$$\begin{aligned} y_1 &= x_1 \\ y_2 &= x_2 \\ y_i &= x_i + a_{i-2}y_{i-1} + b_{i-2}y_{i-2}, \quad i = 3, 4, \ldots, n. \end{aligned}$$

where a, b, and x are vectors 10 elements long stored in one-dimensional arrays A, B, and X of dimension 20, and y overwrites x.

```
      INTEGER*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)
```

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:

- allocate volatile dynamic memory
- allocate nonvolatile dynamic memory
- generate random numbers
- convert between CONVEX and IBM floating-point formats
- sort the elements of a vector into ascending or descending order
- report errors detected in the usage of VECLIB subprograms

Chapter Objective

After reading this chapter you will know how to use the described subprograms.

What You Need to Know to Use These Subprograms

Subprograms described in this chapter do not normally perform vector operations. One exception is related to the VAX VMS-compatible random-number generator, RAN. This generator produces exactly the same sequence of random numbers as on the VAX. RANV supplements RAN; it is a vectorized generator that fills a vector with the same sequence of random numbers as RAN would produce in a loop, but at a much faster rate. SRANV and DRANV are other vectorized random number generators that produce a different sequence of random numbers.

Dynamic Memory Subprograms

This chapter describes several dynamic memory subprograms:

- DALLOC
- DYNAMIC
- MALLOC
- NALLOC
- RALLOC

NALLOC, DALLOC, and RALLOC are a family of VECLIB dynamic memory allocators for FORTRAN 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.

DYNAMIC is a dynamic memory allocator for use in FORTRAN subprograms. The subprogram MALLOC also provides a dynamic storage allocation facility. These subprograms allocate space that is automatically deallocated when program control returns to the calling procedure. The difference between DYNAMIC and MALLOC is that DYNAMIC provides arrays that may be used within the subprogram that calls it, while MALLOC provides arrays that are most conveniently used by passing them to a lower-level subprogram.

Supplemental Reading

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.

Subprogram Descriptions

Deallocate Nonvolatile Dynamic Memory
DALLOC 12-3

Allocate Dynamic Memory
DYNAMIC 12-4

Allocate Dynamic Memory
MALLOC 12-8

Allocate Nonvolatile Dynamic Memory
NALLOC 12-11

Reallocate Nonvolatile Dynamic Memory
RALLOC 12-13

Generate VAX-Compatible Random Numbers
RAN 12-15

Generate VAX-Compatible Random Numbers
RANV 12-17

Convert CONVEX Floating-point Numbers to IBM Floating-Point Format
SC2IBM 12-19

Convert IBM Floating-point Numbers to CONVEX Floating-Point Format
SIBM2C 12-21

Scalar Long Period Random Number Generator
SRAN, DRAN 12-23

Vector Long Period Random Number Generator
SRANV, DRANV 12-25

Sort an Array into Ascending or Descending Order
SSORT, DSORT, ISORT 12-27

VECLIB Error Handler
XERVEC 12-29

Deallocate Nonvolatile Dynamic Memory — DALLOC

Purpose

DALLOC is a dynamic memory deallocator for FORTRAN programs. It uses the ConvexOS 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
   handle error
END IF
```

VECLIB8:

```
INTEGER*4 iptr
INTEGER*8 ier
CALL DALLOC (iptr, ier)
IF ( ier .LT. 0 ) THEN
   handle error
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 ConvexOS general-purpose memory allocation package; refer to *malloc*(3) for details and error conditions.

Example

Deallocate the arrays allocated in "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
```

Purpose

DYNAMIC is a dynamic memory allocator for use in FORTRAN subprograms. It uses the CONVEX subroutine-call/argument-passing mechanism and the runtime stack to provide a facility that is not included in standard FORTRAN 77. VECLIB subprogram MALLOC also provides a dynamic storage allocation facility. The fundamental difference between DYNAMIC and MALLOC is that DYNAMIC provides arrays that may be used within the subprogram that calls it, while MALLOC provides arrays that are most conveniently used by passing them to a lower-level subprogram.

A brief explanation of the subroutine-call/argument-passing mechanism will help you understand how to use DYNAMIC. When a program unit calls a subprogram, it passes the address of the ***argument packet*** to the called subprogram. The argument packet is a list of memory addresses of the actual arguments, consisting of one 32-bit word for each actual argument, an additional word if the subprogram is a complex-valued function subprogram, an additional word for each character argument, and two additional words if the subprogram is a character-valued function. When the subprogram needs to access its arguments, it gets their addresses from the argument packet.

A subprogram is normally called with the same number of actual arguments as dummy arguments. However, when DYNAMIC is being used, the subprogram is called with fewer actual arguments than dummy arguments. The extra dummy arguments correspond to arrays that are to be allocated dynamically by calling DYNAMIC inside the subprogram. When DYNAMIC is called, it allocates space on the runtime stack for the requested arrays and additional space for a new argument packet. It also changes the argument packet pointer of the subprogram that called DYNAMIC to point to the new argument packet. The new argument packet contains addresses of original arguments as well as addresses of dynamically allocated arrays. After DYNAMIC has been called, it appears to the calling subprogram that the subprogram was called with more actual arguments than appeared in the **CALL** statement or function reference, and that the additional arguments are arrays of the sizes specified in the **CALL DYNAMIC** statement.

When a **RETURN** statement is executed in the subprogram that called DYNAMIC, the dynamically allocated space is automatically deallocated. Thus, space allocated during one execution of a subprogram must be reallocated for a subsequent execution. The same memory will not necessarily be allocated by two such executions, and there is no certainty that contents of that memory will be preserved even if the same memory is allocated.

DYNAMIC is included in VECLIB because extra arrays are occasionally required to replace nonvectorized code with vectorized code or a VECLIB subprogram. It is unnecessary when using DYNAMIC to build fixed-size temporary arrays into a subprogram 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

DYNAMIC is called with a variable number of arguments, which must be in pairs, corresponding to the arrays to be dynamically allocated.

VECLIB:

CALL sub (<*nondynamic actual arguments*>)

.

.

SUBROUTINE sub (<*nondynamic dummy args*>,<*dynamic dummy args*>)
INTEGER*4 ier, DYNAMIC, n1, l1, n2, l2, ..., nk, lk
array declarations for <*dynamic dummy args*>
ier = DYNAMIC (n1, l1, n2, l2, ..., nk, lk)
IF (ier .EQ. 0) THEN
handle stack overflow
ENDIF

VECLIB8:

CALL sub (<*nondynamic actual arguments*>)

.

.

SUBROUTINE sub (<*nondynamic dummy args*>,<*dynamic dummy args* >)
INTEGER*8 ier, DYNAMIC, n1, l1, n2, l2, ..., nk, lk
array declarations for <*dynamic dummy args*>
ier = DYNAMIC (n1, l1, n2, l2, ..., nk, lk)
IF (ier .EQ. 0) THEN
handle stack overflow
ENDIF

Input

n1, n2, ..., nk The argument numbers of the first, second, ..., **k**-th dummy arguments (in the **FUNCTION** or **SUBROUTINE** statement of the subprogram calling DYNAMIC) that are to have space dynamically allocated to them. If the calling subprogram is a **SUBROUTINE** subprogram or an integer-, logical-, or real-valued **FUNCTION** subprogram, the dummy arguments are numbered beginning with 1. If the calling subprogram is a complex-valued **FUNCTION**, the dummy arguments are numbered beginning with 2. For example, in the following subprogram headers, B is argument number 2, C is argument number 3, and D is argument number 4.

```
SUBROUTINE          FOO (A, B, C, D)
INTEGER*4 FUNCTION  FOO (A, B, C, D)
LOGICAL*4 FUNCTION  FOO (A, B, C, D)
REAL*4 FUNCTION     FOO (A, B, C, D)
COMPLEX*8 FUNCTION  FOO (   B, C, D)
```

Thus, if D is to be allocated dynamically, a value **n** = 4 is used.

The subprogram that calls DYNAMIC should include an ordinary array declaration for each dynamically allocated array. Refer to "Examples."

l1, l2, ..., lk The length, in bytes, of the first, second, ..., **k**-th array to allocate on the runtime stack. **l** should be the product of the byte length for the data type and the range of elements in each dimension. For a REAL*4 array of dimension 5 by 6, use $l = 4 \times 5 \times 6 = 120$. For any $l \le 0$, $l = 8$ is assumed.

Output

ier Status response. **ier** = 0 if there is insufficient space on the runtime stack to allocate the requested space. You must test **ier**. Refer to "Notes."

Notes

You will obtain unsatisfactory results if the dynamic arrays are referenced before they have been allocated. The FORTRAN compiler may reorder statements in a basic block if, according to ANSI standards, dependencies are not allowed. There is, however, a dependency between a call to DYNAMIC and use of a dynamically allocated array. Because this dependency is hidden from the compiler, the call to DYNAMIC and all references to dynamically allocated arrays must occur in different basic blocks. Refer to the example call in "Usage." In this example, the **IF** statement knows the end of the basic block containing the **CALL DYNAMIC** statement.

DYNAMIC does not work if the subprogram that calls it has any character dummy arguments. This is because of the extra compiler-generated actual arguments with special values that correspond to character dummy arguments.

DYNAMIC allocates each array so it is aligned on a longword boundary.

On the first call to DYNAMIC, the stack size limit of the process is set to the maximum value permitted the process. The new limit stays in effect for the duration of the process.

Example 1

Dynamically allocate an M-by-N REAL*8 array WORK and an INTEGER*4 array INDX of length N+1 in SUBROUTINE FOO. WORK and INDX are the 5th and 6th arguments, respectively. (Compare with "Example 1" in the description of MALLOC.)

```
CALL FOO (M,N,X,Y)
    .
    .
    .
SUBROUTINE FOO (M,N,X,Y,WORK,INDX)
INTEGER*4  M,N,INDX(*),IER,DYNAMIC
REAL*8     WORK(M,N)
IER = DYNAMIC (5,8*M*N,6,4*N+4)
IF ( IER .EQ. 0 ) THEN
  handle stack overflow
END IF
```

After DYNAMIC returns with a nonzero function value, subprogram execution in FOO continues just as if FOO had been called with six actual arguments, the last two being a REAL*8 array of dimension M by N and an INTEGER*4 array of dimension N+1.

Example 2

If FOO in "Example 1" is an integer-, logical-, or real-valued function subprogram with the same dummy argument list instead of a SUBROUTINE subprogram, the same function reference to DYNAMIC will work. If FOO is a complex-valued function subprogram with the same dummy argument list, the function reference to DYNAMIC should be

```
IER = DYNAMIC (6,8*M*N,7,4*N+4)
```

The values of **n1** and **n2** have been increased by one because the subprogram is a complex-valued function subprogram.

Example 3 Replace a slow linear equation solver by a VECLIB version that requires some workspace. Do this by writing a SUBROUTINE subprogram that has the same argument list as the slow solver. This subprogram allocates space by using DYNAMIC and then calls DGEFA and DGESL, which use the dynamically allocated memory just as if it had been statically allocated with a **DIMENSION** statement. (Compare with "Example 2" in the description of MALLOC.)

