Engineering and Scientific Subroutine Library for AIX Guide and Reference

Document Number SA22-7272-00

Notes!

- Before using this information and the product it supports, be sure to read the general information under "Special Notices" on page xvii.
- For a summary of changes for ESSL Version 3, see page xxxi.

First Edition (December 1997)

This edition applies to Version 3 of the IBM* Engineering and Scientific Subroutine Library (ESSL) for Advanced Interactive Executive (AIX)* licensed program, program number 5765-C42.

In this document, ESSL refers to the above version of ESSL for AIX. Changes are periodically made to the information herein.

Order IBM publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

IBM welcomes your comments. A form for your comments appears at the back of this publication. If the form has been removed, address your comments to:

International Business Machines Corporation
Department 55JA, Mail Station P384
522 South Road
Poughkeepsie,NY 12601-5400
United States of America

FAX (United States & Canada): 1+ 914+ 432-9405
FAX (Other Countries): Your International Access Code + 1+ 914+ 432-9405

IBMLink (United States customers only): KGNVMC(MHVRCFS)
IBM Mail Exchange: USIB6TC9 at IBMMAIL
Internet e-mail: mhvrcfs@vnet.ibm.com
World Wide Web: http://www.rs6000.ibm.com

If you would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:

Title and order number of this book
Page number or topic related to your comment

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1997. All rights reserved.
Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Looking for a Subroutine? xv

Special Notices xvii
Trademarks xvii
Programming Interfaces xviii

About This Book xix
How to Use This Book xix
How to Find a Subroutine Description xx
Where to Find Related Publications xx
How to Look Up a Bibliography Reference xxi
Special Terms xxi
Short and Long Precision xxi
Subroutines and Subprograms xxi
How to Interpret the Subroutine Names with a Prefix Underscore xxi
Abbreviated Names xxi
Fonts xxii
Special Notations and Conventions xxii
Scalar Data xxii
Vectors xxiii
Matrices xxiii
Sequences xxiii
Arrays xxiv
Special Characters, Symbols, Expressions, and Abbreviations xxv
How to Interpret the Subroutine Descriptions xxvii
Description xxviii
Syntax xxviii
On Entry xxviii
On Return xxviii
Notes xxix
Function xxix
Special Usage xxix
Error Conditions xxix
Examples xxix

What's New for ESSL for AIX xxxi
What's New for ESSL Version 3 xxxi
Future Migration xxxi

In Brief—What's Provided in ESSL for AIX xxxiii

Part 1. Guide Information 1

Chapter 1. Introduction and Requirements 3
Overview of ESSL 3
Performance and Functional Capability 3
Usability 4
The Variety of Mathematical Functions 4
ESSL—Processing Capabilities 6

© Copyright IBM Corp. 1997

Accuracy of the Computations . . . 6
High Performance of ESSL . . . 6
The Fortran Language Interface to the Subroutines . . . 7
Software and Hardware Products That Can Be Used with ESSL . . . 7
For ESSL—Hardware . . . 8
ESSL—Operating Systems . . . 8
ESSL—Software Products . . . 8
Installation and Customization Products . . . 8
Software Products for Displaying ESSL Online Information . . . 8
ESSL Internet Resources . . . 9
Obtaining Documentation . . . 9
Accessing ESSL's Product Home Pages . . . 9
Getting on the ESSL Mailing List . . . 9
List of ESSL Subroutines . . . 10
Linear Algebra Subprograms . . . 10
Matrix Operations . . . 14
Linear Algebraic Equations . . . 15
Eigensystem Analysis . . . 18
Fourier Transforms, Convolutions and Correlations, and Related Computations . . . 19
Sorting and Searching . . . 21
Interpolation . . . 22
Numerical Quadrature . . . 22
Random Number Generation . . . 23
Utilities . . . 23

Chapter 2. Planning Your Program . . . 25
Selecting an ESSL Subroutine . . . 25
Which ESSL Library Do You Want to Use? . . . 25
What Type of Data Are You Processing in Your Program? . . . 27
How Is Your Data Structured? And What Storage Technique Are You Using? . . . 28
What about Performance and Accuracy? . . . 28
Avoiding Conflicts with Internal ESSL Routine Names That are Exported . . . 28
Setting Up Your Data . . . 28
How Do You Set Up Your Scalar Data? . . . 28
How Do You Set Up Your Arrays? . . . 29
How Should Your Array Data Be Aligned? . . . 29
What Storage Mode Should You Use for Your Data? . . . 29
How Do You Convert from One Storage Mode to Another? . . . 29
Setting Up Your ESSL Calling Sequences . . . 30
What Is an Input-Output Argument? . . . 30
What Are the General Rules to Follow when Specifying Data for the Arguments? 30
What Happens When a Value of 0 Is Specified for N? . . . 31
How Do You Specify the Beginning of the Data Structure in the ESSL Calling Sequence? . . . 31
Using Auxiliary Storage in ESSL . . . 31
Dynamic Allocation of Auxiliary Storage . . . 32
Setting Up Auxiliary Storage When Dynamic Allocation Is Not Used . . . 33
Who Do You Want to Calculate the Size? You or ESSL? . . . 33
How Do You Calculate the Size Using the Formulas? . . . 33
How Do You Get ESSL to Calculate the Size Using ESSL Error Handling? . . . 33
Providing a Correct Transform Length to ESSL . . . 38
What ESSL Subroutines Require Transform Lengths? . . . 38
Who Do You Want to Calculate the Length? You or ESSL? . . . 39
How Do You Calculate the Length Using the Table or Formula? . . . 39

How Do You Get ESSL to Calculate the Length Using ESSL Error Handling? . 39
Getting the Best Accuracy 44
What Precisions Do ESSL Subroutines Operate On? 44
How Does the Nature of the ESSL Computation Affect Accuracy? 44
What Does ESSL Do to Improve Accuracy? 44
What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About? 44
What Does ESSL Do to Improve Accuracy? 45
How is Underflow Handled? 45
Where Can You Find More Information on Accuracy? 45
Getting the Best Performance 45
What General Coding Techniques Can You Use to Improve Performance? 45
Where Can You Find More Information on Performance? 46
Dealing with Errors when Using ESSL 47
What Can You Do about Program Exceptions? 47
What Can You Do about ESSL Input-Argument Errors? 47
What Can You Do about ESSL Computational Errors? 48
What Can You Do about ESSL Resource Errors? 50
What Can You Do about ESSL Attention Messages? 51
How Do You Control Error Handling by Setting Values in the ESSL Error Option Table? 51
How does Error Handling Work in a Threaded Environment? 53
Where Can You Find More Information on Errors? 54

Chapter 3. Setting Up Your Data Structures 55
Concepts 55
Vectors 55
Transpose of a Vector 56
Conjugate Transpose of a Vector 57
In Storage 57
How Stride Is Used for Vectors 58
Sparse Vector 61
Matrices 62
Transpose of a Matrix 62
Conjugate Transpose of a Matrix 63
In Storage 63
How Leading Dimension Is Used for Matrices 64
Symmetric Matrix 65
Positive Definite or Negative Definite Symmetric Matrix 69
Complex Hermitian Matrix 70
Positive Definite or Negative Definite Complex Hermitian Matrix 71
Positive Definite or Negative Definite Symmetric Toeplitz Matrix 71
Positive Definite or Negative Definite Complex Hermitian Toeplitz Matrix 72
Triangular Matrix 74
General Band Matrix 77
Symmetric Band Matrix 82
Positive Definite Symmetric Band Matrix 85
Complex Hermitian Band Matrix 85
Triangular Band Matrix 86
General Tridiagonal Matrix 90
Symmetric Tridiagonal Matrix 91
Positive Definite Symmetric Tridiagonal Matrix 92
Sparse Matrix 92
Sequences 105

Real and Complex Elements in Storage . . . 105
One-Dimensional Sequences . . . 105
Two-Dimensional Sequences . . . 105
Three-Dimensional Sequences . . . 106
How Stride Is Used for Three-Dimensional Sequences . . . 107

Chapter 4. Coding Your Program . . . 111
Fortran Programs . . . 111
Calling ESSL Subroutines and Functions in Fortran . . . 111
Setting Up a User-Supplied Subroutine for ESSL in Fortran . . . 111
Setting Up Scalar Data in Fortran . . . 112
Setting Up Arrays in Fortran . . . 112
Creating Multiple Threads and Calling ESSL from Your Fortran Program . . . 117
Handling Errors in Your Fortran Program . . . 118
Example of Handling Errors in a Multithreaded Application Program . . . 127
C Programs . . . 130
Calling ESSL Subroutines and Functions in C . . . 130
Passing Arguments in C . . . 131
Setting Up a User-Supplied Subroutine for ESSL in C . . . 132
Setting Up Scalar Data in C . . . 132
Setting Up Complex and Logical Data Types in C . . . 133
Setting Up Arrays in C . . . 134
Creating Multiple Threads and Calling ESSL from Your C Program . . . 134
Handling Errors in Your C Program . . . 136
C++ Programs . . . 146
Calling ESSL Subroutines and Functions in C++ . . . 146
Passing Arguments in C++ . . . 147
Setting Up a User-Supplied Subroutine for ESSL in C++ . . . 147
Setting Up Scalar Data in C++ . . . 147
Setting Up Short-Precision Complex Data Types and Logical Data Types in C++ 148
Setting Up Arrays in C++ . . . 150
Creating Multiple Threads and Calling ESSL from Your C++ Program . . . 150
Handling Errors in Your C++ Program . . . 152
PL/I Programs . . . 162

Chapter 5. Processing Your Program . . . 163
Compiling . . . 163
General Procedures . . . 163
Using Your Own Complex Data Definitions in C Programs . . . 163
Using Your Own Short Complex Data Definitions in C++ Programs . . . 164
Compiling and Linking . . . 164
Fortran Programs . . . 164
C Programs . . . 165
C++ Programs . . . 166

Chapter 6. Migrating Your Programs . . . 169
Migrating ESSL Version 2 Programs to Version 3 . . . 169
ESSL Subroutines . . . 169
ESSL Messages . . . 169
Planning for Future Migration . . . 170
Migrating between RS/6000 Processors . . . 170
Auxiliary Storage . . . 170
Bitwise-Identical Results . . . 170
Migrating from Other Libraries to ESSL . . . 170

Migrating from ESSL/370 . . . 171
Migrating from Another IBM Subroutine Library . . . 171
Migrating from LAPACK . . . 171
Migrating from a Non-IBM Subroutine Library . . . 171

Chapter 7. Handling Problems . . . 173
Where to Find More Information About Errors . . . 173
Getting Help from IBM Support . . . 173
National Language Support . . . 174
Dealing with Errors . . . 174
Program Exceptions . . . 175
ESSL Input-Argument Error Messages . . . 175
ESSL Computational Error Messages . . . 175
ESSL Resource Error Messages . . . 176
ESSL Informational and Attention Messages . . . 177
Miscellaneous Error Messages . . . 178
Messages . . . 178
Message Conventions . . . 178
Input-Argument Error Messages . . . 179
Computational Error Messages . . . 187
Resource Error Messages . . . 190
Informational and Attention Error Messages . . . 190
Miscellaneous Error Messages . . . 191

Part 2. Reference Information . . . 193

Chapter 8. Linear Algebra Subprograms . . . 195
Overview of the Linear Algebra Subprograms . . . 195
Vector-Scalar Linear Algebra Subprograms . . . 195
Sparse Vector-Scalar Linear Algebra Subprograms . . . 197
Matrix-Vector Linear Algebra Subprograms . . . 197
Sparse Matrix-Vector Linear Algebra Subprograms . . . 198
Use Considerations . . . 199
Performance and Accuracy Considerations . . . 199
Vector-Scalar Subprograms . . . 201
ISAMAX, IDAMAX, ICAMAX, and IZAMAX—Position of the First or Last Occurrence of the Vector Element Having the Largest Magnitude . . . 202
ISAMIN and IDAMIN—Position of the First or Last Occurrence of the Vector Element Having Minimum Absolute Value . . . 205
ISMAX and IDMAX—Position of the First or Last Occurrence of the Vector Element Having the Maximum Value . . . 208
ISMIN and IDMIN—Position of the First or Last Occurrence of the Vector Element Having Minimum Value . . . 211
SASUM, DASUM, SCASUM, and DZASUM—Sum of the Magnitudes of the Elements in a Vector . . . 214
SAXPY, DAXPY, CAXPY, and ZAXPY—Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in the Vector Y . . . 217
SCOPY, DCOPY, CCOPY, and ZCOPY—Copy a Vector . . . 220
SDOT, DDOT, CDOTU, ZDOTU, CDOTC, and ZDOTC—Dot Product of Two Vectors . . . 223
SNAXPY and DNAXPY—Compute SAXPY or DAXPY N Times . . . 227
SNDOT and DNDOT—Compute Special Dot Products N Times . . . 232

SNRM2, DNRM2, SCNRM2, and DZNRM2—Euclidean Length of a Vector with Scaling of Input to Avoid Destructive Underflow and Overflow 237
SNORM2, DNORM2, CNORM2, and ZNORM2—Euclidean Length of a Vector with No Scaling of Input 240
SROTG, DROTG, CROTG, and ZROTG—Construct a Givens Plane Rotation 243
SROT, DROT, CROT, ZROT, CSROT, and ZDROT—Apply a Plane Rotation 250
SSCAL, DSCAL, CSCAL, ZSCAL, CSSCAL, and ZDSCAL—Multiply a Vector X by a Scalar and Store in the Vector X 254
SSWAP, DSWAP, CSWAP, and ZSWAP—Interchange the Elements of Two Vectors 257
SVEA, DVEA, CVEA, and ZVEA—Add a Vector X to a Vector Y and Store in a Vector Z 260
SVES, DVES, CVES, and ZVES—Subtract a Vector Y from a Vector X and Store in a Vector Z 264
SVEM, DVEM, CVEM, and ZVEM—Multiply a Vector X by a Vector Y and Store in a Vector Z 268
SYAX, DYAX, CYAX, ZYAX, CSYAX, and ZDYAX—Multiply a Vector X by a Scalar and Store in a Vector Y 272
SZAXPY, DZAXPY, CZAXPY, and ZZAXPY—Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in a Vector Z 275
Sparse Vector-Scalar Subprograms 279
SSCTR, DSCTR, CSCTR, ZSCTR—Scatter the Elements of a Sparse Vector X in Compressed-Vector Storage Mode into Specified Elements of a Sparse Vector Y in Full-Vector Storage Mode 280
SGTHR, DGTHR, CGTHR, and ZGTHR—Gather Specified Elements of a Sparse Vector Y in Full-Vector Storage Mode into a Sparse Vector X in Compressed-Vector Storage Mode 283
SGTHRZ, DGTHRZ, CGTHRZ, and ZGTHRZ—Gather Specified Elements of a Sparse Vector Y in Full-Vector Mode into a Sparse Vector X in Compressed-Vector Mode, and Zero the Same Specified Elements of Y . . 286
SAXPYI, DAXPYI, CAXPYI, and ZAXPYI—Multiply a Sparse Vector X in Compressed-Vector Storage Mode by a Scalar, Add to a Sparse Vector Y in Full-Vector Storage Mode, and Store in the Vector Y 289
SDOTI, DDOTI, CDOTUI, ZDOTUI, CDOTCI, and ZDOTCI—Dot Product of a Sparse Vector X in Compressed-Vector Storage Mode and a Sparse Vector Y in Full-Vector Storage Mode 292
Matrix-Vector Subprograms 296
SGEMV, DGEMV, CGEMV, ZGEMV, SGEMX, DGEMX, SGEMTX, and DGEMTX—Matrix-Vector Product for a General Matrix, Its Transpose, or Its Conjugate Transpose 297
SGER, DGER, CGERU, ZGERU, CGERC, and ZGERC—Rank-One Update of a General Matrix 308
SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV, SSLMX, and DSLMX—Matrix-Vector Product for a Real Symmetric or Complex Hermitian Matrix 316
SSPR, DSPR, CHPR, ZHPR, SSYR, DSYR, CHER, ZHER, SSLR1, and DSLR1 —Rank-One Update of a Real Symmetric or Complex Hermitian Matrix 324
SSPR2, DSPR2, CHPR2, ZHPR2, SSYR2, DSYR2, CHER2, ZHER2, SSLR2, and DSLR2—Rank-Two Update of a Real Symmetric or Complex Hermitian Matrix 332
SGBMV, DGBMV, CGBMV, and ZGBMV—Matrix-Vector Product for a General Band Matrix, Its Transpose, or Its Conjugate Transpose 341

SSBMV, DSBMV, CHBMV, and ZHBMV—Matrix-Vector Product for a Real Symmetric or Complex Hermitian Band Matrix . . . 348
STRMV, DTRMV, CTRMV, ZTRMV, STPMV, DTPMV, CTPMV, and ZTPMV—Matrix-Vector Product for a Triangular Matrix, Its Transpose, or Its Conjugate Transpose . . . 353
STBMV, DTBMV, CTBMV, and ZTBMV—Matrix-Vector Product for a Triangular Band Matrix, Its Transpose, or Its Conjugate Transpose . . . 359
Sparse Matrix-Vector Subprograms . . . 365
DSMMX—Matrix-Vector Product for a Sparse Matrix in Compressed-Matrix Storage Mode . . . 366
DSMTM—Transpose a Sparse Matrix in Compressed-Matrix Storage Mode . 369
DSDMX—Matrix-Vector Product for a Sparse Matrix or Its Transpose in Compressed-Diagonal Storage Mode . . . 373

Chapter 9. Matrix Operations . . . 379
Overview of the Matrix Operation Subroutines . . . 379
Use Considerations . . . 380
Specifying Normal, Transposed, or Conjugate Transposed Input Matrices . . . 380
Transposing or Conjugate Transposing: . . . 381
Performance and Accuracy Considerations . . . 381
In General . . . 381
For Large Matrices . . . 381
For Combined Operations . . . 382
Matrix Operation Subroutines . . . 383
SGEADD, DGEADD, CGEADD, and ZGEADD—Matrix Addition for General Matrices or Their Transposes . . . 384
SGESUB, DGESUB, CGESUB, and ZGESUB—Matrix Subtraction for General Matrices or Their Transposes . . . 391
SGEMUL, DGEMUL, CGEMUL, and ZGEMUL—Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes . . . 398
SGEMMS, DGEMMS, CGEMMS, and ZGEMMS—Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes Using Winograd's Variation of Strassen's Algorithm . . . 408
SGEMM, DGEMM, CGEMM, and ZGEMM—Combined Matrix Multiplication and Addition for General Matrices, Their Transposes, or Conjugate Transposes 413
SSYMM, DSYMM, CSYMM, ZSYMM, CHEMM, and ZHEMM—Matrix-Matrix Product Where One Matrix is Real or Complex Symmetric or Complex Hermitian . . . 422
STRMM, DTRMM, CTRMM, and ZTRMM—Triangular Matrix-Matrix Product 430
SSYRK, DSYRK, CSYRK, ZSYRK, CHERK, and ZHERK—Rank-K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix . . . 437
SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, and ZHER2K—Rank-2K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix . 444
SGETMI, DGETMI, CGETMI, and ZGETMI—General Matrix Transpose (In-Place) . . . 452
SGETMO, DGETMO, CGETMO, and ZGETMO—General Matrix Transpose (Out-of-Place) . . . 455

Chapter 10. Linear Algebraic Equations . . . 459
Overview of the Linear Algebraic Equation Subroutines . . . 459
Dense Linear Algebraic Equation Subroutines . . . 459
Banded Linear Algebraic Equation Subroutines . . . 460
Sparse Linear Algebraic Equation Subroutines . . . 461
Linear Least Squares Subroutines . . . 462

Dense and Banded Linear Algebraic Equation Considerations 462
Use Considerations 462
Performance and Accuracy Considerations 463
Sparse Matrix Direct Solver Considerations 463
Use Considerations 463
Performance and Accuracy Considerations 463
Sparse Matrix Skyline Solver Considerations 464
Use Considerations 464
Performance and Accuracy Considerations 465
Sparse Matrix Iterative Solver Considerations 465
Use Considerations 465
Performance and Accuracy Considerations 465
Linear Least Squares Considerations 466
Use Considerations 466
Performance and Accuracy Considerations 466
Dense Linear Algebraic Equation Subroutines 467
SGEF, DGEF, CGEF, and ZGEF—General Matrix Factorization 468
SGES, DGES, CGES, and ZGES—General Matrix, Its Transpose, or Its Conjugate Transpose Solve 471
SGESM, DGESM, CGESM, and ZGESM—General Matrix, Its Transpose, or Its Conjugate Transpose Multiple Right-Hand Side Solve 475
SGETRF, DGETRF, CGETRF and ZGETRF—General Matrix Factorization . 481
SGETRS, DGETRS, CGETRS, and ZGETRS—General Matrix Multiple Right-Hand Side Solve 485
SGEFCD and DGEFCD—General Matrix Factorization, Condition Number Reciprocal, and Determinant 490
SPPF, DPPF, SPOF, DPOF, CPOF, and ZPOF—Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization 494
SPPS and DPPS—Positive Definite Real Symmetric Matrix Solve 502
SPOSM, DPOSM, CPOSM, and ZPOSM—Positive Definite Real Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve 505
SPPFCD, DPPFCD, SPOFCD, and DPOFCD—Positive Definite Real Symmetric Matrix Factorization, Condition Number Reciprocal, and Determinant 510
SGEICD and DGEICD—General Matrix Inverse, Condition Number Reciprocal, and Determinant 516
SPPICD, DPPICD, SPOICD, and DPOICD—Positive Definite Real Symmetric Matrix Inverse, Condition Number Reciprocal, and Determinant 521
STRSV, DTRSV, CTRSV, ZTRSV, STPSV, DTPSV, CTPSV, and ZTPSV—Solution of a Triangular System of Equations with a Single Right-Hand Side 528
STRSM, DTRSM, CTRSM, and ZTRSM—Solution of Triangular Systems of Equations with Multiple Right-Hand Sides 534
STRI, DTRI, STPI, and DTPI—Triangular Matrix Inverse 542
Banded Linear Algebraic Equation Subroutines 547
SGBF and DGBF—General Band Matrix Factorization 548
SGBS and DGBS—General Band Matrix Solve 552
SPBF, DPBF, SPBCHF, and DPBCHF—Positive Definite Symmetric Band Matrix Factorization 555
SPBS, DPBS, SPBCHS, and DPBCHS—Positive Definite Symmetric Band Matrix Solve 559
SGTF and DGTF—General Tridiagonal Matrix Factorization 562
SGTS and DGTS—General Tridiagonal Matrix Solve 565

SGTNP, DGTNP, CGTNP, and ZGTNP—General Tridiagonal Matrix Combined Factorization and Solve with No Pivoting . . . 567
SGTNPF, DGTNPF, CGTNPF, and ZGTNPF—General Tridiagonal Matrix Factorization with No Pivoting . . . 570
SGTNPS, DGTNPS, CGTNPS, and ZGTNPS—General Tridiagonal Matrix Solve with No Pivoting . . . 573
SPTF and DPTF—Positive Definite Symmetric Tridiagonal Matrix Factorization 576
SPTS and DPTS—Positive Definite Symmetric Tridiagonal Matrix Solve . . 578
STBSV, DTBSV, CTBSV, and ZTBSV—Triangular Band Equation Solve . . 580
Sparse Linear Algebraic Equation Subroutines . . . 586
DGSF—General Sparse Matrix Factorization Using Storage by Indices, Rows, or Columns . . . 587
DGSS—General Sparse Matrix or Its Transpose Solve Using Storage by Indices, Rows, or Columns . . . 593
DGKFS—General Sparse Matrix or Its Transpose Factorization, Determinant, and Solve Using Skyline Storage Mode . . . 597
DSKFS—Symmetric Sparse Matrix Factorization, Determinant, and Solve Using Skyline Storage Mode . . . 615
DSRIS—Iterative Linear System Solver for a General or Symmetric Sparse Matrix Stored by Rows . . . 634
DSMCG—Sparse Positive Definite or Negative Definite Symmetric Matrix Iterative Solve Using Compressed-Matrix Storage Mode . . . 645
DSDCG—Sparse Positive Definite or Negative Definite Symmetric Matrix Iterative Solve Using Compressed-Diagonal Storage Mode . . . 653
DSMGCG—General Sparse Matrix Iterative Solve Using Compressed-Matrix Storage Mode . . . 661
DSDGCG—General Sparse Matrix Iterative Solve Using Compressed-Diagonal Storage Mode . . . 668
Linear Least Squares Subroutines . . . 675
SGESVF and DGESVF—Singular Value Decomposition for a General Matrix 676
SGESVS and DGESVS—Linear Least Squares Solution for a General Matrix Using the Singular Value Decomposition . . . 684
SGELLS and DGELLS—Linear Least Squares Solution for a General Matrix Using a QR Decomposition with Column Pivoting . . . 689

Chapter 11. Eigensystem Analysis . . . 695
Overview of the Eigensystem Analysis Subroutines . . . 695
Performance and Accuracy Considerations . . . 695
Eigensystem Analysis Subroutines . . . 697
SGEEV, DGEEV, CGEEV, and ZGEEV—Eigenvalues and, Optionally, All or Selected Eigenvectors of a General Matrix . . . 698
SSPEV, DSPEV, CHPEV, and ZHPEV—Eigenvalues and, Optionally, the Eigenvectors of a Real Symmetric Matrix or a Complex Hermitian Matrix . 710
SSPSV, DSPSV, CHPSV, and ZHPSV—Extreme Eigenvalues and, Optionally, the Eigenvectors of a Real Symmetric Matrix or a Complex Hermitian Matrix 719
SGEGV and DGEGV—Eigenvalues and, Optionally, the Eigenvectors of a Generalized Real Eigensystem, Az=wBz, where A and B Are Real General Matrices . . . 727
SSYGV and DSYGV—Eigenvalues and, Optionally, the Eigenvectors of a Generalized Real Symmetric Eigensystem, Az=wBz, where A Is Real Symmetric and B Is Real Symmetric Positive Definite . . . 733

Chapter 12. Fourier Transforms, Convolutions and Correlations, and Related Computations . . . 739

Overview of the Signal Processing Subroutines . . . 739
Fourier Transforms Subroutines . . . 739
Convolution and Correlation Subroutines . . . 740
Related-Computation Subroutines . . . 740
Fourier Transforms, Convolutions, and Correlations Considerations . . . 741
Use Considerations . . . 741
Initializing Auxiliary Working Storage . . . 743
Determining the Amount of Auxiliary Working Storage That You Need . . . 743
Performance and Accuracy Considerations . . . 743
When Running on the Workstation Processors . . . 744
Defining Arrays . . . 744
Fourier Transform Considerations . . . 744
How the Fourier Transform Subroutines Achieve High Performance . . . 745
Convolution and Correlation Considerations . . . 745
Related Computation Considerations . . . 747
Accuracy Considerations . . . 747
Fourier Transform Subroutines . . . 748
SCFT and DCFT—Complex Fourier Transform . . . 749
SRCFT and DRCFT—Real-to-Complex Fourier Transform . . . 758
SCRFT and DCRFT—Complex-to-Real Fourier Transform . . . 766
SCOSF and DCOSF—Cosine Transform . . . 774
SSINF and DSINF—Sine Transform . . . 782
SCFT2 and DCFT2—Complex Fourier Transform in Two Dimensions . . . 790
SRCFT2 and DRCFT2—Real-to-Complex Fourier Transform in Two Dimensions . . . 797
SCRFT2 and DCRFT2—Complex-to-Real Fourier Transform in Two Dimensions . . . 804
SCFT3 and DCFT3—Complex Fourier Transform in Three Dimensions . . . 812
SRCFT3 and DRCFT3—Real-to-Complex Fourier Transform in Three Dimensions . . . 818
SCRFT3 and DCRFT3—Complex-to-Real Fourier Transform in Three Dimensions . . . 824
Convolution and Correlation Subroutines . . . 830
SCON and SCOR—Convolution or Correlation of One Sequence with One or More Sequences . . . 831
SCOND and SCORD—Convolution or Correlation of One Sequence with Another Sequence Using a Direct Method . . . 837
SCONF and SCORF—Convolution or Correlation of One Sequence with One or More Sequences Using the Mixed-Radix Fourier Method . . . 843
SDCON, DDCON, SDCOR, and DDCOR—Convolution or Correlation with Decimated Output Using a Direct Method . . . 852
SACOR—Autocorrelation of One or More Sequences . . . 856
SACORF—Autocorrelation of One or More Sequences Using the Mixed-Radix Fourier Method . . . 860
Related-Computation Subroutines . . . 866
SPOLY and DPOLY—Polynomial Evaluation . . . 867
SIZC and DIZC—I-th Zero Crossing . . . 870
STREC and DTREC—Time-Varying Recursive Filter . . . 873
SQINT and DQINT—Quadratic Interpolation . . . 876
SWLEV, DWLEV, CWLEV, and ZWLEV—Wiener-Levinson Filter Coefficients . . . 880

Chapter 13. Sorting and Searching . . . 885
Overview of the Sorting and Searching Subroutines . . . 885

Use Considerations . . . 885
Performance and Accuracy Considerations . . . 885
Sorting and Searching Subroutines . . . 887
ISORT, SSORT, and DSORT—Sort the Elements of a Sequence . . . 888
ISORTX, SSORTX, and DSORTX—Sort the Elements of a Sequence and Note the Original Element Positions . . . 890
ISORTS, SSORTS, and DSORTS—Sort the Elements of a Sequence Using a Stable Sort and Note the Original Element Positions . . . 893
IBSRCH, SBSRCH, and DBSRCH—Binary Search for Elements of a Sequence X in a Sorted Sequence Y . . . 896
ISSRCH, SSSRCH, and DSSRCH—Sequential Search for Elements of a Sequence X in the Sequence Y . . . 900

Chapter 14. Interpolation . . . 905
Overview of the Interpolation Subroutines . . . 905
Use Considerations . . . 905
Performance and Accuracy Considerations . . . 905
Interpolation Subroutines . . . 906
SPINT and DPINT—Polynomial Interpolation . . . 907
STPINT and DTPINT—Local Polynomial Interpolation . . . 912
SCSINT and DCSINT—Cubic Spline Interpolation . . . 915
SCSIN2 and DCSIN2—Two-Dimensional Cubic Spline Interpolation . . . 921

Chapter 15. Numerical Quadrature . . . 925
Overview of the Numerical Quadrature Subroutines . . . 925
Use Considerations . . . 925
Choosing the Method . . . 925
Performance and Accuracy Considerations . . . 925
Programming Considerations for the SUBF Subroutine . . . 926
Designing SUBF . . . 926
Coding and Setting Up SUBF in Your Program . . . 927
Numerical Quadrature Subroutines . . . 928
SPTNQ and DPTNQ—Numerical Quadrature Performed on a Set of Points . 929
SGLNQ and DGLNQ—Numerical Quadrature Performed on a Function Using Gauss-Legendre Quadrature . . . 932
SGLNQ2 and DGLNQ2—Numerical Quadrature Performed on a Function Over a Rectangle Using Two-Dimensional Gauss-Legendre Quadrature . . . 935
SGLGQ and DGLGQ—Numerical Quadrature Performed on a Function Using Gauss-Laguerre Quadrature . . . 941
SGRAQ and DGRAQ—Numerical Quadrature Performed on a Function Using Gauss-Rational Quadrature . . . 945
SGHMQ and DGHMQ—Numerical Quadrature Performed on a Function Using Gauss-Hermite Quadrature . . . 949

Chapter 16. Random Number Generation . . . 953
Overview of the Random Number Generation Subroutines . . . 953
Use Considerations . . . 953
Random Number Generation Subroutines . . . 953
SURAND and DURAND—Generate a Vector of Uniformly Distributed Random Numbers . . . 954
SNRAND and DNRAND—Generate a Vector of Normally Distributed Random Numbers . . . 957
SURXOR and DURXOR—Generate a Vector of Long Period Uniformly Distributed Random Numbers . . . 961

Chapter 17. Utilities . . . 965
Overview of the Utility Subroutines . . . 965
Use Considerations . . . 965
Determining the Level of ESSL Installed . . . 965
Finding the Optimal Stride(s) for Your Fourier Transforms . . . 966
Converting Sparse Matrix Storage . . . 966
Utility Subroutines . . . 967
EINFO—ESSL Error Information-Handler Subroutine . . . 968
ERRSAV—ESSL ERRSAV Subroutine for ESSL . . . 971
ERRSET—ESSL ERRSET Subroutine for ESSL . . . 972
ERRSTR—ESSL ERRSTR Subroutine for ESSL . . . 974
IESSL—Determine the Level of ESSL Installed . . . 975
STRIDE—Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines . . . 977
DSRSM—Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode . . . 986
DGKTRN—For a General Sparse Matrix, Convert Between Diagonal-Out and Profile-In Skyline Storage Mode . . . 990
DSKTRN—For a Symmetric Sparse Matrix, Convert Between Diagonal-Out and Profile-In Skyline Storage Mode . . . 996

Part 3. Appendixes . . . 1001

Appendix A. Basic Linear Algebra Subprograms (BLAS) . . . APA-1
Level 1 BLAS . . . APA-1
Level 2 BLAS . . . APA-2
Level 3 BLAS . . . APA-3

Appendix B. LAPACK . . . APB-1
LAPACK . . . APB-1

Glossary . . . GLOS-1

Bibliography . . . BIB-1
References . . . BIB-1
ESSL Publications . . . BIB-4
Evaluation and Planning . . . BIB-4
Installation . . . BIB-5
Application Programming . . . BIB-5
Related Publications . . . BIB-5
AIX Version 4 for the RS/6000 . . . BIB-5
XL Fortran . . . BIB-5
XL C and C++ . . . BIB-5
PL/I . . . BIB-5
Workstation Processors . . . BIB-5
IBM 3838 Array Processor . . . BIB-5

Index . . . INDEX-1

Looking for a Subroutine?

CAXPY... 217
CAXPYI... 289
CCOPY... 220
CDOTC... 223
CDOTCI... 292
CDOTU... 223
CDOTUI... 292
CGBMV... 341
CGEADD... 384
CGEEV... 698
CGEF... 468
CGEMM... 413
CGEMMS... 408
CGEMUL... 398
CGEMV... 297
CGERC... 308
CGERU... 308
CGES... 471
CGESM... 475
CGESUB... 391
CGETMI... 452
CGETMO... 455
CGETRF... 481
CGETRS... 485
CGTHR... 283
CGTHRZ... 286
CGTNP... 567
CGTNPF... 570
CGTNPS... 573
CHBMV... 348
CHEMM... 422
CHEMV... 316
CHER... 324
CHER2... 332
CHER2K... 444
CHERK... 437
CHLEV... 710
CHPEV... 710
CHPMV... 316
CHPR... 324
CHPR2... 332
CHPSV... 719
CNORM2... 240
CPOF... 494
CPOSM... 505
CROT... 250
CROTG... 243
CSCAL... 254
CSCTR... 280
CSROT... 250
CSSCAL... 254
CSWAP... 257
CSYAX... 272
CSYMM... 422
CSYR2K... 444
CSYRK... 437
CTBMV... 359
CTBSV... 580
CTPMV... 353
CTPSV... 528
CTRMM... 430
CTRMV... 353
CTRSM... 534
CTRSV... 528
CVEA... 260
CVEM... 268
CVES... 264
CWLEV... 880
CYAX... 272
CZAXPY... 275
DASUM... 214
DAXPY... 217
DAXPYI... 289
DBSRCH... 896
DCFT... 749
DCOSF... 774
DCFT2... 790
DCFT3... 812
DCOPY... 220
DCRFT... 766
DCRFT2... 804
DCRFT3... 824
DCSIN2... 921
DCSINT... 915
DDCON... 852
DDCOR... 852
DDOT... 223
DDOTI... 292
DGBF... 548
DGBMV... 341
DGBS... 552
DGEADD... 384
DGEEV... 698
DGEF... 468
DGEFCD... 490
DGEGV... 727
DGEICD... 516
DGELLS... 689
DGEMM... 413
DGEMMS... 408
DGEMTX... 297
DGEMUL... 398
DGEMV... 297
DGEMX... 297
DGER... 308
DGER1... 308
DGES... 471
DGESM... 475
DGESUB... 391
DGESVF... 676
DGESVS... 684
DGETMI... 452
DGETMO... 455
DGETRF... 481
DGETRS... 485
DGHMQ... 949
DGKFS... 597
DGKTRN... 990
DGLGQ... 941
DGLNQ... 932
DGLNQ2... 935
DGRAQ... 945
DGSF... 587
DGSS... 593
DGTF... 562
DGTHR... 283
DGTHRZ... 286
DGTNP... 567
DGTNPF... 570
DGTNPS... 573
DGTS... 565
DIZC... 870
DNAXPY... 227
DNDOT... 232
DNORM2... 240
DNRAND... 957
DNRM2... 237
DPBCHF... 555
DPBCHS... 559
DPBF... 555
DPBS... 559
DPINT... 907
DPOF... 494
DPOFCD... 510
DPOICD... 521
DPOLY... 867
DPOSM... 505
DPPF... 494
DPPFCD... 510
DPPICD... 521
DPPS... 502
DPTF... 576
DPTNQ... 929
DPTS... 578
DQINT... 876
DRCFT... 758
DRCFT2... 797
DRCFT3... 818
DROT... 250
DROTG... 243
DSBMV... 348
DSCAL... 254
DSCTR... 280
DSDCG... 653
DSDGCG... 668
DSDMX... 373
DSINF... 782
DSKFS... 615
DSKTRN... 996
DSLEV... 710
DSLMX... 316
DSLR1... 324
DSLR2... 332
DSMCG... 645
DSMGCG... 661
DSMMX... 366
DSMTM... 369
DSORT... 888
DSORTS... 893
DSORTX... 890
DSPEV... 710
DSPMV... 316
DSPR... 324
DSPR2... 332
DSPSV... 719
DSRIS... 634
DSRSM... 986
DSSRCH... 900
DSWAP... 257
DSYGV... 733
DSYMM... 422
DSYMV... 316
DSYR... 324
DSYR2... 332
DSYR2K... 444
DSYRK... 437
DTBMV... 359
DTBSV... 580
DTPI... 542
DTPINT... 912
DTPMV... 353
DTPSV... 528
DTREC... 873
DTRI... 542
DTRMM... 430
DTRMV... 353
DTRSM... 534
DTRSV... 528
DURAND... 954
DURXOR... 961
DVEA... 260
DVEM... 268
DVES... 264
DWLEV... 880
DYAX... 272
DZASUM... 214
DZAXPY... 275
DZNRM2... 237
EINFO... 968
ERRSAV... 971
ERRSET... 972
ERRSTR... 974
IBSRCH... 896
ICAMAX... 202
IDAMAX... 202
IDAMIN... 205
IDMAX... 208
IDMIN... 211
IESSL... 975
ISAMAX... 202
ISAMIN... 205
ISMAX... 208
ISMIN... 211
ISORT... 888
ISORTS... 893
ISORTX... 890
ISSRCH... 900
IZAMAX... 202
SACOR... 856