```
      SUBROUTINE LINSOL (N,A,LDA,B,IPVT)
C
C     SOLVE THE N BY N LINEAR SYSTEM  A*X = B
C
      INTEGER*4 N,LDA,IPVT(N),IER,DYNAMIC
      REAL*8 A(LDA,N),B(N)
C
      IER = DYNAMIC (5,4*N)
      IF ( IER .EQ. 0 ) THEN
         WRITE (6,*) 'OUT OF MEMORY IN LINSOL'
         STOP
      END IF
      CALL DGEFA (A,LDA,N,IPVT,IER)
      IF ( IER .EQ. 0 ) THEN
         WRITE (6,*) 'SINGULAR MATRIX IN LINSOL'
         STOP
      END IF
      CALL DGESL (A,LDA,N,IPVT,B,0)
      RETURN
      END
```

MALLOC — Allocate Dynamic Memory

Purpose

MALLOC is a dynamic memory allocator for FORTRAN programs. It uses the CONVEX runtime stack to provide a facility that is not included in standard FORTRAN 77. VECLIB subprogram DYNAMIC also provides a dynamic storage allocation facility. The difference between MALLOC and DYNAMIC is that MALLOC provides arrays that are most conveniently used by passing them to a lower-level subprogram, while DYNAMIC provides arrays that may be used within the subprogram that calls it.

When MALLOC is called, it changes the hardware stack pointer to reserve a specified amount of space on the runtime stack. It then returns the memory address of this space. If FORTRAN 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 CONVEX FORTRAN compiler to pass the value of its argument to a subroutine rather than to the address; this turns the pointer in the calling subprogram into an array in the lower-level subprogram.

When a **RETURN** statement is executed in the program unit that called MALLOC, the dynamically allocated space is automatically deallocated. Space allocated during one execution of a subprogram must be reallocated for a subsequent execution. The same memory will not necessarily be allocated by two such executions, and there is no certainty that contents of that memory will be preserved, even if the same memory is allocated.

MALLOC is included in VECLIB because extra arrays are occasionally required to replace nonvectorized code with vectorized code or a VECLIB subprogram. It is unnecessary when using MALLOC 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:

```
INTEGER*4 l, iptr, ier
CALL MALLOC (l, iptr, ier)
IF ( ier .LT. 0 ) THEN
   handle stack overflow
ELSE
   CALL sub (..., %VAL(iptr), ...)
END IF
```

VECLIB8:

```
INTEGER*4 iptr
INTEGER*8 l, ier
CALL MALLOC (l, iptr, ier)
IF ( ier .LT. 0 ) THEN
   handle stack overflow
ELSE
   CALL sub (..., %VAL(iptr), ...)
END IF
```

Input

l The length, in bytes, of the array to allocate on the runtime stack. l should be the product of byte length for the data type and range of elements in each dimension. Thus for a REAL*4 array of dimension 5 by 6, use $l = 4 \times 5 \times 6 = 120$. If $l \leq 0$, $l = 8$ is assumed.

Output

iptr The memory address of the allocated array if **ier** = 0 is returned.

ier Status response:

ier = 0 Normal return.
ier = –1 Insufficient runtime stack space.

Notes

MALLOC 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.

On the first call to MALLOC, the process' stack size limit is set to the maximum value permitted the process. The new limit stays in effect for the duration of the process.

MALLOC allocates space from the current stack pointer each time it is called, advancing the stack pointer past the allocated space. Thus, allocations are cumulative. Calling MALLOC in a loop may use an inordinate amount of space on the runtime stack if the calling subprogram does not execute a RETURN statement between calls to deallocate the previously allocated space.

The usage given above differs from that documented in previous editions of the *CONVEX VECLIB User's Guide.* The original usage still is supported, and is described below, but should be considered obsolete.

VECLIB:

```
INTEGER*4 MALLOC, iptr, l
iptr = MALLOC (l)
IF ( iptr .EQ. 0 ) THEN
   handle stack overflow
ELSE
   CALL sub (..., %VAL(iptr), ...)
END IF
```

VECLIB8:

```
INTEGER*4 MALLOC, iptr
INTEGER*8 l
iptr = MALLOC (l)
IF ( iptr .EQ. 0 ) THEN
   handle stack overflow
ELSE
   CALL sub (..., %VAL(iptr), ...)
END IF
```

In the above, MALLOC returns **iptr** = 0 if there is insufficient space on the runtime stack to allocate the requested space.

Example 1 Allocate an m-by-n REAL*8 array and an INTEGER*4 array of length $n+1$ and pass them as arguments to subroutines SUB1 and SUB2. (Compare with "Example 1" in the description of DYNAMIC and with "Example" in the description of NALLOC.)

```
INTEGER*4 M,N,IPTR,JPTR,IER1,IER2
CALL MALLOC (8*M*N,IPTR,IER1)
CALL MALLOC (4*N+4,JPTR,IER2)
IF ( IER1 .LT. 0 .OR. IER2 .LT. 0 ) THEN
   WRITE (6,*) 'MALLOC ERROR:',M,N,IER1,IER2
   STOP
END IF
CALL SUB1 (..., %VAL(IPTR), ..., %VAL(JPTR), ...)
CALL SUB2 (..., %VAL(IPTR), ..., %VAL(JPTR), ...)
```

Example 2 Replace a slow linear equation solver with a VECLIB version that requires some workspace. Do this by writing a SUBROUTINE subprogram that has the same argument list as the slow solver. This subprogram allocates space via MALLOC, then calls DGEFA and DGESL, which use the dynamically allocated memory just as if it had been statically allocated with a **DIMENSION** statement. (Compare with "Example 3" in the description of DYNAMIC.)

```
      CALL LINSOL (N,A,LDA,B)
        .
        .
        .
      SUBROUTINE LINSOL (N,A,LDA,B)
C
C     SOLVE THE N BY N LINEAR SYSTEM  A*X = B
C
      INTEGER*4 N,LDA,IPVT,IER
      REAL*8 A(LDA,N),B(N)
C
      CALL MALLOC (4*N,IPVT,IER)
      IF ( IER .LT. 0 ) THEN
         WRITE (6,*) 'OUT OF MEMORY IN LINSOL'
         STOP
      END IF
      CALL DGEFA (A,LDA,N,%VAL(IPVT),IER)
      IF ( IER .EQ. 0 ) THEN
         WRITE (6,*) 'SINGULAR MATRIX IN LINSOL'
         STOP
      END IF
      CALL DGESL (A,LDA,N,%VAL(IPVT),B,0)
      RETURN
      END
```

Allocate Nonvolatile Dynamic Memory — NALLOC

Purpose

NALLOC is a dynamic memory allocator for FORTRAN programs. It uses the ConvexOS general-purpose memory allocation package to provide a facility that is not included in standard FORTRAN 77. VECLIB subprograms DYNAMIC and MALLOC also provide dynamic storage allocation. The difference between NALLOC and those is that NALLOC provides arrays that exist for the duration of a process unless explicitly deallocated. In contrast, arrays allocated via DYNAMIC or MALLOC are deallocated automatically when the calling subprogram executes a **RETURN** statement.

When NALLOC is called, it obtains a block of memory of the specified size and returns the memory address of that space. If FORTRAN 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 CONVEX 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 replace nonvectorized code with vectorized code or 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:

```
INTEGER*4 l, iptr, ier
CALL NALLOC (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 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:

ier = 0 Normal return.
ier = –1 Illegal array size; l ≤ 0.
ier = –2 Too many nonvolatile arrays allocated.
ier = –3 Error response from ConvexOS function _malloc.

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 ConvexOS general-purpose memory allocation package; refer to *malloc*(3) 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 1" in the description of DYNAMIC, with "Example 1" in the description of MALLOC, and 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),...)
```

Reallocate Nonvolatile Dynamic Memory — RALLOC

Purpose

RALLOC is a dynamic memory reallocator for FORTRAN programs. It uses the ConvexOS general-purpose memory allocation package to provide a facility that is not included in standard FORTRAN 77. 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.

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.

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; $l \le 0$.
ier = -2	**iptr** is not a valid array address.
ier = -3	Error response from ConvexOS function _realloc.

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 ConvexOS general-purpose memory allocation package; refer to *malloc*(3) 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. (Compare with "Example 1" in the description of DYNAMIC and with "Example 1" in the description of MALLOC.)