© Copyright IBM Corp. 1997

SACORF... 860
SASUM... 214
SAXPY... 217
SAXPYI... 289
SBSRCH... 896
SCASUM... 214
SCFT... 749
SCFT2... 790
SCFT3... 812
SCNRM2... 237
SCON... 831
SCOND... 837
SCONF... 843
SCOPY... 220
SCOR... 831
SCORD... 837
SCORF... 843
SCOSF... 774
SCRFT... 766
SCRFT2... 804
SCRFT3... 824
SCSIN2... 921
SCSINT... 915
SDCON... 852
SDCOR... 852
SDOT... 223
SDOTI... 292
SGBF... 548
SGBMV... 341
SGBS... 552
SGEADD... 384
SGEEV... 698
SGEF... 468
SGEFCD... 490
SGEGV... 727
SGEICD... 516
SGELLS... 689
SGEMM... 413
SGEMMS... 408
SGEMTX... 297
SGEMUL... 398
SGEMV... 297
SGEMX... 297
SGER... 308
SGER1... 308
SGES... 471
SGESM... 475
SGESUB... 391
SGESVF... 676
SGESVS... 684
SGETMI... 452
SGETMO... 455
SGETRF... 481
SGETRS... 485
SGHMQ... 949
SGLGQ... 941
SGLNQ... 932
SGLNQ2... 935
SGRAQ... 945
SGTF... 562
SGTHR... 283
SGTHRZ... 286
SGTNP... 567
SGTNPF... 570
SGTNPS... 573
SGTS... 565
SIZC... 870
SNAXPY... 227
SNDOT... 232
SNORM2... 240
SNRAND... 957
SNRM2... 237
SPBCHF... 555
SPBCHS... 559
SPBF... 555
SPBS... 559
SPINT... 907
SPOF... 494
SPOFCD... 510
SPOICD... 521
SPOLY... 867
SPOSM... 505
SPPF... 494
SPPFCD... 510
SPPICD... 521
SPPS... 502
SPTF... 576
SPTNQ... 929
SPTS... 578
SQINT... 876
SRCFT... 758
SRCFT2... 797
SRCFT3... 818
SROT... 250
SROTG... 243
SSBMV... 348
SSCAL... 254
SSCTR... 280
SSINF... 782
SSLEV... 710
SSLMX... 316
SSLR1... 324
SSLR2... 332
SSORT... 888
SSORTS... 893
SSORTX... 890
SSPEV... 710
SSPMV... 316
SSPR... 324
SSPR2... 332
SSPSV... 719
SSSRCH... 900
SSWAP... 257
SSYGV... 733
SSYMM... 422
SSYMV... 316
SSYR... 324
SSYR2... 332
SSYR2K... 444
SSYRK... 437
STBMV... 359
STBSV... 580
STPI... 542
STPINT... 912
STPMV... 353
STPSV... 528
STREC... 873
STRI... 542
STRIDE... 977
STRMM... 430
STRMV... 353
STRSM... 534
STRSV... 528
SURAND... 954
SURXOR... 961
SVEA... 260
SVEM... 268
SVES... 264
SWLEV... 880
SYAX... 272
SZAXPY... 275
ZAXPY... 217
ZAXPYI... 289
ZCOPY... 220
ZDOTC... 223
ZDOTCI... 292
ZDOTU... 223
ZDOTUI... 292
ZDROT... 250
ZDSCAL... 254
ZDYAX... 272
ZGBMV... 341
ZGEADD... 384
ZGEEV... 698
ZGEF... 468
ZGEMM... 413
ZGEMMS... 408
ZGEMUL... 398
ZGEMV... 297
ZGERC... 308
ZGERU... 308
ZGES... 471
ZGESM... 475
ZGESUB... 391
ZGETMI... 452
ZGETMO... 455
ZGETRF... 481
ZGETRS... 485
ZGTHR... 283
ZGTHRZ... 286
ZGTNP... 567
ZGTNPF... 570
ZGTNPS... 573
ZHBMV... 348
ZHEMM... 422
ZHEMV... 316
ZHER... 324
ZHER2... 332
ZHER2K... 444
ZHERK... 437
ZHLEV... 710
ZHPEV... 710
ZHPMV... 316
ZHPR... 324
ZHPR2... 332
ZHPSV... 719
ZNORM2... 240
ZPOF... 494
ZPOSM... 505
ZROT... 250
ZROTG... 243
ZSCAL... 254
ZSCTR... 280
ZSWAP... 257
ZSYMM... 422
ZSYR2K... 444
ZSYRK... 437
ZTBMV... 359
ZTBSV... 580
ZTPMV... 353
ZTPSV... 528
ZTRMM... 430
ZTRMV... 353
ZTRSM... 534
ZTRSV... 528
ZVEA... 260
ZVEM... 268
ZVES... 264
ZWLEV... 880
ZYAX... 272
ZZAXPY... 275

Special Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10694
USA

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
522 South Road
Poughkeepsie, NY 12601-5400
USA
Attention: Information Request

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of fee.

Trademarks

The following terms, denoted by an asterisk (*), are trademarks of the IBM Corporation in the United States or other countries or both:

AIX	PowerPC
C Set ++	POWER2
IBM	RS/6000
Power	SP

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

© Copyright IBM Corp. 1997

Programming Interfaces

This *Engineering and Scientific Subroutine Library (ESSL) Version 3 Guide and Reference* manual is intended to help the customer to do application programming. This *ESSL Version 3 Guide and Reference* manual documents General-use Programming Interface and Associated Guidance Information provided by ESSL Version 3.

General-use programming interfaces allow the customer to write programs that obtain the services of ESSL Version 3.

About This Book

The Engineering and Scientific Subroutine Library (ESSL) for AIX is a set of high-performance mathematical subroutines. ESSL is provided as four run-time libraries, running on RS/6000* POWER*, PowerPC*, and POWER2* processors. ESSL can be used with Fortran, C, C++, and Programming Language/I (PL/I) programs operating under the AIX operating system.

This book is a guide and reference manual for using ESSL in doing application programming. It includes:

- An overview of ESSL and guidance information for designing, coding, and processing your program, as well as migrating existing programs, and diagnosing problems
- Reference information for coding each ESSL calling sequence

This book is written for a wide class of ESSL users: scientists, mathematicians, engineers, statisticians, computer scientists, and system programmers. It assumes a basic knowledge of mathematics in the areas of ESSL computation. It also assumes that users are familiar with Fortran, C, C++, and PL/I programming.

How to Use This Book

Front Matter consists of the Table of Contents, Special Notices, and Preface. Use these to find or interpret information in the book.

Part 1. "Guide Information" provides guidance information for using ESSL. It covers the user-oriented tasks of learning, designing, coding, migrating, processing, and diagnosing. Use the following chapters when performing any of these tasks:

- **Chapter 1, "Learning about ESSL"** gives an introduction to ESSL, providing highlights and general information. Read this chapter first to determine the aspects of ESSL you want to use.
- **Chapter 2, "Designing Your Program"** provides ESSL-specific information that helps you design your program. Read this chapter before designing your program.
- **Chapter 3, "Setting Up Your Data Structures"** describes all types of data structures, such as vectors, matrices, and sequences. Use this information when designing and coding your program.
- **Chapter 4, "Coding Your Program"** tells you how to code your scalar and array data, how to code calls to ESSL in Fortran, C, C++, and PL/I programs, and how to do the coding necessary to handle errors. Use this information when coding your program.
- **Chapter 5, "Processing Your Program"** describes how to process your program under your particular operating system on your hardware. Use this information after you have coded your program and are ready to run it.
- **Chapter 6, "Migrating Your Programs"** explains all aspects of migration to ESSL, to this version of ESSL, to different processors, and to future releases and future processors. Read this chapter before starting to design your program.

© Copyright IBM Corp. 1997

- **Chapter 7, "Handling Problems"** provides diagnostic procedures for analyzing all ESSL problems. When you encounter a problem, use the symptom indexes at the beginning of this chapter to guide you to the appropriate diagnostic procedure.

Part 2. "Reference Information" provides reference information you need to code the ESSL calling sequences. It covers each of the mathematical areas of ESSL, and the utility subroutines. Each chapter begins with an introduction, followed by the subroutine descriptions. Each introduction applies to all the subroutines in that chapter and is especially important in planning your use of the subroutines and avoiding problems. To understand the information in the subroutine descriptions, see "How to Interpret the Subroutine Descriptions" on page xxvii. Use the appropriate chapter when coding your program:

- **Chapter 8, "Linear Algebra Subprograms"**
- **Chapter 9, "Matrix Operations"**
- **Chapter 10, "Linear Algebraic Equations"**
- **Chapter 11, "Eigensystem Analysis"**
- **Chapter 12, "Fourier Transforms, Convolutions and Correlations, and Related Computations"**
- **Chapter 13, "Sorting and Searching"**
- **Chapter 14, "Interpolation"**
- **Chapter 15, "Numerical Quadrature"**
- **Chapter 16, "Random Number Generation"**
- **Chapter 17, "Utilities"**

Appendix A. Basic Linear Algebra Subprograms provides a list of the Level 1, 2, and 3 Basic Linear Algebra Subprograms (BLAS) included in ESSL.

Appendix B. LAPACK provides a list of the LAPACK subroutines included in ESSL.

Glossary contains definitions of terms used in this book.

Bibliography provides information about publications related to ESSL. Use it when you need more information than this book provides.

How to Find a Subroutine Description

If you want to locate a subroutine description and you know the subroutine name, you can use the "Looking for a Subroutine?" on page xv, following the Table of Contents. You can also find them listed individually or under the entry "subroutines, ESSL" in the Index.

Where to Find Related Publications

A list of all ESSL publications, as well as related programming and hardware publications, are listed in the bibliography. Also included is a list of math background publications you may find helpful, along with the necessary information for ordering them from independent sources. See "Bibliography" on page BIB-1.

How to Look Up a Bibliography Reference

Special references are made throughout this book to mathematical background publications and software libraries, available through IBM, publishers, or other companies. All of these are described in detail in the bibliography. A reference to one of these is made by using a bracketed number. The number refers to the item listed under that number in the bibliography. For example, reference [1] cites the first item listed in the bibliography.

Special Terms

Standard data processing and mathematical terms are used in this book. Terminology is generally consistent with that used for Fortran. See the Glossary for definitions of terms used in this book.

Short and Long Precision

Because ESSL can be used with more than one programming language, the terms **short precision** and **long precision** are used in place of the Fortran terms **single precision** and **double precision**.

Subroutines and Subprograms

An ESSL **subroutine** is a named sequence of instructions within the ESSL product library whose execution is invoked by a call. A subroutine can be called in one or more user programs and at one or more times within each program. The ESSL subroutines are referred to as **subprograms** in the area of linear algebra subprograms. The term subprograms is used because it is consistent with the BLAS. Many of the linear algebra subprograms correspond to the BLAS; these are listed in Appendix A on page APA-1.

How to Interpret the Subroutine Names with a Prefix Underscore

A name specified in this book with an underscore (_) prefix, such as _GEMUL, refers to all the versions of the subroutine with that name. To get the entire list of subroutines that name refers to, substitute the first letter for each version of the subroutine. For example, _GEMUL, refers to all versions of the matrix multiplication subroutine: SGEMUL, DGEMUL, CGEMUL, and ZGEMUL. You do **not** use the underscore in coding the names of the ESSL subroutines in your program. You code a complete name, such as SGEMUL. For details about these names, see "The Variety of Mathematical Functions" on page 4.

Abbreviated Names

The abbreviated names used in this book are defined below.

Short Name	Full Name
AIX	Advanced Interactive Executive
BLAS	Basic Linear Algebra Subprograms
ESSL	IBM Engineering and Scientific Subroutine Library

Short Name	Full Name
HTML	Hypertext Markup Language
LAPACK	Linear Algebra Package
PL/I	Programming Language/I
POWER, POWER2, and PowerPC processors	RS/6000 processors
SL MATH	Subroutine Library—Mathematics
SMP	Symmetric Multi-Processing
SSP	Scientific Subroutine Package

Fonts

This book uses a variety of special fonts to distinguish between many mathematical and programming items. These are defined below.

Special Font	Example	Description
Italic with no subscripts	*m*, *inc1x*, *aux*, *iopt*	Calling sequence argument or mathematical variable
Italic with subscripts	x_1, a_{mn}, $x_{j1,j2}$	Element of a vector, matrix, or sequence
Bold italic lowercase	***x***, ***y***, ***z***	Vector or sequence
Bold italic uppercase	***A***, ***B***, ***C***	Matrix
Gothic uppercase	`A`, `B`, `C`, `AGB` `IM=ISMAX(4,X,2)`	Array Fortran statement

Special Notations and Conventions

This section explains the special notations and conventions used in this book to describe various types of data.

Scalar Data

Following are the special notations used in the examples in this book for scalar data items. These notations are used to simplify the examples, and they do not imply usage of any precision. For a definition of scalar data in Fortran, C, C++, and PL/I, see Chapter 4 on page 111.

Data Item	Example	Description
Character item	`'T'`	Character(s) in single quotation marks
Hexadecimal string	`X'97FA00C1'`	String of 4-bit hexadecimal characters
Logical item	`.TRUE.` `.FALSE.`	True or false logical value, as indicated

Data Item	Example	Description
Integer data	`1`	Number with no decimal point
Real data	`1.6`	Number with a decimal point
Complex data	`(1.0,-2.9)`	Real part followed by the imaginary part
Continuation	$1.666\overline{6}$	Continue the last digit (1.6666666... and so forth)

Vectors

A vector is represented as a single row or column of subscripted elements enclosed in square brackets. The subscripts refer to the element positions within the vector:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ . \\ . \\ x_n \end{bmatrix} \qquad \begin{bmatrix} x_1 & x_2 & x_3 & \ldots & x_n \end{bmatrix}$$

For a definition of vector, see “Vectors” on page 55.

Matrices

A matrix is represented as a block of elements enclosed in square brackets. Subscripts refer to the row and column positions, respectively:

$$\begin{bmatrix} a_{11} & . & . & . & a_{1n} \\ . & & & & . \\ . & & & & . \\ . & & & & . \\ a_{m1} & . & . & . & a_{mn} \end{bmatrix}$$

For a definition of matrix, see “Matrices” on page 62.

Sequences

Sequences are used in the areas of sorting, searching, Fourier transforms, convolutions, and correlations. For a definition of sequences, see “Sequences” on page 105.

One-Dimensional Sequences

A one-dimensional sequence is represented as a series of elements enclosed in parentheses. Subscripts refer to the element position within the sequence:

$$(x_1, x_2, x_3, ..., x_n)$$

Two-Dimensional Sequences

A two-dimensional sequence is represented as a series of columns of elements. (They are represented in the same way as a matrix without the square brackets.) Subscripts refer to the element positions within the first and second dimensions, respectively:

$$\begin{matrix} a_{11} & a_{12} & . & . & . & a_{1n} \\ a_{21} & a_{22} & & & & a_{2n} \\ . & . & & & & . \\ . & . & & & & . \\ . & . & & & & . \\ a_{m1} & a_{m2} & . & . & . & a_{mn} \end{matrix}$$

Three-Dimensional Sequences

A three-dimensional sequence is represented as a series of blocks of elements. Subscripts refer to the elements positions within the first, second, and third dimensions, respectively:

$$\begin{matrix} a_{111} & a_{121} & . & . & . & a_{1n1} \\ a_{211} & a_{221} & & & & a_{2n1} \\ . & . & & & & . \\ . & . & & & & . \\ . & . & & & & . \\ a_{m11} & a_{m21} & . & . & . & a_{mn1} \end{matrix} \quad \begin{matrix} a_{112} & a_{122} & . & . & . & a_{1n2} \\ a_{212} & a_{222} & & & & a_{2n2} \\ . & . & & & & . \\ . & . & & & & . \\ . & . & & & & . \\ a_{m12} & a_{m22} & . & . & . & a_{mn2} \end{matrix} \quad ... \quad \begin{matrix} a_{11p} & a_{12p} & . & . & . & a_{1np} \\ a_{21p} & a_{22p} & & & & a_{2np} \\ . & . & & & & . \\ . & . & & & & . \\ . & . & & & & . \\ a_{m1p} & a_{m2p} & . & . & . & a_{mnp} \end{matrix}$$

Arrays

Arrays contain vectors, matrices, or sequences. For a definition of array, see “How Do You Set Up Your Arrays?” on page 29.

One-Dimensional Arrays

A one-dimensional array is represented as a single row of numeric elements enclosed in parentheses:

```
(1.0, 2.0, 3.0, 4.0, 5.0)
```

Elements not significant to the computation are usually not shown in the array. One dot appears for each element not shown. In the following array, five elements are significant to the computation, and two elements not used in the computation exist between each of the elements shown:

```
(1.0, . , . ,2.0, . , . ,3.0, . , . ,4.0, . , . ,5.0)
```

This notation is used to show vector elements inside an array.

Two-Dimensional Arrays

A two-dimensional array is represented as a block of numeric elements enclosed in square brackets:

```
⎡                    ⎤
⎢ 1.0  11.0  5.0  25.0 ⎥
⎢ 2.0  12.0  6.0  26.0 ⎥
⎢ 3.0  13.0  7.0  27.0 ⎥
⎢ 4.0  14.0  8.0  28.0 ⎥
⎣                    ⎦
```

Elements not significant to the computation are usually not shown in the array. One dot appears for each element not shown. The following array contains three rows and two columns not used in the computation:

```
⎡                        ⎤
⎢ .    .   .    .    .   . ⎥
⎢ .    .   .    .    .   . ⎥
⎢ .   1.0  2.0  5.0  4.0  . ⎥
⎢ .   2.0  3.0  6.0  3.0  . ⎥
⎢ .   3.0  4.0  7.0  2.0  . ⎥
⎢ .   4.0  5.0  8.0  1.0  . ⎥
⎢ .    .    .    .    .   . ⎥
⎣                        ⎦
```

This notation is used to show matrix elements inside an array.

Three-Dimensional Arrays

A three-dimensional array is represented as a series of blocks of elements separated by ellipses. Each block appears like a two-dimensional array:

```
⎡                    ⎤   ⎡                      ⎤       ⎡                      ⎤
⎢ 1.0  11.0  5.0  25.0 ⎥   ⎢ 10.0  111.0  15.0  125.0 ⎥       ⎢ 100.0  11.0  15.0  25.0 ⎥
⎢ 2.0  12.0  6.0  26.0 ⎥   ⎢ 20.0  112.0  16.0  126.0 ⎥  ...  ⎢ 200.0  12.0  16.0  26.0 ⎥
⎢ 3.0  13.0  7.0  27.0 ⎥   ⎢ 30.0  113.0  17.0  127.0 ⎥       ⎢ 300.0  13.0  17.0  27.0 ⎥
⎢ 4.0  14.0  8.0  28.0 ⎥   ⎢ 40.0  114.0  18.0  128.0 ⎥       ⎢ 400.0  14.0  18.0  28.0 ⎥
⎣                    ⎦   ⎣                      ⎦       ⎣                      ⎦
```

Elements not significant to the computation are usually not shown in the array. One dot appears for each element not shown, just as for two-dimensional arrays.

Special Characters, Symbols, Expressions, and Abbreviations

The mathematical and programming notations used in this book are consistent with traditional mathematical and programming usage. These conventions are explained below, along with special abbreviations that are associated with specific values.

Item	Description
Greek letters: α, σ, ω, Ω	Symbolic scalar values
$\|a\|$	The absolute value of a
$\boldsymbol{a} \bullet \boldsymbol{b}$	The dot product of $\boldsymbol{a}$ and $\boldsymbol{b}$
x_i	The i-th element of vector $\boldsymbol{x}$
c_{ij}	The element in matrix $\boldsymbol{C}$ at row i and column j
$x_1 \ldots x_n$	Elements from x_1 to x_n
$i = 1, n$	i is assigned the values 1 to n
$\boldsymbol{y} \leftarrow \boldsymbol{x}$	Vector $\boldsymbol{y}$ is replaced by vector $\boldsymbol{x}$
$\boldsymbol{xy}$	Vector $\boldsymbol{x}$ times vector $\boldsymbol{y}$
$\boldsymbol{AX} \cong \boldsymbol{B}$	$\boldsymbol{AX}$ is congruent to $\boldsymbol{B}$
a^k	a raised to the k power
e^x	Exponential function of x
$\boldsymbol{A}^{\mathrm{T}}$; $\boldsymbol{x}^{\mathrm{T}}$	The transpose of matrix $\boldsymbol{A}$; the transpose of vector $\boldsymbol{x}$
$\overline{\boldsymbol{x}}$; $\overline{\boldsymbol{A}}$	The complex conjugate of vector $\boldsymbol{x}$; the complex conjugate of matrix $\boldsymbol{A}$
$\overline{x}_i$; $\overline{c}_{jk}$	The complex conjugate of the complex vector element x_i, where: $$\text{if } x_i = (a_i, b_i), \quad \text{then } \overline{x}_i = (a_i, -b_i)$$ The complex conjugate of the complex matrix element c_{jk}
$\boldsymbol{x}^{\mathrm{H}}$; $\boldsymbol{A}^{\mathrm{H}}$	The complex conjugate transpose of vector $\boldsymbol{x}$; the complex conjugate transpose of matrix $\boldsymbol{A}$
$\sum_{i=1}^{n} x_i$	The sum of elements x_1 to x_n
$\sqrt{a+b}$	The square root of a+b
$\int_a^b f(x)\,dx$	The integral from a to b of $f(x)\,dx$
$\|\boldsymbol{x}\|_2$	The Euclidean norm of vector $\boldsymbol{x}$, defined as: $$\sqrt{\sum_{j=1}^{n} \left|x_j\right|^2}$$

Item	Description
$\|\|\boldsymbol{A}\|\|_1$	The one norm of matrix $\boldsymbol{A}$, defined as: $$\max \left\{ \sum_{i=1}^{m} \left| a_{ij} \right|,\ 1 \le j \le n \right\}$$
$\|\|\boldsymbol{A}\|\|_2$	The spectral norm of matrix $\boldsymbol{A}$, defined as: $$\max\{\|\|\boldsymbol{Ax}\|\|_2 : \|\|\boldsymbol{x}\|\|_2 = 1\}$$
$\|\|\boldsymbol{A}\|\|_F$	The Frobenius norm of matrix $\boldsymbol{A}$, defined as: $$\sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2}$$
$\boldsymbol{A}^{-1}$	The inverse of matrix $\boldsymbol{A}$
$\boldsymbol{A}^{-T}$	The transpose of $\boldsymbol{A}$ inverse
$\|\boldsymbol{A}\|$	The determinant of matrix $\boldsymbol{A}$
m by n matrix $\boldsymbol{A}$	Matrix $\boldsymbol{A}$ has m rows and n columns
sin a	The sine of a
cos b	The cosine of b
SIGN (a)	The sign of a; the result is either + or −
address {a}	The storage address of a
max($\boldsymbol{x}$)	The maximum element in vector $\boldsymbol{x}$
min($\boldsymbol{x}$)	The minimum element in vector $\boldsymbol{x}$
ceiling(x)	The smallest integer that is greater than or equal to x
floor(x)	The largest integer that is not greater than x
int(x)	The largest integer that is less than or equal to x
x mod(m)	x modulo m; the remainder when x is divided by m
∞	Infinity
π	Pi, 3.14159265...

How to Interpret the Subroutine Descriptions

This section explains how to interpret the information in the subroutine descriptions in Part 2 of this book.

Description

Each subroutine description begins with a brief explanation of what the subroutine does. When we combine the description of multiple versions of a subroutine, we give enough information to enable you to easily tell the differences among the subroutines. Differences usually occur in either the function performed or the data types required for each subroutine.

Syntax

This shows the syntax for the Fortran, C, C++, and PL/I calling statements:

Fortran	CALL NAME-1 | NAME-2 | ... | NAME-n (*arg-1*, *arg-2*, ... ,*arg-m*, ...)
C and C++	name-1 | name-2 | ... | name-n (*arg-1*, ... ,*arg-m*);
PL/I	CALL NAME-1 | NAME-2 | ... | NAME-n (*arg-1*, *arg-2*, ... ,*arg-m*, ...);

The syntax indicates:

- The programming language (Fortran, C, C++, or PL/I)
- Each possible subroutine name that you can code in the calling sequence. Each name is separated by the | (or) symbol. You specify only one of these names in your calling sequence. (You do not code the | in the calling sequence.)
- The arguments, listed in the order in which you code them in the calling sequence. You must code them all in your calling sequence.

  You can distinguish between input arguments and output arguments by looking at the "On Entry" and "On Return" sections, respectively. An argument used for both input and output is described in both the "On Entry" and "On Return" sections. In this case, the input value for the argument is overlaid with the output value.

  The names of the arguments give an indication of the type of data that you should specify for the argument; for example:

  – Names beginning with the letters *i* through *n*, such as *m, incx, iopt*, and *isign*, indicate that you specify integer data.

  – Names beginning with the letters *a* through *h* and *o* through *z*, such as *b*, *t*, *alpha*, *sigma*, and *omega*, indicate that you specify real or complex data.

On Entry

This lists the input arguments, which are the arguments you pass to the ESSL subroutine. Each argument description first gives the meaning of the argument, and then gives the form of data required for the argument. (To help you avoid errors, output arguments are also listed, along with a reference to the On Return section.)

On Return

This lists the output arguments, which are the arguments passed back to your program from the ESSL subroutine. Each argument description first gives the meaning of the argument, and then gives the form of data passed back to your program for the argument.

Notes

The notes describe any programming considerations and restrictions that apply to the arguments or the data for the arguments. There may be references to other parts of the book for further information.

Function

This is a functional, or mathematical, description of the function performed by this subroutine. **It explains what computation is performed, not the implementation.** It explains the variations in the computation depending on the input arguments. References are made, where appropriate, to mathematical background books listed in the bibliography. References appear as a number enclosed in square brackets, where the number refers to the item listed under that number in the bibliography. For example, reference [1] cites the first item listed.

Special Usage

These are unique ways you can use the subroutine in your application. In most cases, this book does not address applications of the ESSL subroutines; however, in special situations where the functional capability of the subroutine can be extended by following certain rules for its use, these rules are described in this section.

Error Conditions

These are all the ESSL run-time errors that can occur in the subroutine. They are organized under three headings; Computational Errors, Input-Argument Errors, and Resource Errors. The return code values resulting from these errors are also explained.

Examples

The examples show how you would call the subroutine from a Fortran program. They show a variety of uses of the subroutine. Except where it is important to show differences in use between the various versions of the subroutine, the simplest version of the subroutine is used in the examples. In most cases, this is the short-precision real version of the subroutine. Each example provides a description of the important features of the example, followed by the Fortran calling sequence, the input data, and the resulting output data.

What's New for ESSL for AIX

This section summarizes all the changes made to ESSL for AIX.

What's New for ESSL Version 3

- ESSL for AIX provides two new run-time libraries:
  - The **ESSL Symmetric Multi-Processing (SMP) Library** provides thread-safe versions of the ESSL subroutines for use on the RS/6000 SMP (for example, 604 or 604e) processors. In addition, a subset of these subroutines are also multithreaded versions; that is, they support the shared memory parallel processing programming model. You do not have to change your existing application programs that call ESSL to take advantage of the increased performance of using the SMP processors. You can simply re-link your existing programs. For a list of the multithreaded subroutines in the ESSL SMP Library, see Table 21 on page 26.
  - The **ESSL Thread-Safe Library** provides thread-safe versions of the ESSL subroutines for use on all RS/6000 processors. You may use this library to develop your own multithreaded applications.

    If your existing application program calls ESSL, you only need to re-link your program to take advantage of the increased performance of the ESSL SMP Library or to use the ESSL Thread-Safe Library.
- ESSL provides new subroutines (_GETRF and _GETRS), bringing the total number of subroutines to 458.
- For those ESSL subroutines that require extra working storage to perform computations, ESSL now provides a way to dynamically allocate storage when it does not need to persist after the subroutine call. See "Using Auxiliary Storage in ESSL" on page 31.
- The files for the Hypertext Markup Language (HTML) version of the *ESSL Version 3 Guide and Reference* are packaged with the ESSL product.
- All the ESSL messages are provided in an ESSL message catalog.

Future Migration

If you are concerned with migration to possible future releases of ESSL or possible future hardware, you should read "Planning for Future Migration" on page 170. **That section explains what you can do now to prevent future migration problems.**

© Copyright IBM Corp. 1997

In Brief—What's Provided in ESSL for AIX

- ESSL provides four run-time libraries:
  - The **ESSL SMP Library** provides thread-safe versions of the ESSL subroutines for use on the RS/6000 SMP (for example, 604 or 604e) processors. In addition, a subset of these subroutines are also multithreaded versions; that is, they support the shared memory parallel processing programming model. You do not have to change your existing application programs that call ESSL to take advantage of the increased performance of using the SMP processors. You can simply re-link your existing application programs. For a list of the multithreaded subroutines in the ESSL SMP Library, see Table 21 on page 26.
  - The **ESSL Thread-Safe Library** provides thread-safe versions of the ESSL subroutines for use on all RS/6000 processors. You may use this library to develop your own multithreaded applications.
  - The **ESSL POWER Library** is tuned for the RS/6000 POWER uniprocessors, and PowerPC uniprocessors.
  - The **ESSL POWER2 Library** is tuned for the RS/6000 POWER2 uniprocessors.

  All libraries are designed to provide high levels of performance for numerically intensive computing jobs on these respective processors. All versions provide mathematically equivalent results.
- Callable from Fortran, C, C++, and PL/I programs.
- Provides an online HTML version of the *ESSL Version 3 Guide and Reference*.
- Provides PostScript files for your convenience.
- For a list of subroutines, refer to "List of ESSL Subroutines" on page 10.

© Copyright IBM Corp. 1997

Part 1. Guide Information

This part of the book is organized into seven chapters, providing guidance information on how to use ESSL. It is organized as follows:

- Learning about ESSL
- Designing your program
- Setting up your data structures
- Coding your program
- Processing your program
- Migrating your programs
- Handling problems

© Copyright IBM Corp. 1997

Chapter 1. Introduction and Requirements

This chapter introduces you to the Engineering and Scientific Subroutine Library (ESSL) for Advanced Interactive Executive (AIX*).

Overview of ESSL

This section gives an overview of the ESSL capabilities and requirements.

ESSL is a state-of-the-art collection of subroutines providing a wide range of mathematical functions for many different scientific and engineering applications. Its primary characteristics are performance, functional capability, and usability.

Performance and Functional Capability

The mathematical subroutines, in nine computational areas, are tuned for performance on the RS/6000*. The computational areas are:

- Linear Algebra Subprograms
- Matrix Operations
- Linear Algebraic Equations
- Eigensystem Analysis
- Fourier Transforms, Convolutions and Correlations, and Related Computations
- Sorting and Searching
- Interpolation
- Numerical Quadrature
- Random Number Generation

ESSL provides four run-time libraries:

- The **ESSL Symmetric Multi-Processing (SMP) Library** provides thread-safe versions of the ESSL subroutines for use on the RS/6000 SMP (for example, 604 or 604e) processors. In addition, a subset of these subroutines are also multithreaded versions; that is, they support the shared memory parallel processing programming model. For a list of the multithreaded subroutines in the ESSL SMP Library, see Table 21 on page 26.
- The **ESSL Thread-Safe Library** provides thread-safe versions of the ESSL subroutines for use on all RS/6000 processors. You may use this library to develop your own multithreaded applications.
- The **ESSL POWER Library** is tuned for the RS/6000 POWER* uniprocessors, and PowerPC* uniprocessors.
- The **ESSL POWER2 Library** is tuned for the RS/6000 POWER2* uniprocessors.

All libraries are designed to provide high levels of performance for numerically intensive computing jobs on these respective processors. All versions provide mathematically equivalent results.

The ESSL subroutines can be called from application programs written in Fortran, C, C++, and Programming Language/I (PL/I). ESSL runs under the AIX operating system.

© Copyright IBM Corp. 1997

Usability

ESSL is designed for usability:

- It has an easy-to-use call interface.
- If your existing application programs call ESSL, you only need to re-link your program to take advantage of the increased performance of the ESSL SMP Library or to use the ESSL Thread-Safe Library.
- It has informative error-handling capabilities, enabling you to calculate auxiliary storage sizes and transform lengths.
- An online book that can be displayed using an Hypertext Markup Language (HTML) document browser, is available for use with ESSL.

The Variety of Mathematical Functions

This section describes the mathematical functions included in ESSL.

Areas of Application

ESSL provides a variety of mathematical functions for many different types of scientific and engineering applications. Some of the industries using these applications are: Aerospace, Automotive, Electronics, Petroleum, Finance, Utilities, and Research. Examples of applications in these industries are:

Structural Analysis	Time Series Analysis
Computational Chemistry	Computational Techniques
Fluid Dynamics Analysis	Mathematical Analysis
Seismic Analysis	Dynamic Systems Simulation
Reservoir Modeling	Nuclear Engineering
Quantitative Analysis	Electronic Circuit Design

What ESSL Provides

The subroutines provided in ESSL, summarized in Table 1, fall into the following groups:

- Nine major areas of mathematical computation, providing the computations commonly used by the industry applications listed above
- Utilities, performing general-purpose functions

To help you select the ESSL subroutines that fulfill your needs for performance, accuracy, storage, and so forth, see "Selecting an ESSL Subroutine" on page 25.

Table 1. Summary of ESSL Subroutines

ESSL Area of Computation	Integer Subroutines	Short-Precision Subroutines	Long-Precision Subroutines
Linear Algebra Subprograms:			
• Vector-scalar	0	41	41
• Sparse vector-scalar	0	11	11
• Matrix-vector	0	32	32
• Sparse matrix-vector	0	0	3
Matrix Operations: Addition, subtraction, multiplications, rank-k updates, rank-2k updates, and matrix transposes	0	25	26
Linear Algebraic Equations:			
• Dense linear algebraic equations	0	30	32
• Banded linear algebraic equations	0	18	18
• Sparse linear algebraic equations	0	0	11
• Linear least squares	0	3	3
Eigensystem Analysis: Solutions to the algebraic eigensystem analysis problem and the generalized eigensystem analysis problem	0	8	8
(Signal Processing) Computations:			
• Fourier transforms	0	15	11
• Convolutions and correlations	0	10	2
• Related computations	0	6	6
Sorting and Searching: Sorting, sorting with index, and binary and sequential searching	5	5	5
Interpolation: Polynomial and cubic spline interpolation	0	4	4
Numerical Quadrature: Numerical quadrature on a set of points or on a function	0	6	6
Random Number Generation: Generating vectors of uniformly distributed and normally distributed random numbers	0	3	3
Utilities: General service operations	11	0	3
Total ESSL Subroutines	16	217	225

ESSL—Processing Capabilities

ESSL provides four run-time libraries, the **ESSL SMP Library**, the **ESSL Thread-Safe Library**, the **ESSL POWER2 Library**, and the **ESSL POWER Library**. These libraries are designed to provide high levels of performance for numerically intensive computing jobs on the RS/6000 processors. **To order the IBM Engineering and Scientific Subroutine Library for AIX, Version 3, specify program number 5765-C42.** Most of the subroutine calls are compatible with those in the ESSL/370 product. See the *Engineering and Scientific Subroutine Library Version 2.2 Guide and Reference* for details.

Accuracy of the Computations

ESSL provides accuracy comparable to libraries using equivalent algorithms with identical precision formats. Both short- and long-precision real versions of the subroutines are provided in most areas of ESSL. In some areas, short- and long-precision complex versions are also provided, and, occasionally, an integer version is provided. The data types operated on by the short-precision, long-precision, and integer versions of the subroutines are RS/6000 architecture precisions: ANSI/IEEE 32-bit and 64-bit binary floating-point format, and 32-bit integer. See the *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*, for more detail. (There are ESSL-specific rules that apply to the results of computations on workstation processors using the ANSI/IEEE standards. For details, see "What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?" on page 44.)

For more information on accuracy, see "Getting the Best Accuracy" on page 44.

High Performance of ESSL

Algorithms: The ESSL subroutines have been designed to provide high performance. (See references [30], [41], and [42].) To achieve this performance, the subroutines use state-of-the-art algorithms tailored to specific operational characteristics of the hardware, such as cache size, Translation Lookaside Buffer (TLB) size, and page size.

Most subroutines use the following techniques to optimize performance:

- Managing the cache and TLB efficiently so the hit ratios are maximized; that is, data is blocked so it stays in the cache or TLB for its computation.
- Accessing data stored contiguously—that is, using stride-1 computations.
- Exploiting the large number of available floating-point registers.
- Using algorithms that minimize paging.
- On the PowerPC SMP processor:
  - The ESSL SMP Library is designed to exploit the processing power and shared memory of the SMP processor. In addition, a subset of the ESSL SMP subroutines have been coded to take advantage of increased performance from multithreaded (parallel) programming techniques. For a list of the multithreaded subroutines in the ESSL SMP Library, see Table 21 on page 26.
  - Choosing the number of threads depends on the problem size, the specific subroutine being called, and the number of physical processors you are running on. To achieve optimal performance, experimentation is necessary; however, picking the number of threads equal to the number of online processors generally provides good performance in most cases. In some cases, performance may

increase if you choose the number of threads to be less than the number of online processors.

You should use the the XL Fortran XLSMPOPTS environment variable to specify the number of threads you want to create.

- On the POWER2 processor:
  - Structuring the ESSL subroutines so, where applicable, the compiled code fully utilizes the dual fixed-point and floating-point execution units. Because two Multiply-Add instructions can be executed each cycle, neglecting overhead, this allows four floating-point operations per cycle to be performed.
  - Structuring the ESSL subroutines so, where applicable, the compiled code uses the POWER2 Load and Store Floating Point Quad instructions. For example, in one cycle, two Load Floating Point Quad instructions can be executed. Neglecting overhead, this allows four doublewords to be loaded per cycle.
- On the POWER processor:
  - Using algorithms that balance floating-point operations with loads in the innermost loop.
  - Using algorithms that minimize stores in the innermost loops.
  - Structuring the ESSL subroutines so, where applicable, the compiled code uses the Multiply-Add instructions. Neglecting overhead, these instructions perform two floating-point operations per cycle.

Mathematical Techniques: All areas of ESSL use state-of-the-art mathematical techniques to achieve high performance. For example, the matrix-vector linear algebra subprograms operate on a higher-level data structure, matrix-vector rather than vector-scalar. As a result, they optimize performance directly for your program and indirectly through those ESSL subroutines using them.

The Fortran Language Interface to the Subroutines

The ESSL subroutines follow standard Fortran calling conventions and must run in the Fortran run-time environment. When ESSL subroutines are called from a program in a language other than Fortran, such as C, C++, or PL/I, the Fortran conventions must be used. This applies to all aspects of the interface, such as the linkage conventions and the data conventions. For example, array ordering must be consistent with Fortran array ordering techniques. Data and linkage conventions for each language are given in Chapter 4 on page 111.

Software and Hardware Products That Can Be Used with ESSL

This section describes the hardware and software products you can use with ESSL, as well as those products for installing ESSL and displaying the online documentation.

For ESSL—Hardware

ESSL runs on the IBM* RS/6000 processors supported by the AIX operating systems.

ESSL—Operating Systems

ESSL is supported in the following operating system environment:

- AIX Version 4.2.1 or later modification levels (either product number 5765-655 or 5765-C34)
- AIX Version 4.3 or later modification levels (product number 5765-C34)

ESSL—Software Products

ESSL requires the software products shown in Table 2 for compiling and running.

To assist C and C++ users, an ESSL header file is provided. Use of this file is described in "C Programs" on page 130 and "C++ Programs" on page 146.