```
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),...)
```

VAX-Compatible Random Numbers — RAN

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, which makes it easier to validate conversion of Monte Carlo programs from a VAX to a CONVEX supercomputer. Function RAN returns one random number per call, while companion subroutine RANV, also documented in this chapter, is a vectorized version of RAN that returns an array of random numbers per call at much less execution time per number.

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 $0 \le S_0 < 2^{32}$, they obtain a sequence of seeds $\{ S_n \}$ by setting

$$S_n = (69069 S_{n-1} + 1) \bmod 2^{32}, \quad n > 0.$$

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.

RAN **Continued**

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.

VAX-Compatible Random Numbers RANV

Purpose

This subprogram produces uniform [0,1) pseudorandom numbers identical to those generated by the VAX random-number generator, RAN. CONVEX VECLIB function RAN, documented elsewhere in this chapter, is identical in usage to its VAX counterpart, which makes it easier to validate conversion of Monte Carlo programs from a VAX to a CONVEX supercomputer. Subroutine RANV is a vectorized version of RAN that returns many random numbers per call at much less execution time per number.

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 $0 \le S_0 < 2^{32}$, they obtain a sequence of seeds $\{ S_n \}$ by setting

$$S_n = (69069 S_{n-1} + 1) \bmod 2^{32}, \quad n > 0.$$

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.

CONVEX to IBM Floating-Point Conversion — SC2IBM

Purpose

This subprogram converts a vector x of REAL*4 numbers stored in an array in CONVEX native or IEEE floating-point format into the equivalent REAL*4 numbers y stored in IBM floating-point format.

Native-mode CONVEX REAL*4 floating-point numbers are stored in 32 bits, consisting of 1 sign bit, 8 exponent bits, and 23 mantissa bits. Binary (power of 2) normalization is used. The mantissa is logically 24 bits in length, but because CONVEX floating-point numbers are always normalized, the high-order bit of the mantissa is known to have the value 1 and, therefore, is not stored. The binary point is to the left of this *implicit* or ***hidden*** bit. The quantity in the 8-bit exponent field is 128 greater than the exponent of the power-of-2 scale factor. Thus, if s represents the sign bit, $e...e$ represents the exponent bits, and $m...m$ represents the mantissa bits, the value of a native-mode CONVEX REAL*4 floating-point number is

$$v = \begin{cases} +\,2^{eeeeeeee-128} \times .1mmmmmmmmmmmmmmmmmmmmmmm & \text{if } s = 0 \\ -\,2^{eeeeeeee-128} \times .1mmmmmmmmmmmmmmmmmmmmmmm & \text{if } s = 1 \end{cases}$$

IEEE-mode CONVEX REAL*4 floating-point numbers are stored in 32 bits, consisting of 1 sign bit, 8 exponent bits, and 23 mantissa bits. Binary (power of 2) normalization is used. The mantissa is logically 24 bits in length, but because IEEE floating-point numbers are always normalized, the high-order bit of the mantissa is known to have the value 1 and, therefore, is not stored. The binary point is to the right of this *implicit* or *hidden* bit. The quantity in the 8-bit exponent field is 127 greater than the exponent of the power-of-2 scale factor. Thus, if s represents the sign bit, $e...e$ represents the exponent bits, and $m...m$ represents the mantissa bits, the value of a native-mode CONVEX REAL*4 floating-point number is

$$v = \begin{cases} +\,2^{eeeeeeee-127} \times 1.mmmmmmmmmmmmmmmmmmmmmmm & \text{if } s = 0 \\ -\,2^{eeeeeeee-127} \times 1.mmmmmmmmmmmmmmmmmmmmmmm & \text{if } s = 1 \end{cases}$$

IBM REAL*4 floating-point numbers are stored in 32 bits, consisting of 1 sign bit, 7 exponent bits, and 24 mantissa bits. Hexadecimal (power of 16) normalization is used. A normalized floating-point number may have zero to three high-order zero bits in the mantissa. The hexadecimal point is to the left of the high-order mantissa bit. The quantity in the 7-bit exponent field is 64 greater than the exponent of the power-of-16 scale factor. Thus, if s represents the sign bit, $e...e$ represents the exponent bits, and $m...m$ represents the mantissa bits, the value of an IBM REAL*4 floating-point number is

$$v = \begin{cases} +\,16^{eeeeeee-64} \times .mmmmmmmmmmmmmmmmmmmmmmmm & \text{if } s = 0 \\ -\,16^{eeeeeee-64} \times .mmmmmmmmmmmmmmmmmmmmmmmm & \text{if } s = 1 \end{cases}$$

A CONVEX REAL*4 floating-point number always has 24 bits of precision (the 23 mantissa bits plus the hidden bit), while an IBM REAL*4 floating-point number may have 21 to 24 bits of precision, depending on the number of high-order zero bits in its mantissa. When an IBM number has fewer than 24 bits of precision, the CONVEX number is rounded to the precision of the IBM number.

Usage

VECLIB:

```
INTEGER*4 n, incx, incy
REAL*4    x(lenx), y(leny)
CALL SC2IBM (n, 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**.

x Array of length **lenx** = (**n**–1)×|**incx**|+1 containing n floating-point numbers in CONVEX floating-point format.

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

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

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

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

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

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

Output

y Array of length **leny** = (**n**–1)×|**incy**|+1 containing n floating-point numbers in IBM floating-point format. If **n** $\leq$ 0, then **y** is unchanged.

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.

Example

Convert the REAL*4 vector x into y where x and y are vectors 10 elements long stored in CONVEX floating-point format in one-dimensional array X of dimension 20 into IBM floating-point format and store the result in one-dimensional array Y of dimension 20.

```
INTEGER*4 N,INCX,INCY
REAL*4    X(20),Y(20)
N = 10
INCX = 1
INCY = 1
CALL SC2IBM (N,X,INCX,Y,INCY)
```

IBM to CONVEX Floating-Point Conversion SIBM2C

Purpose

This subprogram converts a vector x of REAL*4 numbers stored in an array in IBM floating-point format into the equivalent REAL*4 numbers y stored in CONVEX native or IEEE floating-point format.

IBM REAL*4 floating-point numbers are stored in 32 bits, consisting of 1 sign bit, 7 exponent bits, and 24 mantissa bits. Hexadecimal (power of 16) normalization is used. A normalized floating-point number may have zero to three high-order zero bits in the mantissa. The hexadecimal point is to the left of the high-order mantissa bit. The quantity in the 7-bit exponent field is 64 greater than the exponent of the power-of-16 scale factor. Thus, if s represents the sign bit, $e...e$ represents the exponent bits, and $m...m$ represents the mantissa bits, the value of an IBM REAL*4 floating-point number is

$$v = \begin{cases} +16^{eeeeeee-64} \times .mmmmmmmmmmmmmmmmmmmmmmmm & \text{if } s = 0 \\ -16^{eeeeeee-64} \times .mmmmmmmmmmmmmmmmmmmmmmmm & \text{if } s = 1 \end{cases}$$

Native-mode CONVEX REAL*4 floating-point numbers are stored in 32 bits, consisting of 1 sign bit, 8 exponent bits, and 23 mantissa bits. Binary (power of 2) normalization is used. The mantissa is logically 24 bits in length, but because CONVEX floating-point numbers are always normalized, the high-order bit of the mantissa is known to have the value 1 and, therefore, is not stored. The binary point is to the left of this *implicit* or *hidden* bit. The quantity in the 8-bit exponent field is 128 greater than the exponent of the power-of-2 scale factor. Thus, if s represents the sign bit, $e...e$ represents the exponent bits, and $m...m$ represents the mantissa bits, the value of a native-mode CONVEX REAL*4 floating-point number is

$$v = \begin{cases} +2^{eeeeeeee-128} \times .1mmmmmmmmmmmmmmmmmmmmmmm & \text{if } s = 0 \\ -2^{eeeeeeee-128} \times .1mmmmmmmmmmmmmmmmmmmmmmm & \text{if } s = 1 \end{cases}$$

IEEE-mode CONVEX REAL*4 floating-point numbers are stored in 32 bits, consisting of 1 sign bit, 8 exponent bits, and 23 mantissa bits. Binary (power of 2) normalization is used. The mantissa is logically 24 bits in length, but because IEEE floating-point numbers are always normalized, the high-order bit of the mantissa is known to have the value 1 and, therefore, is not stored. The binary point is to the right of this *implicit* or *hidden* bit. The quantity in the 8-bit exponent field is 127 greater than the exponent of the power-of-2 scale factor. Thus, if s represents the sign bit, $e...e$ represents the exponent bits, and $m...m$ represents the mantissa bits, the value of a native-mode CONVEX REAL*4 floating-point number is

$$v = \begin{cases} +2^{eeeeeeee-127} \times 1.mmmmmmmmmmmmmmmmmmmmmmm & \text{if } s = 0 \\ -2^{eeeeeeee-127} \times 1.mmmmmmmmmmmmmmmmmmmmmmm & \text{if } s = 1 \end{cases}$$

A CONVEX REAL*4 floating-point number always has 24 bits of precision (the 23 mantissa bits plus the hidden bit), while an IBM REAL*4 floating-point number may have 21 to 24 bits of precision, depending on the number of high-order zero bits in its mantissa. Hence, no precision can be lost when converting from IBM to CONVEX format. However, the dynamic range of a CONVEX REAL*4 floating-point number, approximately 10^{-38} to 10^{+38}, is smaller than the dynamic range of an IBM REAL*4 floating-point number, which is approximately 10^{-76} to 10^{+76}. IBM numbers below the lower end of the CONVEX range will underflow to zero without warning, but IBM numbers near or above the upper end of the CONVEX range may cause an overflow exception.

Usage

VECLIB:

```
INTEGER*4 n, incx, incy
REAL*4    x(lenx), y(leny)
CALL SIBM2C (n, 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**.

x Array of length **lenx** = (**n**–1)×|**incx**|+1 containing n floating-point numbers in IBM floating-point format.

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

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

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

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

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

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

Output

y Array of length **leny** = (**n**–1)×|**incy**|+1 containing n floating-point numbers in CONVEX floating-point format. If **n** $\leq$ 0, then **y** is unchanged.

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.

Example

Convert the vector x of IBM REAL*4 floating point numbers into the equivalent numbers in CONVEX REAL*4 floating-point format and store the result in the vector y, where x and y are vectors 10 elements long stored in one-dimensional array X Y of dimension 20, respectively.

```
INTEGER*4 N,INCX,INCY
REAL*4    X(20),Y(20)
N = 10
INCX = 1
INCY = 1
CALL SIBM2C (N,X,INCX,Y,INCY)
```

Scalar Long Period Random Number Generator — SRAN/DRAN

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, are vectorized versions of SRAN and DRAN that return an array of random numbers per call at much less execution time per number.

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 $0 \le S_0 < 2^{48}$, they obtain a sequence of seeds $\{ S_n \}$ by setting

$$S_n = (31167285 S_{n-1} + 1) \bmod 2^{48}, \quad n > 0.$$

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*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.

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.

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 $0 \le S_0 < 2^{48}$, they obtain a sequence of seeds $\{ S_n \}$ by setting

$$S_n = (31167285 S_{n-1} + 1) \bmod 2^{48}, \quad n > 0.$$

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)
```

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 — 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

iseed The seed that produces the next pseudorandom number in the sequence replaces the input seed.

x Array of length **lenx** = (**n**–1)×|**incx**|+1 containing the n-vector X. If **n** ≤ 0, then **x** is not referenced. Otherwise, the next **n** pseudorandom numbers in the sequence replaces the input.

Notes

CALL SRANV (iseed, n, x, incx) produces the same **n** pseudorandom numbers as returned by **n** successive references to **SRAN(iseed)**.

CALL SRANV (iseed, n, x, incx) produces the same value of **iseed** that would have resulted from the last of **n** successive references to **SRAN(iseed)**.

Starting a sequence twice with the same value of **iseed** will produce the same pseudorandom sequence.

You may have as many independent sequences going at a time as you desire by having a different **iseed** for each one.