Table 2. Software Products Required for Use with ESSL

For Compiling	For Linking, Loading, or Running
XL Fortran for AIX, Version 5.1 or later (program number 5765-C10) **–or–** C Set ++* for AIX, Version 3.1.4.6 or later (program number 5765-421) **–or–** C for AIX, Version 4.1 or later (program number 5765-AAR with feature number 1486) **–or–** PL/I Set for AIX, Version 1.1 or later (part numbers 33H1858 and 33H5425 and program number 5765-549)	XL Fortran Version 5.1 libraries (program number 5765-C10 or 5765-C11) **–and–** C libraries[1]

[1] AIX includes the C and math libraries.

Installation and Customization Products

The ESSL licensed program is distributed on a 4-millimeter or an 8-millimeter cartridge. The *ESSL Version 3 Installation Memo* provides the detailed information you need to install ESSL.

The ESSL product is packaged according to the AIX guidelines, as described in the *IBM AIX Version 4 General Programming Concepts: Writing and Debugging Programs* manual. The product can be installed using the `smit` command, as described in the *IBM AIX Version 4 System Management Guide: Operating System and Devices* manual.

Software Products for Displaying ESSL Online Information

To view the *ESSL Version 3 Guide and Reference* online publications, you need the following:

- Access to a common HTML document browser (such as Netscape Navigator, WebExplorer, or Mosaic).
- The location of the HTML index file provided with the file sets. Contact your system administrator or installer for this location.

ESSL Internet Resources

This section describes how you can use the ESSL resources available over the Internet.

Obtaining Documentation

This book is available in PostScript format. You can obtain PostScript files for the product documentation by:

- Printing the PostScript documentation files that are shipped on the product media.
- You can view or download this book from the IBM RS/6000 web site at `http://www.rs6000.ibm.com`. At the time this manual was published, the full path was `http://www.rs6000.ibm.com/resource/aix_resource/sp_books`. However, the structure of the RS/6000 web site can change over time.

Accessing ESSL's Product Home Pages

The following home pages contain information on ESSL and Parallel ESSL:

- For ESSL for AIX, use:

  `http://www.rs6000.ibm.com/software/Apps/essl.html`

- For Parallel ESSL, use:

  `http://www.rs6000.ibm.com/software/sp_products/esslpara.html`

Getting on the ESSL Mailing List

Information concerning ESSL's home pages and other home pages available for the RS/6000 family of products, plus late breaking information about ESSL, can be obtained by being placed on the ESSL mailing list. In addition, users on the mailing list will receive information about new ESSL function and may receive customer satisfaction surveys and requirements surveys, to provide feedback to ESSL Development on the product and user requirements.

You can be placed on the mailing list by sending a request to either of the following, asking to be placed on the ESSL mailing list:

International Business Machines Corporation
ESSL Development
Department LQJA / MS P963
522 South Rd.
Poughkeepsie, N.Y. 12601-5400

e-mail: essl@us.ibm.com

Note: You should send us e-mail if you would like to be withdrawn from the ESSL mailing list.

When requesting to be placed on the mailing list or asking any questions, please provide the following information:

- Your name
- The name of your company
- Your mailing address
- Your Internet address
- Your phone number

List of ESSL Subroutines

This section provides an overview of the subroutines in each of the areas of ESSL.

Appendix A on page APA-1 contains a list of Level 1, 2, and 3 Basic Linear Algebra Subprograms (BLAS) included in ESSL.

Appendix B on page APB-1 contains a list of Linear Algebra Package (LAPACK) subroutines included in ESSL.

Linear Algebra Subprograms

The linear algebra subprograms consist of:

- Vector-scalar linear algebra subprograms (Table 3)
- Sparse vector-scalar linear algebra subprograms (Table 4)
- Matrix-vector linear algebra subprograms (Table 5)
- Sparse matrix-vector linear algebra subprograms (Table 6)

Notes:

1. The term **subprograms** is used to be consistent with the Basic Linear Algebra Subprograms (BLAS), because many of these subprograms correspond to the BLAS.
2. Some of the linear algebra subprograms were designed in accordance with the Level 1 and Level 2 BLAS de facto standard. If these subprograms do not comply with the standard as approved, IBM will consider updating them to do so. If IBM updates these subprograms, the updates could require modifications of the calling application program.

Vector-Scalar Linear Algebra Subprograms

The vector-scalar linear algebra subprograms include a subset of the standard set of Level 1 BLAS. For details on the BLAS, see reference [73]. The remainder of the vector-scalar linear algebra subprograms are commonly used computations provided for your applications. Both real and complex versions of the subprograms are provided.

Table 3 (Page 1 of 2). List of Vector-Scalar Linear Algebra Subprograms

Descriptive Name	Short-Precision Subprogram	Long-Precision Subprogram	Page
Position of the First or Last Occurrence of the Vector Element Having the Largest Magnitude	ISAMAX†▪ ICAMAX†▪	IDAMAX†▪ IZAMAX†▪	202
Position of the First or Last Occurrence of the Vector Element Having Minimum Absolute Value	ISAMIN†	IDAMIN†	205
Position of the First or Last Occurrence of the Vector Element Having Maximum Value	ISMAX†	IDMAX†	208
Position of the First or Last Occurrence of the Vector Element Having Minimum Value	ISMIN†	IDMIN†	211
Sum of the Magnitudes of the Elements in a Vector	SASUM†▪ SCASUM†▪	DASUM†▪ DZASUM†▪	214

Table 3 (Page 2 of 2). List of Vector-Scalar Linear Algebra Subprograms

Descriptive Name	Short-Precision Subprogram	Long-Precision Subprogram	Page
Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in the Vector Y	SAXPY▪ CAXPY▪	DAXPY▪ ZAXPY▪	217
Copy a Vector	SCOPY▪ CCOPY▪	DCOPY▪ ZCOPY▪	220
Dot Product of Two Vectors	SDOT†▪ CDOTU†▪ CDOTC†▪	DDOT†▪ ZDOTU†▪ ZDOTC†▪	223
Compute SAXPY or DAXPY N Times	SNAXPY	DNAXPY	227
Compute Special Dot Products N Times	SNDOT	DNDOT	232
Euclidean Length of a Vector with Scaling of Input to Avoid Destructive Underflow and Overflow	SNRM2†▪ SCNRM2†▪	DNRM2†▪ DZNRM2†▪	237
Euclidean Length of a Vector with No Scaling of Input	SNORM2† CNORM2†	DNORM2† ZNORM2†	240
Construct a Givens Plane Rotation	SROTG▪ CROTG▪	DROTG▪ ZROTG▪	243
Apply a Plane Rotation	SROT▪ CROT▪ CSROT▪	DROT▪ ZROT▪ ZDROT▪	250
Multiply a Vector X by a Scalar and Store in the Vector X	SSCAL▪ CSCAL▪ CSSCAL▪	DSCAL▪ ZSCAL▪ ZDSCAL▪	254
Interchange the Elements of Two Vectors	SSWAP▪ CSWAP▪	DSWAP▪ ZSWAP▪	257
Add a Vector X to a Vector Y and Store in a Vector Z	SVEA CVEA	DVEA ZVEA	260
Subtract a Vector Y from a Vector X and Store in a Vector Z	SVES CVES	DVES ZVES	264
Multiply a Vector X by a Vector Y and Store in a Vector Z	SVEM CVEM	DVEM ZVEM	268
Multiply a Vector X by a Scalar and Store in a Vector Y	SYAX CYAX CSYAX	DYAX ZYAX ZDYAX	272
Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in a Vector Z	SZAXPY CZAXPY	DZAXPY ZZAXPY	275

† This subprogram is invoked as a function in a Fortran program.

▪ Level 1 BLAS

Sparse Vector-Scalar Linear Algebra Subprograms

The sparse vector-scalar linear algebra subprograms operate on sparse vectors; that is, only the nonzero elements of the vector are stored. These subprograms provide similar functions to the vector-scalar subprograms. These subprograms represent a subset of the sparse extensions to the Level 1 BLAS described in reference [29]. Both real and complex versions of the subprograms are provided.

Table 4. List of Sparse Vector-Scalar Linear Algebra Subprograms

Descriptive Name	Short-Precision Subprogram	Long-Precision Subprogram	Page
Scatter the Elements of a Sparse Vector X in Compressed-Vector Storage Mode into Specified Elements of a Sparse Vector Y in Full-Vector Storage Mode	SSCTR CSCTR	DSCTR ZSCTR	280
Gather Specified Elements of a Sparse Vector Y in Full-Vector Storage Mode into a Sparse Vector X in Compressed-Vector Storage Mode	SGTHR CGTHR	DGTHR ZGTHR	283
Gather Specified Elements of a Sparse Vector Y in Full-Vector Mode into a Sparse Vector X in Compressed-Vector Mode, and Zero the Same Specified Elements of Y	SGTHRZ CGTHRZ	DGTHRZ ZGTHRZ	286
Multiply a Sparse Vector X in Compressed-Vector Storage Mode by a Scalar, Add to a Sparse Vector Y in Full-Vector Storage Mode, and Store in the Vector Y	SAXPYI CAXPYI	DAXPYI ZAXPYI	289
Dot Product of a Sparse Vector X in Compressed-Vector Storage Mode and a Sparse Vector Y in Full-Vector Storage Mode	SDOTI† CDOTCI† CDOTUI†	DDOTI† ZDOTCI† ZDOTUI†	292

† This subprogram is invoked as a function in a Fortran program.

Matrix-Vector Linear Algebra Subprograms

The matrix-vector linear algebra subprograms operate on a higher-level data structure—matrix-vector rather than vector-scalar—using optimized algorithms to improve performance. These subprograms include a subset of the standard set of Level 2 BLAS. For details on the Level 2 BLAS, see [34] and [35]. Both real and complex versions of the subprograms are provided.

Table 5 (Page 1 of 2). List of Matrix-Vector Linear Algebra Subprograms

Descriptive Name	Short-Precision Subprogram	Long-Precision Subprogram	Page
Matrix-Vector Product for a General Matrix, Its Transpose, or Its Conjugate Transpose	SGEMV◂ CGEMV◂ SGEMX§ SGEMTX§	DGEMV◂ ZGEMV◂ DGEMX§ DGEMTX§	297

Table 5 (Page 2 of 2). List of Matrix-Vector Linear Algebra Subprograms

Descriptive Name	Short-Precision Subprogram	Long-Precision Subprogram	Page
Rank-One Update of a General Matrix	SGER◂ CGERU◂ CGERC◂	DGER◂ ZGERU◂ ZGERC◂	308
Matrix-Vector Product for a Real Symmetric or Complex Hermitian Matrix	SSPMV◂ CHPMV◂ SSYMV◂ CHEMV◂ SSLMX§	DSPMV◂ ZHPMV◂ DSYMV◂ ZHEMV◂ DSLMX§	316
Rank-One Update of a Real Symmetric or Complex Hermitian Matrix	SSPR◂ CHPR◂ SSYR◂ CHER◂ SSLR1§	DSPR◂ ZHPR◂ DSYR◂ ZHER◂ DSLR1§	324
Rank-Two Update of a Real Symmetric or Complex Hermitian Matrix	SSPR2◂ CHPR2◂ SSYR2◂ CHER2◂ SSLR2§	DSPR2◂ ZHPR2◂ DSYR2◂ ZHER2◂ DSLR2§	332
Matrix-Vector Product for a General Band Matrix, Its Transpose, or Its Conjugate Transpose	SGBMV◂ CGBMV◂	DGBMV◂ ZGBMV◂	341
Matrix-Vector Product for a Real Symmetric or Complex Hermitian Band Matrix	SSBMV◂ CHBMV◂	DSBMV◂ ZHBMV◂	348
Matrix-Vector Product for a Triangular Matrix, Its Transpose, or Its Conjugate Transpose	STRMV◂ CTRMV◂ STPMV◂ CTPMV◂	DTRMV◂ ZTRMV◂ DTPMV◂ ZTPMV◂	353
Matrix-Vector Product for a Triangular Band Matrix, Its Transpose, or Its Conjugate Transpose	STBMV◂ CTBMV◂	DTBMV◂ ZTBMV◂	359

◂ Level 2 BLAS

§ This subroutine is provided only for migration from earlier releases of ESSL and is not intended for use in new programs.

Sparse Matrix-Vector Linear Algebra Subprograms

The sparse matrix-vector linear algebra subprograms operate on sparse matrices; that is, only the nonzero elements of the matrix are stored. These subprograms provide similar functions to the matrix-vector subprograms.

Table 6. List of Sparse Matrix-Vector Linear Algebra Subprograms		
Descriptive Name	**Long- Precision Subprogram**	**Page**
Matrix-Vector Product for a Sparse Matrix in Compressed-Matrix Storage Mode	DSMMX	366
Transpose a Sparse Matrix in Compressed-Matrix Storage Mode	DSMTM	369
Matrix-Vector Product for a Sparse Matrix or Its Transpose in Compressed-Diagonal Storage Mode	DSDMX	373

Matrix Operations

Some of the matrix operation subroutines were designed in accordance with the Level 3 BLAS de facto standard. If these subroutines do not comply with the standard as approved, IBM will consider updating them to do so. If IBM updates these subroutines, the updates could require modifications of the calling application program. For details on the Level 3 BLAS, see reference [32]. The matrix operation subroutines also include the commonly used matrix operations: addition, subtraction, multiplication, and transposition.

Table 7 (Page 1 of 2). List of Matrix Operation Subroutines			
Descriptive Name	**Short-Precision Subroutine**	**Long-Precision Subroutine**	**Page**
Matrix Addition for General Matrices or Their Transposes	SGEADD CGEADD	DGEADD ZGEADD	384
Matrix Subtraction for General Matrices or Their Transposes	SGESUB CGESUB	DGESUB ZGESUB	391
Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes	SGEMUL CGEMUL	DGEMUL ZGEMUL DGEMLP§	398
Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes Using Winograd's Variation of Strassen's Algorithm	SGEMMS CGEMMS	DGEMMS ZGEMMS	408
Combined Matrix Multiplication and Addition for General Matrices, Their Transposes, or Conjugate Transposes	SGEMM♦ CGEMM♦	DGEMM♦ ZGEMM♦	413
Matrix-Matrix Product Where One Matrix is Real or Complex Symmetric or Complex Hermitian	SSYMM♦ CSYMM♦ CHEMM♦	DSYMM♦ ZSYMM♦ ZHEMM♦	422
Triangular Matrix-Matrix Product	STRMM♦ CTRMM♦	DTRMM♦ ZTRMM♦	430
Rank-K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix	SSYRK♦ CSYRK♦ CHERK♦	DSYRK♦ ZSYRK♦ ZHERK♦	437

Table 7 (Page 2 of 2). List of Matrix Operation Subroutines			
Descriptive Name	**Short-Precision Subroutine**	**Long-Precision Subroutine**	**Page**
Rank-2K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix	SSYR2K♦ CSYR2K♦ CHER2K♦	DSYR2K♦ ZSYR2K♦ ZHER2K♦	444
General Matrix Transpose (In-Place)	SGETMI CGETMI	DGETMI ZGETMI	452
General Matrix Transpose (Out-of-Place)	SGETMO CGETMO	DGETMO ZGETMO	455
♦ Level 3 BLAS § This subroutine is provided only for migration from earlier release of ESSL and is not intended for use in new programs.			

Linear Algebraic Equations

The linear algebraic equations consist of:

- Dense linear algebraic equations (Table 8)
- Banded linear algebraic equations (Table 9)
- Sparse linear algebraic equations (Table 10)
- Linear least squares (Table 11)

Notes:

1. Some of the linear algebraic equations were designed in accordance with the Level 2 BLAS, Level 3 BLAS, and LAPACK de facto standard. If these subprograms do not comply with the standard as approved, IBM will consider updating them to do so. If IBM updates these subprograms, the updates could require modifications of the calling application program. For details on the Level 2 and 3 BLAS, see [32] and [34]. For details on LAPACK, see [8].

Dense Linear Algebraic Equations

The dense linear algebraic equation subroutines provide solutions to linear systems of equations for both real and complex general matrices and their transposes, positive definite real symmetric and complex Hermitian matrices, and triangular matrices. Some of these subroutines correspond to the Level 2 BLAS, Level 3 BLAS, and LAPACK routines described in references [32], [34], and [8].

Table 8. List of Dense Linear Algebraic Equation Subroutines

Descriptive Name	Short-Precision Subroutine	Long-Precision Subroutine	Page
General Matrix Factorization	SGEF CGEF SGETRF[Δ] CGETRF[Δ]	DGEF ZGEF DGETRF[Δ] ZGETRF[Δ] DGEFP[§]	468 481
General Matrix, Its Transpose, or Its Conjugate Transpose Solve	SGES CGES	DGES ZGES	471
General Matrix, Its Transpose, or Its Conjugate Transpose Multiple Right-Hand Side Solve	SGESM CGESM SGETRS[Δ] CGETRS[Δ]	DGESM ZGESM DGETRS[Δ] ZGETRS[Δ]	475 485
General Matrix Factorization, Condition Number Reciprocal, and Determinant	SGEFCD	DGEFCD	490
Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization	SPPF SPOF CPOF	DPPF DPOF ZPOF DPPFP[§]	494
Positive Definite Real Symmetric Matrix Solve	SPPS	DPPS	502
Positive Definite Real Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve	SPOSM CPOSM	DPOSM ZPOSM	505
Positive Definite Real Symmetric Matrix Factorization, Condition Number Reciprocal, and Determinant	SPPFCD SPOFCD	DPPFCD DPOFCD	510
General Matrix Inverse, Condition Number Reciprocal, and Determinant	SGEICD	DGEICD	516
Positive Definite Real Symmetric Matrix Inverse, Condition Number Reciprocal, and Determinant	SPPICD SPOICD	DPPICD DPOICD	521
Solution of a Triangular System of Equations with a Single Right-Hand Side	STRSV[◂] CTRSV[◂] STPSV[◂] CTPSV[◂]	DTRSV[◂] ZTRSV[◂] DTPSV[◂] ZTPSV[◂]	528
Solution of Triangular Systems of Equations with Multiple Right-Hand Sides	STRSM[♦] CTRSM[♦]	DTRSM[♦] ZTRSM[♦]	534
Triangular Matrix Inverse	STRI STPI	DTRI DTPI	542

◂ Level 2 BLAS

♦ Level 3 BLAS

Δ LAPACK

§ This subroutine is provided only for migration from earlier releases of ESSL and is not intended for use in new programs. Documentation for this subroutine is no longer provided.

Banded Linear Algebraic Equations

The banded linear algebraic equation subroutines provide solutions to linear systems of equations for real general band matrices, real positive definite symmetric band matrices, real or complex general tridiagonal matrices, real positive definite symmetric tridiagonal matrices, and real or complex triangular band matrices.

Table 9. List of Banded Linear Algebraic Equation Subroutines

Descriptive Name	Short-Precision Subroutine	Long-Precision Subroutine	Page
General Band Matrix Factorization	SGBF	DGBF	548
General Band Matrix Solve	SGBS	DGBS	552
Positive Definite Symmetric Band Matrix Factorization	SPBF SPBCHF	DPBF DPBCHF	555
Positive Definite Symmetric Band Matrix Solve	SPBS SPBCHS	DPBS DPBCHS	559
General Tridiagonal Matrix Factorization	SGTF	DGTF	562
General Tridiagonal Matrix Solve	SGTS	DGTS	565
General Tridiagonal Matrix Combined Factorization and Solve with No Pivoting	SGTNP CGTNP	DGTNP ZGTNP	567
General Tridiagonal Matrix Factorization with No Pivoting	SGTNPF CGTNPF	DGTNPF ZGTNPF	570
General Tridiagonal Matrix Solve with No Pivoting	SGTNPS CGTNPS	DGTNPS ZGTNPS	573
Positive Definite Symmetric Tridiagonal Matrix Factorization	SPTF	DPTF	576
Positive Definite Symmetric Tridiagonal Matrix Solve	SPTS	DPTS	578
Triangular Band Equation Solve	STBSV◂ CTBSV◂	DTBSV◂ ZTBSV◂	580

◂ Level 2 BLAS

Sparse Linear Algebraic Equations

The sparse linear algebraic equation subroutines provide direct and iterative solutions to linear systems of equations both for general sparse matrices and their transposes and for sparse symmetric matrices.