The subprograms treat **iseed** as an unsigned 48-bit quantity. To write it out for later continuation of the same sequence, either use unformatted I/O statements or write it out with an octal, unsigned integer, or hexadecimal format descriptor of the form **O16**, **SU,I15**, or **Z12**, respectively.

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.

Purpose These subprograms use a vectorized 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** ≤ 1, the subprograms do not reference **x**.

x Array of length **lenx** = (**n**–1)×|**incx**|+1 containing the data to be sorted.

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

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

VECLIB Error Handler — XERVEC

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 *
**************************************************************************
```

or

```
**************************************************************************
* XERVEC: error detected by subprogram name: text of error message        *
**************************************************************************
```

or

```
**************************************************************************
* 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 perhaps 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 given above and *iarg* is not part of the error message. If *iarg* < 0, the error message is printed in the third form given above and *iarg* is the error number.

messag The text of the error message to be printed if **iarg** ≤ 0. Not used as input if **iarg** > 0.

A

Calling CONVEX VECLIB from C

Introduction

Subprogram calls in CONVEX FORTRAN use the same calling convention as used by other CONVEX language processors. This CONVEX standard permits CONVEX 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 CONVEX VECLIB subprograms from C programs. The same conventions apply to calling CONVEX SCILIB and CONVEX LAPACK subprograms from C programs. For non-VECLIB-specific information on this topic, see the *CONVEX Interlanguage Programming Guide.*

General Interlanguage Programming Rules for VECLIB

The following rules for calling a CONVEX VECLIB subprogram from within a C language program have been adapted from and follow the conventions in the *CONVEX Interlanguage Programming Guide.*

- 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 */
```

- According to the CONVEX language processor calling conventions, a FORTRAN complex function subprogram does not return a function value directly. Instead, a complex function is passed an extra argument at the beginning of the argument list. This argument is a pointer to a `COMPLEX*8` or `COMPLEX*16` variable into which the subprogram stores the function return value. Refer to Table A-1 and section "Calling Functions" for examples.

- Using the above `typedef` statements, FORTRAN and C declarations are related as shown in Table A-1.

- If it is necessary to load library I77 to satisfy unresolved symbols at link time (see "Linking VECLIB Subprograms into a C Program") you must initialize the FORTRAN runtime I/O system. Include the call

```
f_init ();
```

in the initialization portion of the program. Refer to the *CONVEX Interlanguage Programming Guide* for more information about mixing FORTRAN and C I/O.

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 (...)	void cf_ (fn_val, ...)
	complex8_t *fn_val; /* return value through fn_val */
COMPLEX*16 FUNCTION zf (...)	void zf_ (fn_val, ...)
	complex16_t *fn_val; /* return value through fn_val */

Header Files

C header files for VECLIB and VECLIB8 are installed in */usr/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 **-cfc**, **-pd**, and **-pd8** switches of the CONVEX FORTRAN compiler since these switches are not available with the CONVEX 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 discussion following 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. The only exceptions are that function prototypes and declarations for subprograms DYNAMIC and MALLOC are omitted from *veclib.h* and *veclib8.h* because including MALLOC would interfere with the C function `malloc` and DYNAMIC is dependent on the call linkage generated by the CONVEX FORTRAN compiler. Besides, the functionality of both MALLOC and DYNAMIC are readily available in the C language.

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 to the order of the corresponding –l options on your link line. See "Interactions Between VECLIB, SCILIB, and LAPACK" in Chapter 1 for details about how to order the –l 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.

```
#infdef 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.

- Because of the special method of returning the function value from COMPLEX*8 and COMPLEX*16 function subprograms, they are declared of type `void`. A special variable, `fn_val`, of type `complex8_t` or `complex16_t` as appropriate, is inserted at the beginning of the argument list for the function value. For example, the function prototype for ZDOTC is

```
extern void zdotc_ (complex16_t* fn_val, int* n, complex16_t x[],
        int* incx, complex16_t y[], int* incy);
```

- The following is the function prototype for the FORTRAN I/O initialization subprogram, `f_init`:

```
extern void f_init ();
```

- If you are compiling with the CONVEX C **–pcc** command line option, the ANSI C prototypes are replaced by Kernighan-and-Ritchie-style declarations. For example, the three prototypes above are replaced with

```
extern void dgemm_ ();

extern void zdotc_ ();
```

and

```
extern void f_init ();
```

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} };

    zdotc_ (&s, &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 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.

- Several VECLIB subprograms contain FORTRAN I/O statements, primarily for error reporting, or make other reference to the FORTRAN runtime libraries. The FORTRAN libraries (U77, F77, I77, D77 and either mathC1 or mathC2), normally linked automatically when *fc* is used, are not linked when *cc* is used. If the link command line given in the preceding bullet fails because of unresolved symbols, see Table A-2 for additional library files to specify on the link command line.

- The following is the longest command required to compile a program with *cc* and link with VECLIB for execution of a C1 processor:

  cc –o *test test.***c –lveclib –lU77 –lF77 –lI77 –lD77 –lmathC1**

Table A-2: Libraries to Satisfy Unresolved Symbols at Link Time

Unresolved Symbol	Occurs With	Documentation Reference	Satisfy With –l				
			U77	F77	I77	D77	mathC?†
etime	-lveclib	VECLIB ch. 6,7	√				
loc	-lveclib	VECLIB ch. 8	√				
_for$do_fio	-lveclib	VECLIB ch. 6,7,8			√	√	√
_for$do_uio	-lveclib	VECLIB ch. 6,7,8			√	√	√
_for$e_rsue	-lveclib	VECLIB ch. 6,7,8			√	√	√
_for$e_wsfe	-lveclib	VECLIB ch. 6,7,8			√	√	√
_for$e_wsfi	-lveclib	VECLIB ch. 6,7,8			√	√	√
_for$e_wsue	-lveclib	VECLIB ch. 6,7,8			√	√	√
_for$rew	-lveclib	VECLIB ch. 6,7,8			√	√	√
_for$s_rsue	-lveclib	VECLIB ch. 6,7,8			√	√	√
_for$s_wsfe	-lveclib	VECLIB ch. 6,7,8			√	√	√
_for$s_wsfi	-lveclib	VECLIB ch. 6,7,8			√	√	√
_for$s_wsue	-lveclib	VECLIB ch. 6,7,8			√	√	√
_for$do_lio	-lveclib	LSQPACK			√	√	√
_for$e_wsle	-lveclib	LSQPACK			√	√	√
_for$s_wsle	-lveclib	LSQPACK			√	√	√
_for$s_cat	-llapack	LAPACK		√			
_for$s_copy	-llapack	LAPACK		√			
_for$s_eq_l	-llapack	LAPACK		√			
_for$s_ne_l	-llapack	LAPACK		√			
_for$s_cat	-llapack8	LAPACK		√			
_for$s_copy	-llapack8	LAPACK		√			
_for$s_eq_l	-llapack8	LAPACK		√			
_for$s_ne_l	-llapack8	LAPACK		√			

† If the program will be run on a C1 Series architecture, use the mathC1 library; if it will be run only on C2 or C3 Series architectures, use the mathC2 library.

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 CONVEX SCILIB and CONVEX 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 a call traceback onto 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 onto 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  vsigl.c
#include <veclib.h>

main ()
{
    int n = 1;
    float alpha = 0.0;
    float x[10];
    int incx = 0;

    sflr1c_ (&n, &alpha, x, &incx);
}
% cc vsigl.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, ConvexOS 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, `initVECLIB`, 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 initVECLIB(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 `initVECLIB`, 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;

    initVECLIB (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 `initVECLIB` 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 are the core image and the exit status.

NOTE

The initialization functions for CONVEX SCILIB and CONVEX LAPACK also are called `initVECLIB`.

Any previously established signal handler for signal `sig` will be replaced by VECLIB's default signal handler.

Supplying Your Own Signal Handler

After calling `initVECLIB` to specify the 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.\n");
    exit(123);
}

main ()
{
    int n = 1;
    float alpha = 0.0;
    float x[10];
    int incx = 0;
    int oldsig;

    initVECLIB (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 `initVECLIB` 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, ConvexOS executes `veclib_sig`.

B

Calling CONVEX VECLIB From Ada

Introduction

Subprogram calls in CONVEX FORTRAN use the same calling convention as used by other CONVEX language processors. This CONVEX standard permits CONVEX VECLIB subprograms, which are written in FORTRAN or assembly language, to be called from Ada. This appendix describes calling conventions necessary to call VECLIB subprograms from Ada programs.

At the end of this appendix there is a general template for a VECLIB interface package. VECLIB from Ada in several ways, the best way is to integrate VECLIB calls in a package, as shown in that example.

General Rules

You should follow these rules to call a VECLIB subprogram from within an Ada program:

- Several VECLIB subprograms contain FORTRAN I/O statements, primarily for error reporting. To initialize the FORTRAN I/O runtime system, invoke the FORTRAN library function `f_init`. If you are using a VECLIB interface package, (see "Sample VECLIB Interface Package") place the call to `f_init` in the executable section of the VECLIB interface package body. If you are not using a VECLIB interface package, place the call to `f_init` at the beginning of your main program.

- All arguments to VECLIB subprograms are passed by address. This rule means you must pass only objects of type SYSTEM.ADDRESS as arguments to VECLIB subprograms.

- As explained in Chapter 2, VECLIB expects arrays to be stored in column-major order; however, Ada stores them in row-major order. Two ways to handle this incompatibility are:

  — In your Ada program, store the transpose of the matrix you are using so it appears 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 Ada) and use the transpose option to solve the intended problem.

- VECLIB subprograms that expect character arguments may be called from Ada with objects or constants of type STRING. For subroutine XERBLA, the quantity passed for argument name must be at least six characters long, and only six characters are significant.

- Because Ada does not directly support complex arithmetic, storage layouts for FORTRAN types COMPLEX*8 and COMPLEX*16 are usually defined by Ada record types. You can use the COMPLEX_ARITH generic package or use the following type statements:

```
type SHORT_COMPLEX is record
  REAL, IMAG: SHORT_FLOAT;
end record;

type COMPLEX is record
  REAL, IMAG: FLOAT;
end record;
```

- According to the CONVEX language processor calling conventions, a FORTRAN COMPLEX function subprogram does not directly return a function value. Instead, a COMPLEX function is passed an extra argument at the beginning of the argument list. This argument points to a COMPLEX*8 or COMPLEX*16 variable into which the function value is placed by the FORTRAN function. Consequently, a FORTRAN function that returns a COMPLEX data type is equivalent to an Ada procedure.

- Using the above Ada type declarations for COMPLEX and SHORT_COMPLEX, FORTRAN and Ada declarations are related as follows:

Table B-1: Relationship Between FORTRAN and Ada Declarations

FORTRAN	Ada
LOGICAL*4 m	M: INTEGER;
LOGICAL*8 n	N: INTEGER;
INTEGER*4 i	I: INTEGER;
INTEGER*8 l	L: LONG_INTEGER;
REAL*4 s	S: SHORT_FLOAT;
REAL*8 d	D: FLOAT;
COMPLEX*8 c	C: SHORT_COMPLEX;
COMPLEX*16 z	Z: COMPLEX;
CHARACTER*(n) t	T: STRING (1..N);
SUBROUTINE sub (...)	procedure SUB (...);
INTEGER*4 FUNCTION if1 (...)	function IF1 (...) return INTEGER;
REAL*4 FUNCTION sf (...)	function SF (...) return SHORT_FLOAT;
REAL*8 FUNCTION df (...)	function DF (...) return FLOAT;
COMPLEX*8 FUNCTION cf (...)	procedure CF (TEMP, ...); TEMP: SHORT_COMPLEX; -- return value through temp
COMPLEX*16 FUNCTION zf (...)	procedure ZF (TEMP, ...); TEMP: COMPLEX; -- return value through temp

- In addition to the VECLIB library, FORTRAN libraries must be linked into an Ada application that calls VECLIB. You can access the FORTRAN libraries with the Ada `-flink` option:

  ada ada_applic -lveclib -flink

Again, the order of the library names in the *a.ld* command is important.

Calling Procedures

This section gives an example of calling a VECLIB subprogram to perform the elementary vector operation $y = ax + y$. The VECLIB interface package used in this example is at the end of this appendix.

```
with VECLIB_ADA;
with SHORT_FLOAT_IO;
with TEXT_IO;

procedure TEST_SAXPY is
 X, Y : VECLIB_ADA.SHORT_FLOAT_1D(1 .. 10) :=
                                (1 .. 2 => 1.00, 3 => 3.00, 4 => 4.00, 5 .. 10 => 1.00);

begin
 TEXT_IO.PUT_LINE("SAXPY VECLIB test.");
 TEXT_IO.NEW_LINE;

 TEXT_IO.PUT_LINE("x before:");
 for I in 1 .. X'LENGTH loop
  SHORT_FLOAT_IO.PUT(X(I), FORE => 2, AFT => 2, EXP => 0);
 end loop;
 TEXT_IO.NEW_LINE;
 TEXT_IO.PUT_LINE("y before:");
 for I in 1 .. Y'LENGTH loop
  SHORT_FLOAT_IO.PUT(Y(I), FORE => 2, AFT => 2, EXP => 0);
 end loop;
 TEXT_IO.NEW_LINE;

 VECLIB_ADA.SAXPY_ADA(10, 5.00, X, 1, Y, 1);

 TEXT_IO.PUT_LINE("x after:");
 for I in 1 .. X'LENGTH loop
  SHORT_FLOAT_IO.PUT(X(I), FORE => 2, AFT => 2, EXP => 0);
 end loop;
 TEXT_IO.NEW_LINE;
 TEXT_IO.PUT_LINE("y after:");
 for I in 1 .. Y'LENGTH loop
  SHORT_FLOAT_IO.PUT(Y(I), FORE => 2, AFT => 2, EXP => 0);
 end loop;
 TEXT_IO.NEW_LINE;
end TEST_SAXPY;
```

Calling Functions

The method of calling a VECLIB function subprogram depends on its data type. The following examples show how to call some functions from Ada.

FLOAT Functions

The following program computes the dot product of two REAL*8 = FLOAT vectors of length 2 using the VECLIB subprogram DDOT:

```
with VECLIB_ADA;
with TEXT_IO;
with FLOAT_IO;

procedure TEST_DDOT is
 X : VECLIB_ADA.FLOAT_1D(1 .. 2) := (1.0, -2.0);
 Y : VECLIB_ADA.FLOAT_1D(1 .. 2) := (-3.0, -2.0);
 S : FLOAT;
begin
 TEXT_IO.PUT_LINE("DDOT VECLIB test.");
 TEXT_IO.NEW_LINE;
 TEXT_IO.PUT_LINE("x before:");
 for I in 1 .. X'LENGTH loop
  FLOAT_IO.PUT(X(I), FORE => 5, AFT => 2, EXP => 0);
 end loop;
 TEXT_IO.NEW_LINE;

 TEXT_IO.PUT_LINE("y before:");
 for I in 1 .. Y'LENGTH loop
  FLOAT_IO.PUT(Y(I), FORE => 5, AFT => 2, EXP => 0);
 end loop;
 TEXT_IO.NEW_LINE;

 S := VECLIB_ADA.DDOT_ADA(2, X, 1, Y, 1);

 TEXT_IO.PUT_LINE("s:");
 FLOAT_IO.PUT(S, FORE => 5, AFT => 2, EXP => 0);
 TEXT_IO.NEW_LINE;

end TEST_DDOT;
```

COMPLEX Functions

The following example illustrates how to call COMPLEX*16 = COMPLEX VECLIB function subprogram ZDOTC:

```
with VECLIB_ADA;
with COMPLEX_IO;
with TEXT_IO;

procedure TEST_ZDOTC is
 ZX, ZY : VECLIB_ADA.COMPLEX_1D(1 .. 10) := (1 .. 10 => (1.00, 2.00));
 ZZ    : VECLIB_ADA.COMPLEX;

begin
 TEXT_IO.PUT_LINE("ZDOTC VECLIB test.");
 TEXT_IO.NEW_LINE;

 ZX(3) := (4.00, -4.00);
 ZY(6) := (-1.00, 15.00);
 TEXT_IO.PUT_LINE("zx before:");
 COMPLEX_IO.PUT(ZX);
 TEXT_IO.PUT_LINE("zy before:");
 COMPLEX_IO.PUT(ZY);

 ZZ := VECLIB_ADA.ZDOTC_ADA(10, ZX, 1, ZY, 1);

 TEXT_IO.PUT_LINE("zz:");
 COMPLEX_IO.PUT(ZZ);
 TEXT_IO.NEW_LINE;

end TEST_ZDOTC;
```

```
with VECLIB_ADA;
with TEXT_IO;
with FLOAT_IO;
package COMPLEX_IO is
 procedure PUT(X : VECLIB_ADA.COMPLEX);
 procedure PUT(X : VECLIB_ADA.COMPLEX_1D);
end COMPLEX_IO;

package body COMPLEX_IO is

 procedure PUT(X : VECLIB_ADA.COMPLEX) is
 begin
  TEXT_IO.PUT(" (");
  FLOAT_IO.PUT(X.REAL, FORE => 3, AFT => 2, EXP => 0);
  TEXT_IO.PUT(", ");
  FLOAT_IO.PUT(X.IMAG, FORE => 3, AFT => 2, EXP => 0);
  TEXT_IO.PUT(")  ");
```

```
    end PUT;

    procedure PUT(X : VECLIB_ADA.COMPLEX_1D) is
    begin
     for I in X'range loop
      PUT(X(I));
      if ((I mod 3) = 0) then
                                    TEXT_IO.NEW_LINE;
      end if;
     end loop;
     TEXT_IO.NEW_LINE;
    end PUT;

  end COMPLEX_IO;
```

Sample VECLIB Interface Package

The following code illustrates a VECLIB interface package. The interface subprogram names are modified versions of the VECLIB subprogram names. The interface subprograms handle the differences in calling sequences between FORTRAN and Ada so that all machine-dependent code will be encapsulated within the VECLIB interface package.

```
package VECLIB_ADA is
 type SHORT_FLOAT_1D  is array(INTEGER range <>) of SHORT_FLOAT;
 type FLOAT_1D        is array(INTEGER range <>) of FLOAT;

 type SHORT_COMPLEX is record
  REAL, IMAG : SHORT_FLOAT;
 end record;
 type SHORT_COMPLEX_1D is array(INTEGER range <>) of SHORT_COMPLEX;

 type COMPLEX is record
  REAL, IMAG : FLOAT;
 end record;
 type COMPLEX_1D      is array(INTEGER range <>) of COMPLEX;

 procedure SAXPY_ADA(N      : INTEGER;
                     A      : SHORT_FLOAT;
                     X      : SHORT_FLOAT_1D;
                     INCR_X : INTEGER;
                     Y      : in out SHORT_FLOAT_1D;
                     INCR_Y : INTEGER);

 function DDOT_ADA(N      : INTEGER;
                   X      : FLOAT_1D;
                   INCR_X : INTEGER;
                   Y      : FLOAT_1D;
                   INCR_Y : INTEGER) return FLOAT;
```

```
 function ZDOTC_ADA(N      : INTEGER;
                    X      : COMPLEX_1D;
                    INCR_X : INTEGER;
                    Y      : COMPLEX_1D;
                    INCR_Y : INTEGER) return COMPLEX;

 procedure XERBLA_ADA(NAME : STRING; IARG : INTEGER);

private
 procedure F_INIT;
 pragma INTERFACE(C, F_INIT);

end VECLIB_ADA;
with SYSTEM;
use  SYSTEM;
package body VECLIB_ADA is

 procedure SAXPY(N      : ADDRESS;
                 A      : ADDRESS;
                 X      : ADDRESS;
                 INCR_X : ADDRESS;
                 Y      : ADDRESS;
                 INCR_Y : ADDRESS);
 pragma INTERFACE(FORTRAN, SAXPY);

 procedure SAXPY_ADA(N      : INTEGER;
                     A      : SHORT_FLOAT;
                     X      : SHORT_FLOAT_1D;
                     INCR_X : INTEGER;
                     Y      : in out SHORT_FLOAT_1D;
                     INCR_Y : INTEGER) is
 begin
  SAXPY(N'ADDRESS, A'ADDRESS, X'ADDRESS, INCR_X'ADDRESS,
     Y'ADDRESS, INCR_Y'ADDRESS);
 end SAXPY_ADA;

 function DDOT(N      : ADDRESS;
               X      : ADDRESS;
               INCR_X : ADDRESS;
               Y      : ADDRESS;
               INCR_Y : ADDRESS) return FLOAT;
 pragma INTERFACE(FORTRAN, DDOT);

 function DDOT_ADA(N      : INTEGER;
                   X      : FLOAT_1D;
                   INCR_X : INTEGER;
                   Y      : FLOAT_1D;
                   INCR_Y : INTEGER) return FLOAT is
  RESULT : FLOAT;
 begin
```

```
     RESULT := DDOT(N'ADDRESS, X'ADDRESS, INCR_X'ADDRESS,
             Y'ADDRESS, INCR_Y'ADDRESS);
     return RESULT;
    end DDOT_ADA;

    procedure ZDOTC(RESULT : ADDRESS;
            N      : ADDRESS;
            X      : ADDRESS;
            INCR_X : ADDRESS;
            Y      : ADDRESS;
            INCR_Y : ADDRESS);
    pragma INTERFACE(FORTRAN, ZDOTC);

    function ZDOTC_ADA(N      : INTEGER;
             X      : COMPLEX_1D;
             INCR_X : INTEGER;
             Y      : COMPLEX_1D;
             INCR_Y : INTEGER) return COMPLEX is
     RESULT : COMPLEX;
    begin
     ZDOTC(RESULT'ADDRESS, N'ADDRESS, X'ADDRESS, INCR_X'ADDRESS,
        Y'ADDRESS, INCR_Y'ADDRESS);
     return RESULT;
    end ZDOTC_ADA;

    procedure XERBLA(NAME : ADDRESS; IARG : ADDRESS);
    pragma INTERFACE(FORTRAN, XERBLA);

    procedure XERBLA_ADA(NAME : STRING; IARG : INTEGER) is
    begin
     XERBLA(NAME'ADDRESS, IARG'ADDRESS);
    end XERBLA_ADA;
   begin
    F_INIT;
   end VECLIB_ADA;
```