Table 10 (Page 1 of 2). List of Sparse Linear Algebraic Equation Subroutines

Descriptive Name	Long- Precision Subroutine	Page
General Sparse Matrix Factorization Using Storage by Indices, Rows, or Columns	DGSF	587
General Sparse Matrix or Its Transpose Solve Using Storage by Indices, Rows, or Columns	DGSS	593

Table 10 (Page 2 of 2). List of Sparse Linear Algebraic Equation Subroutines		
Descriptive Name	**Long- Precision Subroutine**	**Page**
General Sparse Matrix or Its Transpose Factorization, Determinant, and Solve Using Skyline Storage Mode	DGKFS DGKFSP§	597
Symmetric Sparse Matrix Factorization, Determinant, and Solve Using Skyline Storage Mode	DSKFS DSKFSP§	615
Iterative Linear System Solver for a General or Symmetric Sparse Matrix Stored by Rows	DSRIS	634
Sparse Positive Definite or Negative Definite Symmetric Matrix Iterative Solve Using Compressed-Matrix Storage Mode	DSMCG‡	645
Sparse Positive Definite or Negative Definite Symmetric Matrix Iterative Solve Using Compressed-Diagonal Storage Mode	DSDCG	653
General Sparse Matrix Iterative Solve Using Compressed-Matrix Storage Mode	DSMGCG‡	661
General Sparse Matrix Iterative Solve Using Compressed-Diagonal Storage Mode	DSDGCG	668
§ This subroutine is provided only for migration from earlier releases of ESSL and is not intended for use in new programs. Documentation for this subroutine is no longer provided. ‡ This subroutine is provided only for migration from earlier releases of ESSL and is not intended for use in new programs. Use DSRIS instead.		

Linear Least Squares

The linear least squares subroutines provide least squares solutions to linear systems of equations for real general matrices. Two methods are provided: one that uses the singular value decomposition and another that uses a QR decomposition with column pivoting.

Table 11. List of Linear Least Squares Subroutines			
Descriptive Name	**Short-Precision Subroutine**	**Long-Precision Subroutine**	**Page**
Singular Value Decomposition for a General Matrix	SGESVF	DGESVF	676
Linear Least Squares Solution for a General Matrix Using the Singular Value Decomposition	SGESVS	DGESVS	684
Linear Least Squares Solution for a General Matrix Using a QR Decomposition with Column Pivoting	SGELLS	DGELLS	689

Eigensystem Analysis

The eigensystem analysis subroutines provide solutions to the algebraic eigensystem analysis problem $\boldsymbol{Az} = w\boldsymbol{z}$ and the generalized eigensystem analysis problem $\boldsymbol{Az} = w\boldsymbol{Bz}$ (Table 12). Many of the eigensystem analysis subroutines use the algorithms presented in *Linear Algebra* by Wilkinson and Reinsch [93] or use adaptations of EISPACK routines, as described in the *EISPACK Guide Lecture Notes in Computer Science* in reference [81]

or in the *EISPACK Guide Extension Lecture Notes in Computer Science* in reference [55]. (EISPACK is available from the sources listed in reference [49].)

Table 12. List of Eigensystem Analysis Subroutines

Descriptive Name	Short-Precision Subroutine	Long-Precision Subroutine	Page
Eigenvalues and, Optionally, All or Selected Eigenvectors of a General Matrix	SGEEV CGEEV	DGEEV ZGEEV	698
Eigenvalues and, Optionally, the Eigenvectors of a Real Symmetric Matrix or a Complex Hermitian Matrix	SSPEV CHPEV	DSPEV ZHPEV	710
Extreme Eigenvalues and, Optionally, the Eigenvectors of a Real Symmetric Matrix or a Complex Hermitian Matrix	SSPSV CHPSV	DSPSV ZHPSV	719
Eigenvalues and, Optionally, the Eigenvectors of a Generalized Real Eigensystem, Az=wBz, where A and B Are Real General Matrices	SGEGV	DGEGV	727
Eigenvalues and, Optionally, the Eigenvectors of a Generalized Real Symmetric Eigensystem, Az=wBz, where A Is Real Symmetric and B Is Real Symmetric Positive Definite	SSYGV	DSYGV	733

Fourier Transforms, Convolutions and Correlations, and Related Computations

This signal processing area provides:

- Fourier transform subroutines (Table 13)
- Convolution and correlation subroutines (Table 14)
- Related-computation subroutines (Table 15)

Fourier Transforms

The Fourier transform subroutines perform mixed-radix transforms in one, two, and three dimensions.

Table 13 (Page 1 of 2). List of Fourier Transform Subroutines

Descriptive Name	Short-Precision Subroutine	Long-Precision Subroutine	Page
Complex Fourier Transform	SCFT SCFTP§	DCFT	749
Real-to-Complex Fourier Transform	SRCFT	DRCFT	758
Complex-to-Real Fourier Transform	SCRFT	DCRFT	766
Cosine Transform	SCOSF SCOSFT§	DCOSF	774
Sine Transform	SSINF	DSINF	782

Table 13 (Page 2 of 2). List of Fourier Transform Subroutines

Descriptive Name	Short-Precision Subroutine	Long-Precision Subroutine	Page
Complex Fourier Transform in Two Dimensions	SCFT2 SCFT2P§	DCFT2	790
Real-to-Complex Fourier Transform in Two Dimensions	SRCFT2	DRCFT2	797
Complex-to-Real Fourier Transform in Two Dimensions	SCRFT2	DCRFT2	804
Complex Fourier Transform in Three Dimensions	SCFT3 SCFT3P§	DCFT3	812
Real-to-Complex Fourier Transform in Three Dimensions	SRCFT3	DRCFT3	818
Complex-to-Real Fourier Transform in Three Dimensions	SCRFT3	DCRFT3	824

§ This subroutine is provided only for migration from earlier releases of ESSL and is not intended for use in new programs. Documentation for this subroutine is no longer provided.

Convolutions and Correlations

The convolution and correlation subroutines provide the choice of using Fourier methods or direct methods. The Fourier-method subroutines contain a high-performance mixed-radix capability. There are also several direct-method subroutines that provide decimated output.

Table 14. List of Convolution and Correlation Subroutines

Descriptive Name	Short-Precision Subroutine	Long-Precision Subroutine	Page
Convolution or Correlation of One Sequence with One or More Sequences	SCON§ SCOR§		831
Convolution or Correlation of One Sequence with Another Sequence Using a Direct Method	SCOND SCORD		837
Convolution or Correlation of One Sequence with One or More Sequences Using the Mixed-Radix Fourier Method	SCONF SCORF		843
Convolution or Correlation with Decimated Output Using a Direct Method	SDCON SDCOR	DDCON DDCOR	852
Autocorrelation of One or More Sequences	SACOR§		856
Autocorrelation of One or More Sequences Using the Mixed-Radix Fourier Method	SACORF		860

§ These subroutines are provided only for migration from earlier releases of ESSL and are not intended for use in new programs.

Related Computations

The related-computation subroutines consist of a group of computations that can be used in general signal processing applications. They are similar to those provided on the IBM 3838 Array Processor; however, the ESSL subroutines generally solve a wider range of problems.

Table 15. List of Related-Computation Subroutines

Descriptive Name	Short-Precision Subroutine	Long-Precision Subroutine	Page
Polynomial Evaluation	SPOLY	DPOLY	867
I-th Zero Crossing	SIZC	DIZC	870
Time-Varying Recursive Filter	STREC	DTREC	873
Quadratic Interpolation	SQINT	DQINT	876
Wiener-Levinson Filter Coefficients	SWLEV CWLEV	DWLEV ZWLEV	880

Sorting and Searching

The sorting and searching subroutines operate on three types of data: integer, short-precision real, and long-precision real (Table 16). The sorting subroutines perform sorts with or without index designations. The searching subroutines perform either a binary or sequential search.

Table 16. List of Sorting and Searching Subroutines

Descriptive Name	Integer Subroutine	Short-Precision Subroutine	Long-Precision Subroutine	Page
Sort the Elements of a Sequence	ISORT	SSORT	DSORT	888
Sort the Elements of a Sequence and Note the Original Element Positions	ISORTX	SSORTX	DSORTX	890
Sort the Elements of a Sequence Using a Stable Sort and Note the Original Element Positions	ISORTS	SSORTS	DSORTS	893
Binary Search for Elements of a Sequence X in a Sorted Sequence Y	IBSRCH	SBSRCH	DBSRCH	896
Sequential Search for Elements of a Sequence X in the Sequence Y	ISSRCH	SSSRCH	DSSRCH	900

Interpolation

The interpolation subroutines provide the capabilities of doing polynomial interpolation, local polynomial interpolation, and both one- and two-dimensional cubic spline interpolation (Table 17).

Table 17. List of Interpolation Subroutines

Descriptive Name	Short-Precision Subroutine	Long-Precision Subroutine	Page
Polynomial Interpolation	SPINT	DPINT	907
Local Polynomial Interpolation	STPINT	DTPINT	912
Cubic Spline Interpolation	SCSINT	DCSINT	915
Two-Dimensional Cubic Spline Interpolation	SCSIN2	DCSIN2	921

Numerical Quadrature

The numerical quadrature subroutines provide Gaussian quadrature methods for integrating a tabulated function and a user-supplied function over a finite, semi-infinite, or infinite region of integration (Table 18).

Table 18. List of Numerical Quadrature Subroutines

Descriptive Name	Short-Precision Subroutine	Long-Precision Subroutine	Page
Numerical Quadrature Performed on a Set of Points	SPTNQ	DPTNQ	929
Numerical Quadrature Performed on a Function Using Gauss-Legendre Quadrature	SGLNQ†	DGLNQ†	932
Numerical Quadrature Performed on a Function Over a Rectangle Using Two-Dimensional Gauss-Legendre Quadrature	SGLNQ2†	DGLNQ2†	935
Numerical Quadrature Performed on a Function Using Gauss-Laguerre Quadrature	SGLGQ†	DGLGQ†	941
Numerical Quadrature Performed on a Function Using Gauss-Rational Quadrature	SGRAQ†	DGRAQ†	945
Numerical Quadrature Performed on a Function Using Gauss-Hermite Quadrature	SGHMQ†	DGHMQ†	949

† This subprogram is invoked as a function in a Fortran program.

Random Number Generation

Random number generation subroutines generate uniformly distributed random numbers or normally distributed random numbers (Table 19).

Table 19. List of Random Number Generation Subroutines

Descriptive Name	Short-Precision Subroutine	Long-Precision Subroutine	Page
Generate a Vector of Uniformly Distributed Random Numbers	SURAND	DURAND	954
Generate a Vector of Normally Distributed Random Numbers	SNRAND	DNRAND	957
Generate a Vector of Long Period Uniformly Distributed Random Numbers	SURXOR	DURXOR	961

Utilities

The utility subroutines perform general service functions that support ESSL, rather than mathematical computations (Table 20).

Table 20. List of Utility Subroutines

Descriptive Name	Subroutine	Page
ESSL Error Information-Handler Subroutine	EINFO	968
ESSL ERRSAV Subroutine for ESSL	ERRSAV	971
ESSL ERRSET Subroutine for ESSL	ERRSET	972
ESSL ERRSTR Subroutine for ESSL	ERRSTR	974
Set the Vector Section Size (VSS) for the ESSL/370 Scalar Library	IVSSET§	
Set the Extended Vector Operations Indicator for the ESSL/370 Scalar Library	IEVOPS§	
Determine the Level of ESSL Installed	IESSL	975
Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines	STRIDE	977
Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode	DSRSM	986
For a General Sparse Matrix, Convert Between Diagonal-Out and Profile-In Skyline Storage Mode	DGKTRN	990
For a Symmetric Sparse Matrix, Convert Between Diagonal-Out and Profile-In Skyline Storage Mode	DSKTRN	996

§ This subroutine is provided for migration from earlier releases of ESSL and is not intended for use in new programs. Documentation for this subroutine is no longer provided.

Chapter 2. Planning Your Program

This chapter provides information about ESSL that you need when planning your program. Its purpose is to help you in performing the following tasks:

- Selecting an ESSL subroutine
- Avoiding Conflicts with Internal ESSL Routine Names That are Exported
- Setting up your data
- Setting up your ESSL calling sequences
- Using auxiliary storage in ESSL
- Providing a correct transform length to ESSL
- Getting the best accuracy
- Getting the best performance
- Dealing with errors when using ESSL

If you are using ESSL with PL/I Set for AIX, Version 1, see the following PL/I publications for details on calling subroutines and functions:

- *PL/I Set for AIX Programming Guide*
- *PL/I Set for AIX Language Reference*

Selecting an ESSL Subroutine

Your choice of which ESSL subroutine to use is based mainly on the functional needs of your program. However, you have a choice of several variations of many of the subroutines. In addition, there are instances where certain subroutines cannot be used. This section describes these variations and limitations. See the answers to each question below that applies to you.

Which ESSL Library Do You Want to Use?

ESSL provides four run-time libraries:

- The **ESSL SMP Library** provides thread-safe versions of the ESSL subroutines for use on the RS/6000 SMP processors. In addition, a subset of these subroutines are also multithreaded versions; that is, they support the shared memory parallel processing programming model. For a list of the multithreaded subroutines in the ESSL SMP Library, see Table 21 on page 26.

  For more information about XL Fortran, see *IBM XL Fortran for AIX User's Guide Version 5* and *IBM XL Fortran for AIX Language Reference Version 5*.

- The **ESSL Thread-Safe Library** provides thread-safe versions of the ESSL subroutines for use on all RS/6000 processors. You may choose to use this library if you decide to develop your own multithreaded programs that call the thread-safe ESSL subroutines.

  The number of threads you choose to use depends on the problem size, the specific subroutine being called, and the number of physical processors you are running on. To achieve optimal performance, experimentation is necessary; however, picking the number of threads equal to the number of online processors generally provides good performance in most cases. In a few cases, performance may increase if you choose the number of threads to be less than the number of online processors. For more information about thread concepts, see *IBM AIX Version 4 General Programming Concepts: Writing and Debugging Programs*.

© Copyright IBM Corp. 1997

- The **ESSL POWER2 Library** is tuned for the RS/6000 POWER2 uniprocessors.
- The **ESSL POWER Library** is tuned for the RS/6000 POWER uniprocessors, and PowerPC uniprocessors.

Table 21 (Page 1 of 2). Multithreaded ESSL SMP Subroutines

Subroutine Names
Vector-Scalar Linear Algebra Subprograms: SASUM, DASUM, SCASUM, DZASUM SAXPY, DAXPY, CAXPY, ZAXPY SCOPY, DCOPY, CCOPY, ZCOPY SDOT, DDOT, CDOTU, ZDOTU, CDOTC, ZDOTC SNDOT, DNDOT SNORM2, DNORM2, CNORM2, ZNORM2 SROT, DROT, CROT, ZROT, CSROT, ZDROT SSCAL, DSCAL, CSCAL, ZSCAL, CSSCAL, ZDSCAL SSWAP, DSWAP, CSWAP, ZSWAP SVEA, DVEA, CVEA, ZVEA SVES, DVES, CVES, ZVES SVEM, DVEM, CVEM, ZVEM SYAX, DYAX, CYAX, ZYAX, CSYAX, ZDYAX SZAXPY, DZAXPY, CZAXPY, ZZAXPY
Matrix-Vector Linear Algebra Subprograms: SGEMV, DGEMV, CGEMV, ZGEMV SGER, DGER, CGERU, ZGERU, CGERC, ZGERC SSPMV, DSPMV, CHPMV, ZHPMV SSYMV, DSYMV, CHEMV, ZHEMV SSPR, DSPR, CHPR, ZHPR SSYR, DSYR, CHER, ZHER SSPR2, DSPR2, CHPR2, ZHPR2 SSYR2, DSYR2, CHER2, ZHER2 SGBMV, DGBMV, CGBMV, ZGBMV SSBMV, DSBMV, CHBMV, ZHBMV STRMV, DTRMV, CTRMV, ZTRMV STPMV, DTPMV, CTPMV, ZTPMV STBMV, DTBMV, CTBMV, ZTBMV
Matrix Operations: SGEADD, DGEADD, CGEADD, ZGEADD SGESUB, DGESUB, CGESUB, ZGESUB DGEMUL DGEMM, ZGEMM DSYMM, ZSYMM, ZHEMM DTRMM, ZTRMM DSYRK, ZSYRK, ZHERK DSYR2K, ZSYR2K, ZHER2K SGETMI, DGETMI, CGETMI, ZGETMI SGETMO, DGETMO, CGETMO, ZGETMO

Table 21 (Page 2 of 2). Multithreaded ESSL SMP Subroutines
Subroutine Names
Dense Linear Algebraic Equations: SGEF, DGEF, CGEF, ZGEF SGETRF, DGETRF, CGETRF, ZGETRF DPOF STRSV, DTRSV, CTRSV, ZTRSV STPSV, DTPSV, CTPSV, ZTPSV DTRSM, ZTRSM STRI, DTRI
Sparse Linear Algebraic Equations: DSRIS†
Fourier Transforms: SCFT, DCFT SRCFT, DRCFT SCRFT, DCRFT SCFT2, DCFT2 SRCFT2, DRCFT2 SCRFT2, DCRFT2 SCFT3, DCFT3 SRCFT3, DRCFT3 DCRFT3, DCRFT3
Convolution and Correlation: SCOND, SCORD SDCON, SDCOR, DDCON, DDCOR
Many of the dense linear algebraic equations and eigensystem analysis subroutines make one or more calls to the multithreaded versions of the matrix-vector linear algebra and matrix operation subroutines shown in this table. SCOSF, DCOSF, SSINF, and DSINF make one or more calls to the multithreaded versions of the Fourier Transform subroutines shown in this table. These subroutines benefit from the increased performance of the multithreaded versions of the ESSL SMP subroutines. † DSRIS only runs on multiple threads when `IPARM(4) = 1` or 2.

What Type of Data Are You Processing in Your Program?

The version of the ESSL subroutine you select should agree with the data you are using. ESSL provides a short- and long-precision version of most of its subroutines processing short- and long-precision data, respectively. In a few cases, it also provides an integer version processing integer data or returning just integer data. The subroutine names are distinguished by a one- or two-letter prefix based on the following letters:

S for short-precision real
D for long-precision real
C for short-precision complex
Z for long-precision complex
I for integer

The precision of your data affects the accuracy of your results. This is discussed in "Getting the Best Accuracy" on page 44. For a description of these data types, see "How Do You Set Up Your Scalar Data?" on page 28.

How Is Your Data Structured? And What Storage Technique Are You Using?

Some subroutines process specific data structures, such as sparse vectors and matrices or dense and banded matrices. In addition, these data structures can be stored using various storage techniques. You should select the proper subroutine on the basis of the type of data structure you have and the storage technique you want to use. If possible, you should use a storage technique that conserves storage and potentially improves performance. For more about storage techniques, see "Setting Up Your Data."

What about Performance and Accuracy?

ESSL provides variations among some of its subroutines. You should consider performance and accuracy when deciding which subroutine is the best to use. Study the "Function" section in each subroutine description. It helps you understand exactly what each subroutine does, and helps you determine which subroutine is best for you. For example, some subroutines perform multiple computations of a certain type. This might give you better performance than a subroutine that does each computation individually. In other cases, one subroutine may do scaling while another does not. If scaling is not necessary for your data, you get better performance by using the subroutine without scaling.

Avoiding Conflicts with Internal ESSL Routine Names That are Exported

Do not use names for your own subroutines, functions, and global variables that are the same as the ESSL exported names. All internal ESSL routine names that are exported begin with the **ESV** prefix, so you should avoid using this prefix for your own names.

Setting Up Your Data

This section explains how to set up your scalar and array data and points you to where you can find more detail.

How Do You Set Up Your Scalar Data?

A scalar item is a single item of data, whether it is a constant, a variable, or an element of an array. ESSL assumes that your scalar data conforms to the appropriate standards, as described below. The scalar data types and how you should code them for each programming language are listed under "Coding Your Scalar Data" in each language section in Chapter 4 on page 111.

Internal Representation

Scalar data passed to ESSL from all types of programs, including Fortran, C, and C++, should conform to the ANSI/IEEE 32-bit and 64-bit binary floating-point format, as described in the *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*

How Do You Set Up Your Arrays?

An array represents an area of storage in your program, containing data stored in a series of locations. An array has a single name. It is made up of one or more pieces of scalar data, all the same type. These are the elements of the array. It can be passed to the ESSL subroutine as input, returned to your program as output, or used for both input and output, in which case the original contents are overwritten.

Arrays can contain conceptual (mathematical) data structures, such as vectors, matrices, or sequences. There are many different types of data structures. Each type of data structure requires a unique arrangement of data in an array and does not necessarily have to include all the elements of the array. In addition, the elements of these data structures are not always contiguous in storage within an array. Stride and leading dimension arguments passed to ESSL subroutines define the separations in array storage for the elements of the vector, matrix, and sequence. All these aspects of data structures are described in Chapter 3 on page 55. You must first understand array storage techniques to fully understand the concepts of data structures, stride, and leading dimension, especially if you are using them in unconventional ways.

ESSL subroutines assume that all arrays passed to them are stored using the Fortran array storage techniques (in column-major order), and they process your data accordingly. For details, see "Setting Up Arrays in Fortran" on page 112. On the other hand, C, C++, and PL/I programs store arrays in row-major order. For details on what you can do, see:

- For C, see page "Setting Up Arrays in C" on page 134.
- For C++, see page "Setting Up Arrays in C++" on page 150.
- For PL/I, see the following PL/I publications:
  - *PL/I Set for AIX Programming Guide*
  - *PL/I Set for AIX Language Reference*

How Should Your Array Data Be Aligned?

All arrays, regardless of the type of data, should be aligned on a doubleword boundary to ensure optimal performance; however, when running on a POWER2 processor, it is best to align your long-precision arrays on a quadword boundary. For information on how your programming language aligns data, see your programming language manuals.

What Storage Mode Should You Use for Your Data?

The amount of storage used by arrays and the storage arrangement of data in the arrays can affect overall program performance. As a result, ESSL provides subroutines that operate on different types of data structures, stored using various storage modes. You should chose a storage mode that conserves storage and potentially improves performance. For definitions of the various data structures and their corresponding storage modes, see Chapter 3 on page 55. You can also find special storage considerations, where applicable, in the "Notes" section of each subroutine description.

How Do You Convert from One Storage Mode to Another?

This section describes how you can convert from one storage mode to another.

Conversion Subroutines

ESSL provides several subroutines that help you convert from one storage mode to another:

- DSRSM is used to migrate your existing program from sparse matrices stored by rows to sparse matrices stored in compressed-matrix storage mode. This converts the matrices into a storage format that is compatible with the input requirements for some ESSL sparse matrix subroutines, such as DSMMX.
- DGKTRN and DSKTRN are used to convert your sparse matrix from one skyline storage mode to another, if necessary, before calling the subroutines DGKFS/DGKFSP or DSKFS/DSKFSP, respectively.

Sample Programs

In addition, sample programs are provided with many of the storage mode descriptions in Chapter 3 on page 55. You can use these sample programs to convert your data to the desired storage mode by adapting them to your application program.

Setting Up Your ESSL Calling Sequences

This section gives the general rules for setting up the ESSL calling sequences. The information given here applies to all types of programs, running in all environments. For a description and examples of how to code the ESSL calling sequences in your particular programming language, see the following sections:

- "Fortran Programs" on page 111
- "C Programs" on page 130
- "C++ Programs" on page 146

For details on the conventions used in this book to describe the calling sequence syntax, see "How to Interpret the Subroutine Descriptions" on page xxvii. It describes how required and optional arguments are indicated in the calling sequence and the naming conventions used for different data types.

What Is an Input-Output Argument?

Some arguments are used for both input and output. The contents of the input argument are overlaid with the output value(s) on return to your program. Be careful that you save any data you need to preserve before calling the ESSL subroutine.

What Are the General Rules to Follow when Specifying Data for the Arguments?

You should follow the syntax rules given for each argument in "On Entry" in the subroutine description. Input-argument error messages may be issued, and your program may terminate when you make an error specifying the input arguments. For example:

- Data passed to ESSL must be of the correct type: integer, character, real, complex, short-precision, or long-precision. There is no conversion of data. Assuming you are using the ESSL header file with your C and C++ programs, you first need to define the following:
  - Complex and logical data in C programs, using the guidelines given on page 133.
  - Short-precision complex and logical data in C++ programs, using the guidelines given on page 148.

- Character values must be one of the specified values. For example, it may have to be 'N', 'T', or 'C'.
- Numeric values must fall within the correct range for that argument. For example, a numeric value may need to be greater than or equal to 0, or it may have to be a nonzero value.
- Arrays must be defined correctly; that is, they must have the correct dimensions, or the dimensions must fall within the correct range. For example, input and output matrices may need to be conformable, or the number of rows in the matrix must be less than or equal to the leading dimension specified. (ESSL assumes all arrays are stored in column-major order.)

What Happens When a Value of 0 Is Specified for N?

For most ESSL subroutines, if you specify 0 for the number of elements to be processed in a vector or the order of a matrix (usually argument *n*), no computation is performed. After checking for input-argument errors, the subroutine returns immediately and no result is returned. In the other subroutines, an error message may be issued.

How Do You Specify the Beginning of the Data Structure in the ESSL Calling Sequence?

When you specify a vector, matrix, or sequence in your calling sequence, it does not necessarily have to start at the beginning of the array. It can begin at any point in the array. For example, if you want vector ***x*** to start at element 3 in array A, which is declared A(1:12), specify A(3) in your calling sequence for argument *x*, such as in the following SASUM calling sequence in your Fortran program:

```
             N    X   INCX
             |    |     |
X = SASUM( 4 , A(3) , 2 )
```

Also, for example, if you want matrix ***A*** to start at the second row and third column of array A, which is declared A(0:10,2:8), specify A(1,4) in your calling sequence for argument *a*, such as in the following SGEADD calling sequence in your Fortran program:

```
                  A     LDA  TRANSA  B  LDB TRANSB  C  LDC  M   N
                  |      |     |     |   |    |     |   |   |   |
CALL  SGEADD( A(1,4) , 11 , 'N'  , B , 4 , 'N'  , C , 4 , 4 , 3 )
```

For more examples of specifying vectors and matrices, see Chapter 3 on page 55.

Using Auxiliary Storage in ESSL

For the ESSL subroutines listed in Table 22 on page 32, you need to provide extra working storage to perform the computation. This section describes the use of dynamic allocation for providing auxiliary storage in ESSL and how to calculate the amount of auxiliary storage you need by use of formulas or error-handling capabilities provided in ESSL, if dynamic allocation is not an option.

Auxiliary storage, or working storage, is supplied through one or more arguments, such as *aux*, in the calling sequence for the ESSL subroutine. **If the working storage does not need to persist after the subroutine call, it is suggested you use dynamic allocation.** For example, in the Fourier Transforms subroutines, you may allocate *aux2* dynamically,

but not *aux1*. See the subroutine descriptions in Part 2 of this book for details and variations.

Table 22. ESSL Subroutines Requiring Auxiliary Working Storage
Subroutine Names
Linear Algebra Subprograms: DSMTM
Matrix Operations: _GEMMS
Dense Linear Algebraic Equations: _GEFCD _PPFCD _GEICD _PPICD _POFCD _POICD DGEFP$^{\Delta}$ DPPFP$^{\Delta}$
Sparse Linear Algebraic Equations: DGSF DGSS DGKFS DGKFSP$^{\Delta}$ DSKFS DSKFSP$^{\Delta}$ DSRIS DSMCG DSDCG DSMGCG DSDGCG
Linear Least Squares: _GESVF _GELLS
Eigensystem Analysis: _GEEV _SPEV _HPEV _SPSV _HPSV _GEGV _SYGV
Fourier Transforms: _CFT _RCFT _CRFT _COSF _SINF SCOSFT$^{\Delta}$ _CFT2 _RCFT2 _CRFT2 _CFT3 _RCFT3 _CRFT3 SCFTP$^{\Delta}$ SCFT2P$^{\Delta}$ SCFT3P$^{\Delta}$
Convolutions and Correlations: SCONF SCORF SACORF
Related Computations: _WLEV
Interpolation: _TPINT _CSIN2
Random Number Generation: _NRAND
Utilities: DGKTRN DSKTRN
$^{\Delta}$ Documentation for this subroutine is no longer provided. The *aux* and *naux* arguments for the subroutine are specified the same as for the corresponding serial ESSL subroutine.

Dynamic Allocation of Auxiliary Storage

Dynamic allocation for the auxiliary storage is performed when error 2015 is unrecoverable and *naux*=0. For details on which *aux* arguments allow dynamic allocation, see the subroutine descriptions in Part 2 of this book.

Setting Up Auxiliary Storage When Dynamic Allocation Is Not Used

You set up the storage area in your program and pass it to ESSL through arguments, specifying the size of the *aux* work area in the *naux* argument.

Who Do You Want to Calculate the Size? You or ESSL?

You have a choice of two methods for determining how much auxiliary storage you should specify:

- Use the formulas provided in the subroutine description to derive **sufficient values** for your current and future needs. Use them if **ease of migration** to future machines and future releases of ESSL is your primary concern. For details, see "How Do You Calculate the Size Using the Formulas?."
- Use the ESSL error-handling facilities to return to you a **minimum value** for the particular processor you are currently running on. (Values vary by platform.) Use this approach if **conserving storage** is your primary concern. For details, see "How Do You Get ESSL to Calculate the Size Using ESSL Error Handling?."

How Do You Calculate the Size Using the Formulas?

The formulas provided for calculating *naux* indicate a **sufficient** amount of auxiliary storage required, which, in most cases, is larger than the minimum amount, returned by ESSL error handling. There are two types of formulas:

- **Simple formulas**

  These are given in the *naux* argument syntax descriptions. In general, these formulas result in the minimum required value, but, in a few cases, they provide overestimates.

- **Processor-independent formulas**

  These are given in separate sections in the subroutine description. In general, these provide overestimates.

Both types of formulas provide values that are sufficient for all processors. As a result, you can migrate to any other processor and to future releases of ESSL without being concerned about having to increase the amount of storage for *aux*. You do, of course, need to weigh your storage requirements against the convenience of using this larger value.

To calculate the amount of storage using the formulas, you must substitute values for specific variables, such as *n*, *m*, *n1*, or *n2*. These variables are arguments specified in the ESSL calling sequence or derived from the arguments in the calling sequence.

How Do You Get ESSL to Calculate the Size Using ESSL Error Handling?

This section describes how you can get ESSL to calculate auxiliary storage.

Here Are the Two Ways You Can Do It

Ask yourself which of the following ways you prefer to obtain the information from ESSL:

- **By leaving error 2015 unrecoverable,** you can obtain the minimum required value of *naux* from the input-argument error message, but your program terminates.

- **By making error 2015 recoverable,** you can obtain the minimum required value of *naux* from the input-argument error message and have the updated *naux* argument returned to your program.

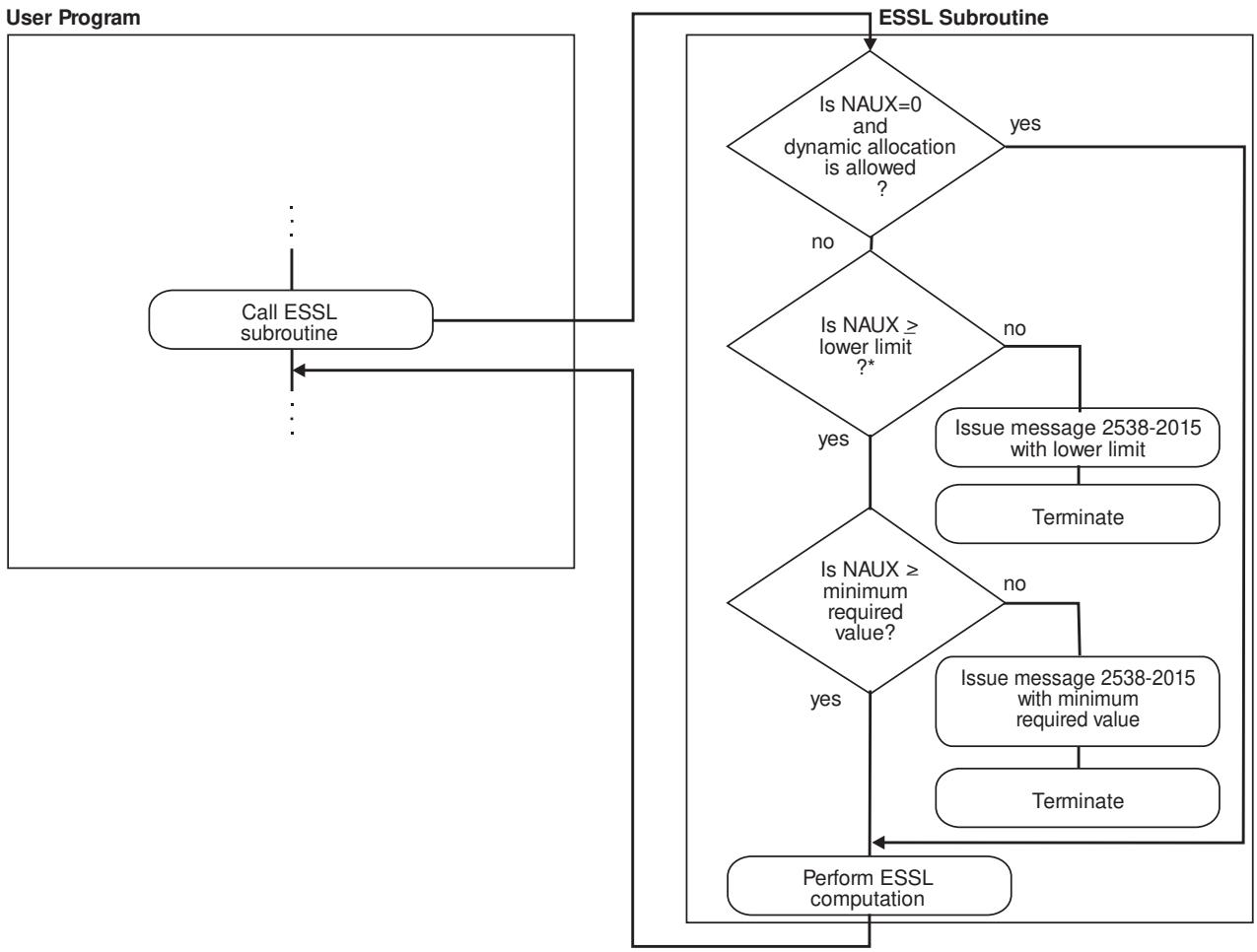
For both techniques, the amount returned by the ESSL error-handling facility is the **minimum** amount of auxiliary storage required to run your program successfully **on the particular processor you are currently running on.** The ESSL error-handling capability usually returns a smaller value than you derive by using the formulas listed for the subroutine. This is because the formulas provide a good estimate, but ESSL can calculate exactly what is needed on the basis of your data.

The values returned by ESSL error handling **may not apply to future processors.** You should not use them if you plan to run your program on a future processor. You should use them only if you are concerned with minimizing the amount of auxiliary storage used by your program.

The First Way

In this case, you obtain the minimum required value of *naux* from the error message, but your program terminates. The following description assumes that dynamic allocation is not selected as an option.

Leave error 2015 as unrecoverable, without calls to EINFO and ERRSET. Run your program with the *naux* values smaller than required by the subroutine for the particular processor you are running on. As a general guideline, specify values smaller than those listed in the formulas. However, if a lower limit is specified in the syntax (only for several *naux1* arguments in the Fourier transform, convolution, and correlation subroutines), you should not go below that limit. The ESSL error monitor returns the necessary sizes of the *aux* storage areas in the input-argument error message. This does, however, terminate your program when the error is encountered. (If you accidentally specify a sufficient amount of storage for the ESSL subroutine to perform the computation, error handling does not issue an error message and processing continues normally.) Figure 1 on page 35 illustrates what happens when error 2015 is unrecoverable.



* This check applies only to several NAUX1 arguments in the Fourier transform, convolution, and correlation subroutines.

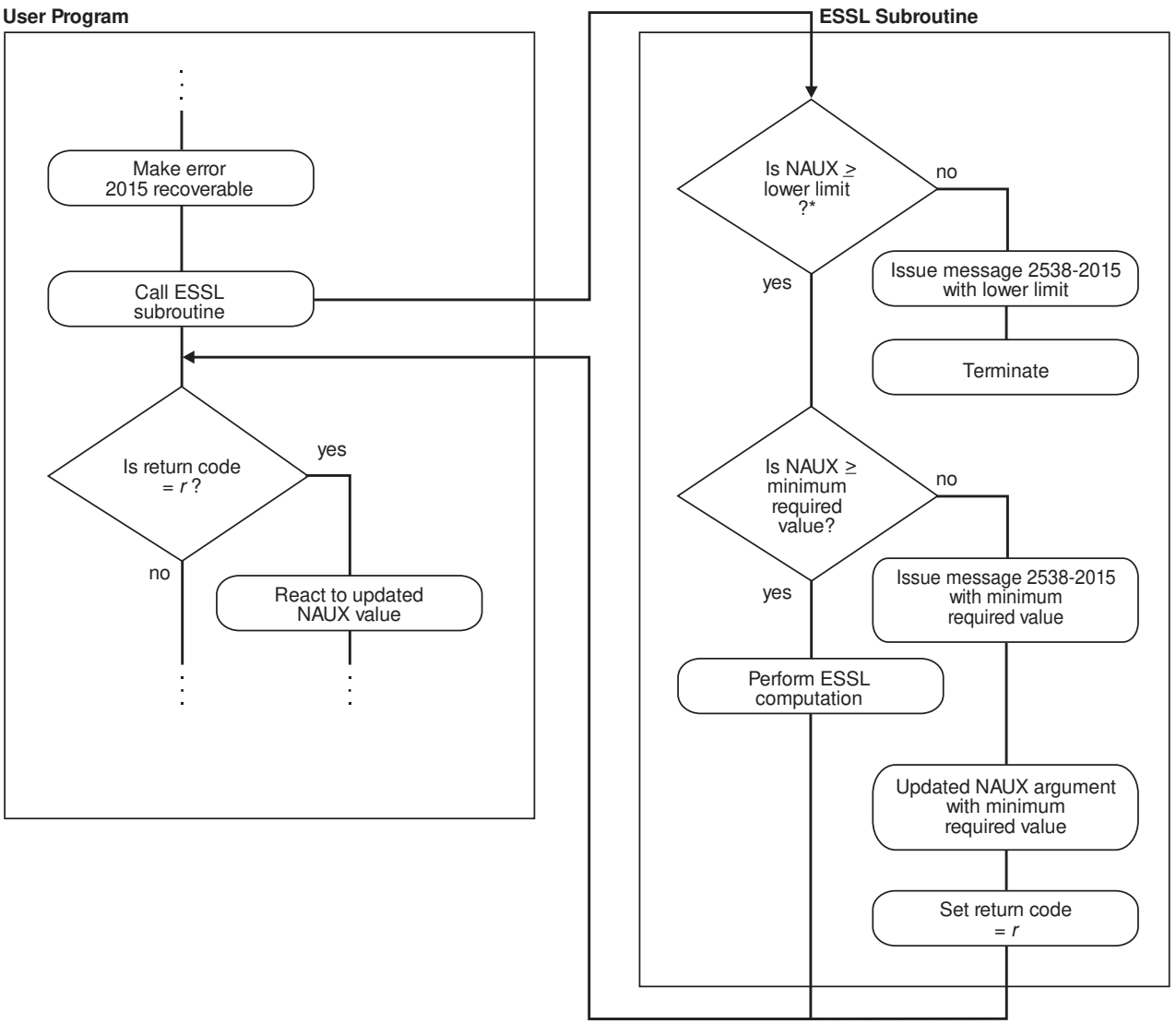
Figure 1. How to Obtain an NAUX Value from an Error Message, but Terminate

The Second Way

In this case, you obtain the minimum required value of *naux* from the error message and from the updated *naux* argument returned to your program.