Index

A

accessing CONVEX VECLIB 1-2
Ada, calling VECLIB from B-1
Ada language B-1
allocate dynamic memory 12-4, 12-8
allocate dynamic own array 12-11, 12-13
Application Compiler 1-6
arithmetic format 1-6
ASAP, automatic self allocating processors 1-4
automatic reordering 8-27
automatic self allocating processors (ASAP) 1-4

B

backward storage 2-3
BAKVEC 5-14
BALANC 5-14
BALBAK 5-14
band matrix 3-5, 3-22, 3-58, 3-62, 4-6, 4-10, 4-13, 4-17, 4-31, 4-33, 4-36, 4-40, 4-43, 4-46, 4-58
BANDR 5-14
BANDV 5-14
Basic Linear Algebra Subprograms 1-2, 2-1
BEGIN_TASKS compiler directive 1-5
bibliography xxv
BISECT 5-14
BLAS 1-2, 2-1, 3-86
BLAS, Extended 1-2, 3-1, 4-1, 5-1
BLAS indexing conventions 2-3
BLAS, Level 1 1-2
BLAS, Level 2 1-2, 3-1, 4-1, 5-1
BLAS, Level 3 1-2, 3-1, 4-1, 5-1
BLAS, Sparse 1-2, 2-1
BQR 5-14

C

C, calling VECLIB from A-1
C header files A-3
C language A-1
C1DFFT 9-3
C2DFFT 9-8
C3DFFT 9-12
Calling VECLIB from Ada B-1
Calling VECLIB from C A-1
CAXPY 2-25
CAXPYC 2-25
CAXPYI 2-28
CBABK2 5-14
CBAL 5-14
CCHDC 4-60
CCHDD 4-60
CCHEX 4-60
CCHUD 4-60
CCOPY 2-36
CCOPYC 2-36
CDOTC 2-39
CDOTCI 2-42
CDOTU 2-39
CDOTUI 2-42
–cfc compiler option 1-2, 1-8
CFFTS 9-16
CG 5-14
CGBCO 4-6
CGBDI 4-10
CGBFA 4-13
CGBMV 3-5
CGBSL 4-17
CGECO 4-20
CGEDI 4-23
CGEFA 4-26
CGEMM 3-9
CGEMMS 1-3, 3-12
CGEMV 3-16
CGERC 3-19
CGERU 3-19
CGESL 4-28
CGTHR 2-46
CGTHRZ 2-48
CGTSL 4-31
CGTSV 1-3, 4-33
CH 5-14
CHBMV 3-22
check accuracy of eigenvalue/eigenvector results 7-49
CHEMM 3-37
CHEMV 3-41
CHER 3-44
CHER2 3-47
CHER2K 3-50
CHERK 3-54
CHICO 4-60
CHIDI 4-60
CHIFA 4-60
CHISL 4-60
Cholesky factorization 4-36, 4-43, 4-48, 4-54
CHPCO 4-60
CHPDI 4-60
CHPFA 4-60
CHPMV 3-26
CHPR 3-30
CHPR2 3-33
CHPSL 4-60
CINVIT 5-14
clear vector 2-87
clip vector 2-30, 2-32, 2-34
CLSTEQ 2-50
CLSTNE 2-50
COMBAK 5-14
COMHES 5-14
COMLR 5-14
COMLR2 5-14
compiler directives 1-5
COMQR 5-14
COMQR2 5-14
condition number 4-4, 4-6, 4-20, 4-36, 4-48, 6-27
conversion, floating-point 12-19, 12-21
CONVEX VECLIB Programmer's Reference 1-10

convolution 10-2
correlation 10-2
correlation and convolution subprograms 10-1
CORTB 5-14
CORTH 5-14
count vector elements 2-11
CPBCO 4-36
CPBDI 4-40
CPBFA 4-43
CPBSL 4-46
CPOCO 4-48
CPODI 4-51
CPOFA 4-54
CPOSL 4-56
CPPCO 4-60
CPPDI 4-61
CPPFA 4-61
CPPSL 4-61
CPTSL 4-58
CQRDC 4-61
CQRSL 4-61
CRC1FT 9-24
CRC2FT 9-29
CRC3FT 9-34
CRCFTS 9-40
CROT 2-63
CROTG 2-66
CRSCL 2-74
CSCAL 2-76
CSCALC 2-76
CSCTR 2-78
CSICO 4-61
CSIDI 4-61
CSIFA 4-61
CSISL 4-61
CSPCO 4-61
CSPDI 4-62
CSPFA 4-62
CSPSL 4-62
CSROT 2-63
CSRSCL 2-74
CSSCAL 2-76
CSUM 2-80
CSVDC 4-62
CSWAP 2-82
CSYMM 3-37
CSYR2K 3-50
CSYRK 3-54
CTBMV 3-58
CTBSV 3-62
CTPMV 3-66
CTPSV 3-70
CTRCO 4-62
CTRDI 4-62
CTRMM 3-74
CTRMV 3-77
CTRSL 4-62
CTRSM 3-80
CTRSV 3-83
CWDOTC 2-84
CWDOTU 2-84
CXpa 1-5
CZERO 2-87

D

D1DFFT 9-5
D2DFFT 9-10
D3DFFT 9-14
DALLOC 12-3
DAMAX 2-19
DAMIN 2-21
DASUM 2-23
DAXPY 2-25
DAXPYI 2-28
DCHDC 4-60
DCHDD 4-60
DCHEX 4-60
DCHUD 4-60
DCLIP 2-30
DCLIPL 2-32
DCLIPR 2-34
DCONV 10-2
DCOPY 2-36
DDOT 2-39
DDOTI 2-42
deallocate dynamic own array 12-3
deallocate working storage 6-30, 7-50, 8-44
determinant 4-4, 4-10, 4-23, 4-40, 4-51
DFFTS 9-20
DFLR1C 11-5
DFLR1M 11-2
DFLR1P 11-2
DFLR2C 11-10
DFLR2M 11-7
DFLR2P 11-7
DFLRLM 11-13
DFLRLP 11-13
DFRAC 2-44
DFT 9-1
DGBCO 4-6
DGBDI 4-10
DGBFA 4-13
DGBMV 3-5
DGBSL 4-17
DGECO 4-20
DGEDI 4-23
DGEFA 4-26
DGEMM 3-9
DGEMMS 3-12
DGEMV 3-16
DGER 3-19
DGESL 4-28
DGTHR 2-46
DGTHRZ 2-48
DGTSL 4-31
DGTSV 1-3, 4-33
discrete Fourier transform 9-1
DLSTEQ 2-50
DLSTGE 2-50
DLSTGT 2-50

DLSTLE 2-50
DLSTLT 2-50
DLSTNE 2-50
DMAX 2-53
DMIN 2-55
DNRM2 2-57
DNRSQ 2-59
documentation, online 1-10
documentation, ordering xxvi
dot product 2-39, 2-42
dot product, weighted 2-84
DPBCO 4-36
DPBDI 4-40
DPBFA 4-43
DPBSL 4-46
DPOCO 4-48
DPODI 4-51
DPOFA 4-54
DPOSL 4-56
DPPCO 4-60
DPPDI 4-61
DPPFA 4-61
DPPROD 11-16
DPPSL 4-61
DPSUM 11-18
DPTSL 4-58
DQRDC 4-61
DQRSL 4-61
DRAMP 2-61
DRAN 12-23
DRANV 12-25
DRC1FT 9-26
DRC2FT 9-31
DRC3FT 9-37
DRCFTS 9-44
DROT 2-63
DROTG 2-66
DROTI 2-68
DROTM 2-70
DROTMG 2-72
DRSCL 2-74
DSBMV 3-22
DSCAL 2-76
DSCTR 2-78
DSEVCK 7-49
DSEVDA 7-50
DSEVE1 7-12
DSEVES 7-38
DSEVEX 7-41
DSEVI1 7-20
DSEVIC 7-21
DSEVIE 7-23
DSEVIF 7-27
DSEVIM 7-25
DSEVIN 7-18
DSEVOC 7-51
DSEVOR 7-28
DSEVPS 7-52
DSEVRC 7-45
DSEVRL 7-47
DSEVRS 7-53
DSEVSV 7-54
DSEVV1 7-29
DSEVVC 7-31
DSEVVD 7-33
DSEVVE 7-34
DSEVVM 7-36
DSICO 4-61
DSIDI 4-61
DSIFA 4-61
DSISL 4-61
DSKYDA 8-44
DSKYDF 8-40
DSKYDS 8-42
DSKYFA 8-38
DSKYFS 8-13
DSKYFX 8-15
DSKYI1 8-19
DSKYIC 8-20
DSKYIE 8-21
DSKYIF 8-26
DSKYIM 8-22
DSKYIN 8-18
DSKYIS 8-24
DSKYOC 8-45
DSKYOR 8-27
DSKYOU 8-28
DSKYPS 8-46
DSKYRS 8-47
DSKYSL 8-39
DSKYSR 8-48
DSKYSV 8-49
DSKYV1 8-29
DSKYVC 8-30
DSKYVE 8-32
DSKYVM 8-34
DSKYVS 8-36
DSLECO 6-27
DSLEDA 6-30
DSLEFA 6-28
DSLEFS 6-10
DSLEI1 6-14
DSLEIC 6-15
DSLEIE 6-16
DSLEIF 6-19
DSLEIM 6-17
DSLEIN 6-13
DSLEOC 6-31
DSLEOR 6-20
DSLEPS 6-32
DSLERS 6-33
DSLESL 6-29
DSLESR 6-34
DSLESV 6-35
DSLEV1 6-21
DSLEVC 6-22
DSLEVE 6-24
DSLEVM 6-25
DSLR2 11-23
DSLR3 11-26
DSLRL 11-20
DSORT 12-27

DSPCO 4-61
DSPDI 4-62
DSPFA 4-62
DSPMV 3-26
DSPR 3-30
DSPR2 3-33
DSPSL 4-62
DSUM 2-80
DSVDC 4-62
DSWAP 2-82
DSYMM 3-37
DSYMV 3-41
DSYR 3-44
DSYR2 3-47
DSYR2K 3-50
DSYRK 3-54
DTBMV 3-58
DTBSV 3-62
DTPMV 3-66
DTPSV 3-70
DTRCO 4-62
DTRDI 4-62
DTRMM 3-74
DTRMV 3-77
DTRSL 4-62
DTRSM 3-80
DTRSV 3-83
DWDOT 2-84
DYNAMIC 12-4
dynamic storage allocation 12-3, 12-4, 12-8, 12-11, 12-13
dynamic storage deallocation 12-3
DZAMAX 2-19
DZAMIN 2-21
DZASUM 2-23
DZERO 2-87
DZNRM2 2-57
DZNRSQ 2-59

E
eigenvalues 5-1, 5-3, 5-5, 5-8, 5-10, 5-12
eigenvalues, sparse 7-1
eigenvectors 5-1, 5-3, 5-5, 5-12
eigenvectors, sparse 7-1
EISPACK 1-2, 5-1
EISPACK Guide 1-11
EISPACK Guide Extension 1-11
ELMBAK 5-14
ELMHES 5-14
ELTRAN 5-14
end of matrix structure input 6-19, 7-27, 8-26
END_TASKS compiler directive 1-5
envelope data structure 8-1, 8-2
error handler 3-86
error handling, VECLIB 1-9
error reporting 1-9
Euclidean norm 2-57, 2-59
Extended BLAS 1-2, 3-1, 4-1, 5-1

F
factorization, numeric 6-27, 6-28, 8-38, 8-40
factorization, sparse 6-27, 6-28, 8-38, 8-40
factorization, symbolic 6-20
fast Fourier transforms 9-1
FFT, one-dimensional 9-3, 9-5, 9-24, 9-26
FFT, one-dimensional, simultaneous 9-16, 9-20, 9-40, 9-44
FFT, real-to-complex 9-24, 9-26, 9-29, 9-31, 9-34, 9-37, 9-40, 9-44
FFT, three-dimensional 9-12, 9-14, 9-34, 9-37
FFT, two-dimensional 9-8, 9-10, 9-29, 9-31
FIGI 5-14
FIGI2 5-14
filtering 10-2
find 2-17, 2-50
first order linear recurrence 11-2, 11-5, 11-7, 11-10, 11-13
floating point format 1-6
floating-point conversion 12-19, 12-21
FORCE_PARALLEL compiler directive 1-5
FORTRAN array argument association 2-2
FORTRAN storage of arrays 2-2
forward storage 2-3
fractional part 2-44
further reference xxv

G
gather 2-46, 2-48
Givens rotation 2-63, 2-66, 2-68
Givens rotation, modified 2-70, 2-72

H
header files for C A-3
high-level subprograms 1-9
HQR 5-14
HQR2 5-14
HTRIB3 5-14
HTRIBK 5-14
HTRID3 5-14
HTRIDI 5-14

I
IAMAX 2-19
IAMIN 2-21
IASUM 2-23
ICAMAX 1-3, 1-4, 2-7
ICAMIN 2-9
ICCTEQ 2-11
ICCTNE 2-11
ICLIP 2-30
ICLIPL 2-32
ICLIPR 2-34
ICOPY 2-36
ICSVEQ 2-17
ICSVNE 2-17
IDAMAX 2-7
IDAMIN 2-9
IDCTEQ 2-11

IDCTGE 2-11
IDCTGT 2-11
IDCTLE 2-11
IDCTLT 2-11
IDCTNE 2-11
IDMAX 2-13
IDMIN 2-15
IDSVEQ 2-17
IDSVGE 2-17
IDSVGT 2-17
IDSVLE 2-17
IDSVLT 2-17
IDSVNE 2-17
IEEE arithmetic format 1-6
ier 1-9
IGTHR 2-46
IGTHRZ 2-48
IIAMAX 2-7
IIAMIN 2-9
IICTEQ 2-11
IICTGE 2-11
IICTGT 2-11
IICTLE 2-11
IICTLT 2-11
IICTNE 2-11
IIMAX 2-13
IIMIN 2-15
IISVEQ 2-17
IISVGE 2-17
IISVGT 2-17
IISVLE 2-17
IISVLT 2-17
IISVNE 2-17
ILSTEQ 2-50
ILSTGE 2-50
ILSTGT 2-50
ILSTLE 2-50
ILSTLT 2-50
ILSTNE 2-50
IMAX 2-53
IMIN 2-55
IMTQL1 5-14
IMTQL2 5-14
IMTQLV 5-14
increment 2-3
increment arguments 2-3
INCX 2-3
index 2-17, 2-50
index of maximum 2-13
index of maximum absolute value 2-7
index of minimum 2-15
index of minimum absolute value 2-9
initialize skyline linear equations 8-18
initialize sparse eigenvalues/eigenvectors 7-18
initialize sparse linear equations 6-13
initialize vector to zero 2-87
inner product 2-39, 2-42
inner product, weighted 2-84
input, matrix structure, by column 6-15, 7-21, 8-20
input, matrix structure, by finite element 6-16, 7-23, 8-21
input, matrix structure, by matrix 6-17, 7-25, 8-22
input, matrix structure, by single entry 6-14, 7-20, 8-19
input, matrix structure, by skyline matrix 8-24
input, matrix structure, end of 6-19, 7-27, 8-26
input, matrix value, by column 6-22, 7-31, 8-30
input, matrix value, by finite element 6-24, 7-34, 8-32
input, matrix value, by matrix 6-25, 7-36, 8-34
input, matrix value, by single entry 6-21, 7-29, 8-29
input, matrix value, by skyline matrix 8-36
input, matrix value, to main diagonal 7-33
interlanguage programming A-1
inverse 4-4, 4-23, 4-51
INVIT 5-14
IPSUM 11-18
IRAMP 2-61
ISAMAX 1-3, 1-4, 2-7
ISAMIN 1-3, 2-9
ISCTEQ 2-11
ISCTGE 2-11
ISCTGT 2-11
ISCTLE 2-11
ISCTLT 2-11
ISCTNE 2-11
ISCTR 2-78
ISMAX 1-3, 2-13
ISMIN 1-3, 2-15
ISORT 12-27
ISSVEQ 2-17
ISSVGE 2-17
ISSVGT 2-17
ISSVLE 2-17
ISSVLT 2-17
ISSVNE 2-17
ISUM 2-80
ISWAP 2-82
IZAMAX 2-7
IZAMIN 2-9
IZCTEQ 2-11
IZCTNE 2-11
IZERO 2-87
IZSVEQ 2-17
IZSVNE 2-17

L

Lanczos method 7-1
LAPACK 1-2, 1-3, 3-86, A-1, A-3, A-8, A-9
Level 1 BLAS 1-2
Level 2 BLAS 1-2, 3-1, 3-86, 4-1, 5-1
Level 3 BLAS 1-2, 3-1, 3-86, 4-1, 5-1
linear equations 4-1

linear equations, skyline 8-1
linear equations, sparse 6-1
linear recurrence 11-2, 11-5, 11-7, 11-10, 11-13, 11-20, 11-23, 11-26
LINPACK 1-2, 4-1
LINPACK Users' Guide 1-11
locate 2-17, 2-50
low-level subprograms 1-9
–lscilib compiler option 1-3
LU factorization 4-6, 4-13, 4-20, 4-26
–lveclib compiler option 1-2
–lveclib8 compiler option 1-2

M

MALLOC 12-8
man pages 1-10
matrix inverse 4-4, 4-23, 4-51
matrix structure input by column 6-15, 7-21, 8-20
matrix structure input by finite element 6-16, 7-23, 8-21
matrix structure input by matrix 6-17, 7-25, 8-22
matrix structure input by single entry 6-14, 7-20, 8-19
matrix structure input by skyline matrix 8-24
matrix value input by column 6-22, 7-31, 8-30
matrix value input by finite element 6-24, 7-34, 8-32
matrix value input by matrix 6-25, 7-36, 8-34
matrix value input by single entry 6-21, 7-29, 8-29
matrix value input by skyline matrix 8-36
matrix value input to main diagonal 7-33
matrix-matrix multiply 3-9, 3-37
matrix-matrix multiply, triangular 3-74
matrix-vector multiply 3-5, 3-16, 3-22, 3-26, 3-41, 3-58, 3-66, 3-77
maximum 2-7, 2-13, 2-19, 2-53
message level 6-31, 7-51, 8-45
MINFIT 5-14
minimum 2-9, 2-15, 2-21, 2-55
modified Givens rotation 2-70, 2-72

N

NALLOC 12-11
native arithmetic format 1-6
NEXT_TASK compiler directive 1-5
nonvolatile dynamic memory 12-3, 12-11, 12-13
norm 2-19, 2-23, 2-57, 2-59
numeric factorization 6-27, 6-28, 8-38, 8-40

O

one-call usage 6-10
one-call usage for skyline equations 8-13, 8-15
online documentation 1-10
ordering documentation xxvi
ORTBAK 5-15
ORTHES 5-15
ORTRAN 5-15
output control 6-31, 7-51, 8-45
output unit 6-31, 7-51, 8-45

P

–p8 compiler option 1-2, 1-8
packed matrix 3-26, 3-30, 3-33, 3-66, 3-70
parallel processing 1-4
partial product 11-16
partial sum 11-18
–pd8 compiler option 1-2, 1-8
performance analysis 1-5
polynomial, evaluate 11-14
positive definite matrix 4-36, 4-40, 4-43, 4-46, 4-48, 4-51, 4-54, 4-56, 4-58
print statistics 6-32, 7-52, 8-46
product, partial 11-16
profile data structure 8-1, 8-2
profiling 1-5
programmer's reference 1-10

Q

QZHES 5-15
QZIT 5-15
QZVAL 5-15
QZVEC 5-15

R

RALLOC 12-13
ramp function 2-61
RAN 12-1, 12-15
random-number generator 12-1, 12-15, 12-17, 12-23, 12-25
rank-2*k* update 3-50
rank-*k* update 3-54
rank-one update 3-19, 3-30, 3-44
rank-two update 3-33, 3-47
RANV 12-1, 12-17
RATQR 5-15
reallocate dynamic own array 12-13
REBAK 5-15
REBAKB 5-15
recurrence, first order 11-2, 11-5, 11-7, 11-10, 11-13
recurrence, linear 11-2, 11-5, 11-7, 11-10, 11-13, 11-20, 11-23, 11-26
recurrence, second order 11-20, 11-23, 11-26
REDUC 5-15
REDUC2 5-15
reentrance 1-5
reordering 6-20, 7-28
reordering, automatic 8-27
reordering, user-supplied 8-28
restore sparse eigenvalue/eigenvector state 7-53

restore sparse linear equations state 6-33, 8-47
retrieve runtime statistics 6-34, 8-48
return eigenvalue results 7-47
return sparse eigenvalue/eigenvector results 7-45
RG 5-15
RGG 5-15
RS 5-3
RSB 5-15
RSG 5-15
RSGAB 5-15
RSGBA 5-15
RSM 5-15
RSP 5-15
RST 5-15
RT 5-15
runtime statistics 6-34, 8-48

S

S1DFFT 9-5
S2DFFT 9-10
S3DFFT 9-14
SAMAX 2-19
SAMIN 2-21
SASUM 2-23
save sparse eigenvalue/eigenvector state 7-54
save sparse linear equation state 6-35, 8-49
SAXPY 2-25
SAXPYI 2-28
SC2IBM 12-19
SCAMAX 2-19
SCAMIN 2-21
SCASUM 2-23
scatter 2-78
SCHDC 4-60
SCHDD 4-60
SCHEX 4-60
SCHUD 4-60
SCILIB 1-3, 3-86, A-1, A-3, A-8, A-9
SCLIP 2-30
SCLIPL 2-32
SCLIPR 2-34
SCNRM2 2-57
SCNRSQ 2-59
SCONV 10-2
SCOPY 2-36
SDOT 2-39
SDOTI 2-42
search vector 2-17, 2-50
second order linear recurrence 11-20, 11-23, 11-26
SFFTS 9-20
SFLR1C 11-5
SFLR1M 11-2
SFLR1P 11-2
SFLR2C 11-10
SFLR2M 11-7
SFLR2P 11-7
SFLRLM 11-13
SFLRLP 11-13
SFRAC 2-44
SGBCO 4-6
SGBDI 4-10
SGBFA 4-13
SGBMV 3-5
SGBSL 4-17
SGECO 4-20
SGEDI 4-23
SGEFA 4-26
SGEMM 3-9
SGEMMS 1-3, 3-12
SGEMV 3-16
SGER 3-19
SGESL 4-28
SGTHR 2-46
SGTHRZ 2-48
SGTSL 4-31
SGTSV 1-3, 4-33
SIBM2C 12-21
skyline data structure 8-1
skyline equations, one-call usage 8-13, 8-15
skyline linear equations 8-1
skyline linear equations, initialize 8-18
SLSTEQ 2-50
SLSTGE 2-50
SLSTGT 2-50
SLSTLE 2-50
SLSTLT 2-50
SLSTNE 2-50
SMAX 2-53
SMIN 2-55
SNRM2 2-57
SNRSQ 2-59
solve right hand side vector 8-39
solve right-hand side vector 8-42
solve sparse linear equations 6-29
sort array 12-27
sparse 2-1, 2-28, 2-42, 2-46, 2-48, 2-50, 2-68, 2-78, 6-1, 7-1
Sparse BLAS 1-2, 2-1
sparse eigenextraction 7-38, 7-41
sparse eigenvalues and eigenvectors 7-1, 7-12
sparse eigenvalues/eigenvectors, initialize 7-18
sparse linear equations 6-1
sparse linear equations, initialize 6-13
SPBCO 4-36
SPBDI 4-40
SPBFA 4-43
SPBSL 4-46
SPOCO 4-48
SPODI 4-51
SPOFA 4-54
SPOSL 4-56
SPPCO 4-60
SPPDI 4-61
SPPFA 4-61
SPPROD 11-16
SPPSL 4-61
SPSUM 11-18

SPTSL 4-58
SQRDC 4-61
SQRSL 4-61
SRAMP 2-61
SRAN 12-23
SRANV 12-25
SRC1FT 9-26
SRC2FT 9-31
SRC3FT 9-37
SRCFTS 9-44
SROT 2-63
SROTG 2-66
SROTI 2-68
SROTM 2-70
SROTMG 2-72
SRSCL 2-74
SSBMV 3-22
SSCAL 2-76
SSCTR 2-78
SSICO 4-61
SSIDI 4-61
SSIFA 4-61
SSISL 4-61
SSLR2 11-23
SSLR3 11-26
SSLRL 11-20
SSORT 12-27
SSPCO 4-61
SSPDI 4-62
SSPFA 4-62
SSPMV 3-26
SSPR 3-30
SSPR2 3-33
SSPSL 4-62
SSUM 2-80
SSVDC 4-62
SSWAP 2-82
SSYMM 3-37
SSYMV 3-41
SSYR 3-44
SSYR2 3-47
SSYR2K 3-50
SSYRK 3-54
standardization 1-2
STBMV 3-58
STBSV 3-62
storage, backward 2-3
storage, forward 2-3
STPMV 3-66
STPSV 3-70
Strassen matrix-matrix multiply 3-12
STRCO 4-62
STRDI 4-62
stride 2-3
stride arguments 2-3
STRMM 3-74
STRMV 3-77
STRSL 4-62
STRSM 3-80
STRSV 3-83
sum, partial 11-18
supplemental reading xxv, 1-11, 2-4, 3-3, 4-5, 5-2, 6-8, 7-10, 8-10, 9-2, 10-1, 12-2
SVD 5-15
SWDOT 2-84
symbolic factorization 6-20, 7-28
SZERO 2-87

T

TAC, technical assistance center xxvi
technical assistance center, TAC xxvi
thread, definition 1-4
TINVIT 5-15
TQL1 5-15
TQL2 5-5
TQLRAT 5-8
TRBAK1 5-15
TRBAK3 5-15
TRED1 5-10
TRED2 5-12
TRED3 5-15
triangular factorization 4-6, 4-13, 4-20, 4-26
triangular matrix-matrix multiply 3-74
triangular solve 3-62, 3-70, 3-80, 3-83
tridiagonal linear equations 4-31, 4-33, 4-58
TRIDIB 5-15
TSTURM 5-15

U

user-supplied reordering 8-28

V

variable-band data structure 8-1, 8-2
VECLIB 1-2, 1-3
VECLIB error handling 1-9
VECLIB man pages 1-10
VECLIB8 1-2
VectorPak 1-2

X

XERBLA 3-86
XERVEC 12-29

Z

Z1DFFT 9-3
Z2DFFT 9-8
Z3DFFT 9-12
ZAXPY 2-25
ZAXPYC 2-25
ZAXPYI 2-28
ZCHDC 4-60
ZCHDD 4-60
ZCHEX 4-60
ZCHUD 4-60
ZCOPY 2-36
ZCOPYC 2-36
ZDOTC 2-39
ZDOTCI 2-42
ZDOTU 2-39
ZDOTUI 2-42

ZDROT 2-63
ZDRSCL 2-74
ZDSCAL 2-76
zero vector 2-87
ZFFTS 9-16
ZGBCO 4-6
ZGBDI 4-10
ZGBFA 4-13
ZGBMV 3-5
ZGBSL 4-17
ZGECO 4-20
ZGEDI 4-23
ZGEFA 4-26
ZGEMM 3-9
ZGEMMS 3-12
ZGEMV 3-16
ZGERC 3-19
ZGERU 3-19
ZGESL 4-28
ZGTHR 2-46
ZGTHRZ 2-48
ZGTSL 4-31
ZGTSV 1-3, 4-33
ZHBMV 3-22
ZHEMM 3-37
ZHEMV 3-41
ZHER 3-44
ZHER2 3-47
ZHER2K 3-50
ZHERK 3-54
ZHICO 4-60
ZHIDI 4-60
ZHIFA 4-60
ZHISL 4-60
ZHPCO 4-60
ZHPDI 4-60
ZHPFA 4-60
ZHPMV 3-26
ZHPR 3-30
ZHPR2 3-33
ZHPSL 4-60
ZLSTEQ 2-50
ZLSTNE 2-50
ZPBCO 4-36
ZPBDI 4-40
ZPBFA 4-43
ZPBSL 4-46
ZPOCO 4-48
ZPODI 4-51
ZPOFA 4-54
ZPOSL 4-56
ZPPCO 4-60
ZPPDI 4-61
ZPPFA 4-61
ZPPSL 4-61
ZPTSL 4-58
ZQRDC 4-61
ZQRSL 4-61
ZRC1FT 9-24
ZRC2FT 9-29
ZRC3FT 9-34
ZRCFTS 9-40
ZROT 2-63
ZROTG 2-66
ZRSCL 2-74
ZSCAL 2-76
ZSCALC 2-76
ZSCTR 2-78
ZSICO 4-61
ZSIDI 4-61
ZSIFA 4-61
ZSISL 4-61
ZSPCO 4-61
ZSPDI 4-62
ZSPFA 4-62
ZSPSL 4-62
ZSUM 2-80
ZSVDC 4-62
ZSWAP 2-82
ZSYMM 3-37
ZSYR2K 3-50
ZSYRK 3-54
ZTBMV 3-58
ZTBSV 3-62
ZTPMV 3-66
ZTPSV 3-70
ZTRCO 4-62
ZTRDI 4-62
ZTRMM 3-74
ZTRMV 3-77
ZTRSL 4-62
ZTRSM 3-80
ZTRSV 3-83
ZWDOTC 2-84
ZWDOTU 2-84
ZZERO 2-87

Order Number
DSW-132

Document Number
710-011030-002