Use EINFO and ERRSET with an ESSL error exit routine, ENOTRM, to make error 2015 recoverable. This allows you to dynamically determine in your program the minimum sizes required for the auxiliary working storage areas, specified in the *naux* arguments. Run your program with the *naux* values smaller than required by the subroutine for the particular processor you are running on. As a general guideline, specify values smaller than those listed in the formulas. However, if a lower limit is specified in the syntax (only for several *naux1* arguments in the Fourier transform, convolution, and

correlation subroutines), you should not go below that limit. The ESSL error monitor returns the necessary sizes of the *aux* storage areas in the input-argument error message and a return code is passed back to your program, indicating that updated values are also returned in the *naux* arguments. You can then react to these updated values during run time in your program. ESSL does not perform any computation when this error occurs. For details on how to do this, see Chapter 4 on page 111. (If you accidentally specify a sufficient amount of storage for the ESSL subroutine to perform the computation, error handling does not issue an error message and processing continues normally.) Figure 2 illustrates what happens when error 2015 is recoverable.



* This check applies only to several NAUX1 arguments in the Fourier transform, convolution, and correlation subroutines.

Figure 2. How to Obtain an NAUX Value from an Error Message and in Your Program

Here Is an Example of What Happens When You Use These Two Techniques

The following example illustrates all the actions taken by the ESSL error-handling facility for each possible value of a recoverable input argument, *naux*. A key point here is that if you want to have the updated argument value returned to your program, you must make error 2015 recoverable and then specify an *naux* value greater than or equal to 20 and less than 300. For values out of that range, the error recovery facility is not in effect. (These values of *naux*, 20 and 300, are used only for the purposes of this example and do not relate to any of the ESSL subroutines.)

NAUX **Meaning of the NAUX Value**

20 Lower limit of *naux* required for using recoverable input-argument error-handling facilities in ESSL. (This applies only to several *naux1* arguments in the Fourier transform, convolution, and correlation subroutines. You can find the lower limit in the syntax description for the *naux1* argument. For a list of subroutines, see Table 22 on page 32.)

300 Minimum value of *naux*, required for successful running (on the processor the program is being run on).

Table 23 describes the actions taken by ESSL in every possible situation for the values given in this example.

Table 23. Example of Input-Argument Error Recovery for Auxiliary Storage Sizes

NAUX Value	Action When 2015 Is an Unrecoverable Input-Argument Error	Action When 2015 Is a Recoverable Input-Argument Error
naux < 20	An input-argument error message is issued. The value in the error message is the lower limit, 20. The application program stops.	An input-argument error message is issued. The value in the error message is the lower limit, 20. The application program stops.
20 ≤ *naux* < 300	An input-argument error message is issued. The value in the error message is the minimum required value, 300. The application program stops.	ESSL returns the value of *naux* as 300 to the application program, and an input-argument error message is issued. The value in the error message is the minimum required value, 300. ESSL does no computation, and control is returned to the application program.
naux ≥ 300	Your application program runs successfully.	Your application program runs successfully.

Here Is How You Code It in Your Program

If you leave error 2015 unrecoverable, you **do not code anything** in your program. You just look at the error messages to get the sizes of auxiliary storage. On the other hand, if you want to make error 2015 recoverable to obtain the auxiliary storage sizes dynamically in your program, you need to **add some coding statements** to your program. For details on coding these statements in each programming language, see the following examples:

- For Fortran, see page 121
- For C, see page 139
- For C++, see page 156

You may want to provide a separate subroutine to calculate the auxiliary storage size whenever you need it. Figure 3 on page 38 shows how you might code a separate Fortran subroutine. Before calling SCFT in your program, call this subroutine, SCFT which calculates the minimum size and stores it in the *naux* arguments. Upon return, your program checks the return code. If it is nonzero, the *naux* arguments were updated, as planned. You should then make sure adequate storage is available and call SCFT. On the other hand, if the return code is zero, error handling was not invoked, the *naux* arguments were not updated, and the initialization step was performed for SCFT.

```
      SUBROUTINE SCFT
     * N, M, ISIGN, SCALE, AUX1, NAUX1,AUX2,NAUX2)
      REAL*4 X(0:*),Y(0:*),SCALE
      REAL*8 AUX1(7),AUX2(0:*)
      INTEGER*4 INIT,INC1X,INC2X,INC1Y,INC2Y,N,M,ISIGN,NAUX1,NAUX2
      EXTERNAL ENOTRM
      CHARACTER*8   S2015
          CALL EINFO(0)
          CALL ERRSAV(2015,S2015)
          CALL ERRSET(2015,0,-1,1,ENOTRM,0)
C     SETS NAUX1 AND NAUX2 TO THE MINIMUM VALUES REQUIRED TO USE
C     THE RECOVERABLE INPUT-ARGUMENT ERROR-HANDLING FACILITY
          NAUX1 = 7
          NAUX2 = 0
          CALL SCFT(INIT,X,INC1X,INC2X,Y,INC1Y,INC2Y,
     *              N,M,ISIGN,SCALE,AUX1,NAUX1,AUX2,NAUX2,*10)
          CALL ERRSTR(2015,S2015)
          RETURN
10        CONTINUE
          CALL ERRSTR(2015,S2015)
          RETURN 1
          END
```

Figure 3. Fortran Subroutine to Calculate Auxiliary Storage Sizes

Providing a Correct Transform Length to ESSL

This section describes how to calculate the length of your transform by use of formulas or error-handling capabilities provided in ESSL.

What ESSL Subroutines Require Transform Lengths?

For the ESSL subroutines listed in Table 24 on page 39, you need to provide one or more transform lengths for the computation of a Fourier transform. These transform lengths are supplied through one or more arguments, such as *n*, *n1*, *n2*, and *n3*, in the calling sequence for the ESSL subroutine. Only certain lengths of transforms are permitted in the computation.

Table 24. ESSL Subroutines Requiring Transform Lengths
Subroutine Names
Fourier Transforms: _CFT _RCFT _CRFT _COSF _SINF SCOSFT _CFT2 _RCFT2 _CRFT2 _CFT3 _RCFT3 _CRFT3 SCFTP SCFT2P SCFT3P

Who Do You Want to Calculate the Length? You or ESSL?

You have a choice of two methods for determining an acceptable length for your transform to be processed by ESSL:

- Use the formula or large table in "Acceptable Lengths for the Transforms" on page 741 to determine an acceptable length. For details, see "How Do You Calculate the Length Using the Table or Formula?."
- Use the ESSL error-handling facilities to return to you an acceptable length. For details, see "How Do You Get ESSL to Calculate the Length Using ESSL Error Handling?."

How Do You Calculate the Length Using the Table or Formula?

The lengths ESSL accepts for transforms in the Fourier transform subroutines are listed in "Acceptable Lengths for the Transforms" on page 741. You should use the table in that section to find the two values your length falls between. You then specify the **larger** length for your transform. If you find a perfect match, you can use that value without having to change it. The formula provided in that section expresses how to calculate the acceptable values listed in the table. If necessary, you can use the formula to dynamically check lengths in your program.

How Do You Get ESSL to Calculate the Length Using ESSL Error Handling?

This section describes how to get ESSL to calculate transform lengths.

Here Are the Two Ways You Can Do It

Ask yourself which of the following ways you prefer to obtain the information from ESSL:

- **By leaving error 2030 unrecoverable,** you can obtain an acceptable value for *n* from the input-argument error message, but your program terminates.
- **By making error 2030 recoverable,** you obtain an acceptable value for *n* from the input-argument error message and have the updated *n* argument returned to your program.

Because the Fourier transform subroutines allow only certain lengths for transforms, ESSL provides this error-handling capability to return acceptable lengths to your program. It returns them in the transform length arguments. The value ESSL returns is the **next larger acceptable length** for a transform, based on the length you specify in the *n* argument.

The First Way

In this case, you obtain an acceptable value of *n* from the error message, but your program terminates.

Leave error 2030 as unrecoverable, without calls to EINFO and ERRSET. Run your program with a close approximation of the transform length you want to use. If this happens not to be an acceptable length, the ESSL error monitor returns an acceptable length of the transform in input-argument error message. This does, however, terminates your program when the error is encountered. (If you do happen to specify an acceptable length for the transform, error handling does not issue an error message and processing continues normally.) Figure 4 illustrates what happens when error 2030 is unrecoverable.

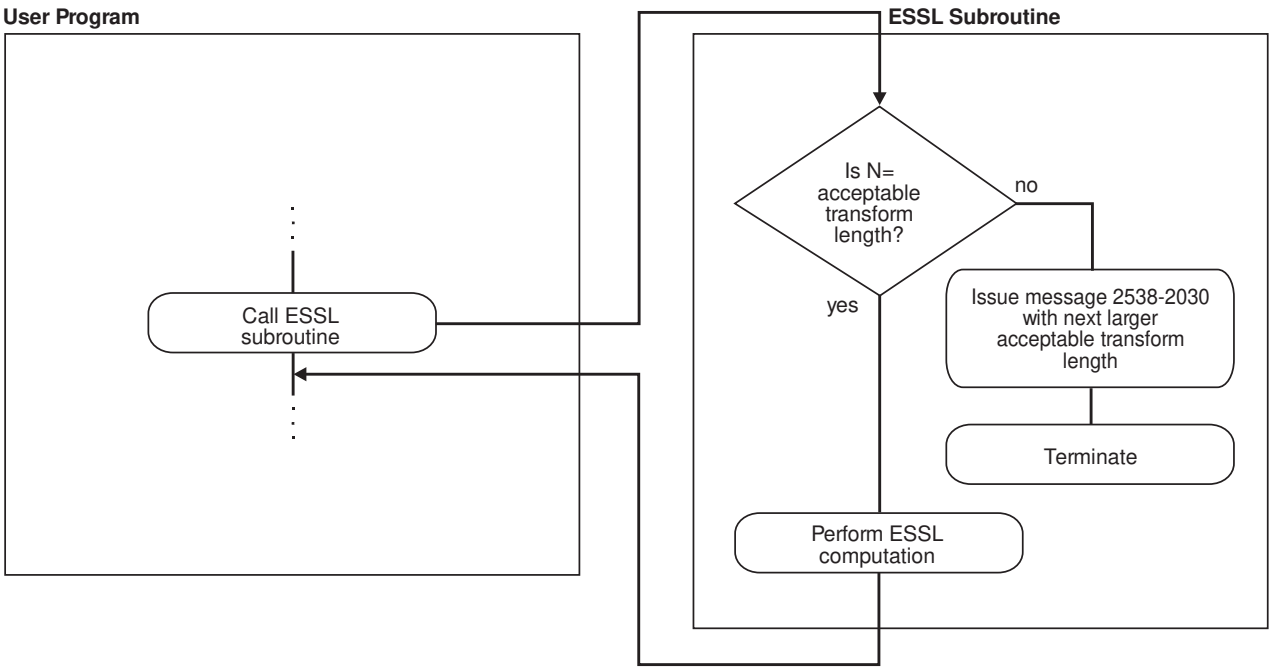


Figure 4. How to Obtain an N Value from an Error Message, but Terminate

The Second Way

In this case, you obtain an acceptable value of *n* from the error message and from the updated *n* argument returned to your program.

Use EINFO and ERRSET with an ESSL error exit routine, ENOTRM, to make error 2030 recoverable. This allows you to dynamically determine in your program an acceptable length for your transform, specified in the *n* argument(s). Run your program with a close approximation of the transform length you want to use. If this happens not to be an acceptable length, the ESSL error monitor returns an acceptable length of the transform in the input-argument error message and a return code is passed back to your program, indicating that updated values are also returned in the *n* argument(s). You can then react to these updated values during run time in your program. ESSL does not perform any computation when this error occurs. For details on how to do this, see Chapter 4 on page 111. (If you do happen to specify an acceptable length for the transform, error handling does not issue an error message and processing continues normally.) Figure 5 on page 41 illustrates what happens when error 2030 is recoverable.

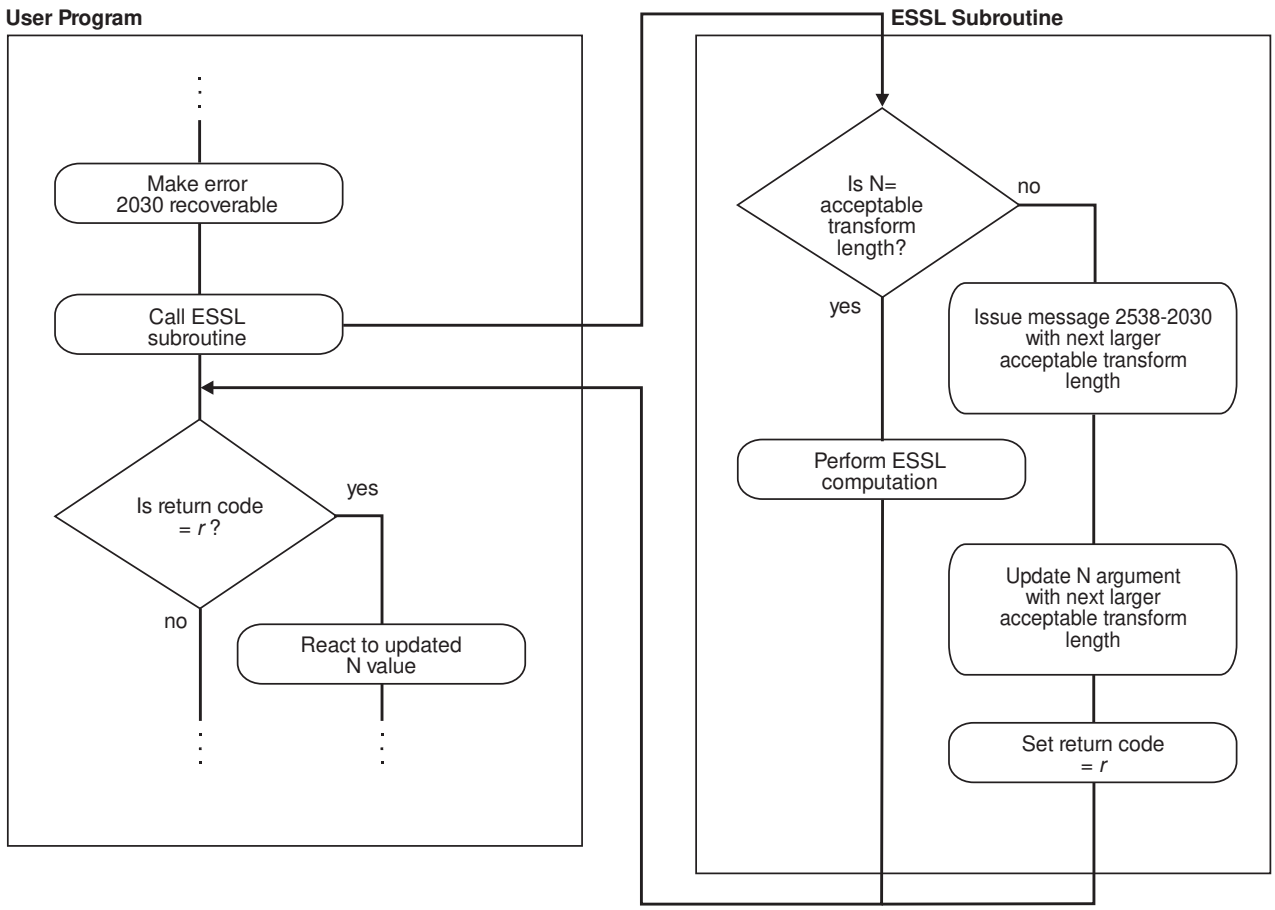


Figure 5. How to Obtain an N Value from an Error Message and in Your Program

Here Is an Example of What Happens When You Use These Two Techniques

The following example illustrates all the actions taken by the ESSL error-handling facility for each possible value of a recoverable input argument, *n*. The values of *n* used in the example are as follows:

N	Meaning of the N Value
7208960	An acceptable transform length, required for successful computing of a Fourier transform
7340032	The next larger acceptable transform length, required for successful computing of a Fourier transform

Table 25 on page 42 describes the actions taken by ESSL in every possible situation for the values given in this example.

Table 25. Example of Input-Argument Error Recovery for Transform Lengths

N Value	Action When 2030 Is an Unrecoverable Input-Argument Error	Action When 2030 Is a Recoverable Input-Argument Error
n = 7208960 **–or–** *n* = 7340032	Your application program runs successfully.	Your application program runs successfully.
7208960 < *n* < 7340032	An input-argument error message is issued. The value in the error message is 7340032. The application program stops.	ESSL returns the value of *n* as 7340032 to the application program, and an input-argument error message is issued. The value in the error message is 7340032. ESSL does no computation, and control is returned to the application program.

Here Is How You Code It in Your Program

If you leave error 2030 unrecoverable, you **do not code anything** in your program. You just look at the error messages to get the transform lengths. On the other hand, if you want to make error 2030 recoverable to obtain the transform lengths dynamically in your program, you need to **add some coding statements** to your program. For details on coding these statements in each programming language, see the following examples:

- For Fortran, see page 121
- For C, see page 139
- For C++, see page 156

You may want to provide a separate subroutine to calculate the transform length whenever you need it. Figure 6 on page 43 shows how you might code a separate Fortran subroutine. Before calling SCFT in your program, you call this subroutine, SCFT which calculates the correct length and stores it in *n*. Upon return, your program checks the return code. If it is nonzero, the *n* argument was updated, as planned. You then do any necessary data setup and call SCFT. On the other hand, if the return code is zero, error handling was not invoked, the *n* argument was not updated, and the initialization step was performed for SCFT.

```
      SUBROUTINE SCFT
     * N, M, ISIGN, SCALE, AUX1, NAUX1,AUX2,NAUX2)
      REAL*4 X(0:*),Y(0:*),SCALE
      REAL*8 AUX1(7),AUX2(0:*)
      INTEGER*4 INIT,INC1X,INC2X,INC1Y,INC2Y,N,M,ISIGN,NAUX1,NAUX2
      EXTERNAL ENOTRM
      CHARACTER*8   S2030
          CALL EINFO(0)
          CALL ERRSAV(2030,S2030)
          CALL ERRSET(2030,0,-1,1,ENOTRM,0)
          CALL SCFT(INIT,X,INC1X,INC2X,Y,INC1Y,INC2Y,
     *              N,M,ISIGN,SCALE,AUX1,NAUX1,AUX2,NAUX2,*10)
          CALL ERRSTR(2030,S2030)
          RETURN
10        CONTINUE
          CALL ERRSTR(2030,S2030)
          RETURN 1
          END
```

Figure 6. Fortran Subroutine to Calculate Transform Length

You might want to combine the request for auxiliary storage sizes along with your request for transform lengths. Figure 7 shows how you might code a separate Fortran subroutine combining both requests. It combines the functions performed by the subroutines in Figure 3 on page 38 and Figure 6.

```
      SUBROUTINE SCFT
     * N, M, ISIGN, SCALE, AUX1, NAUX1,AUX2,NAUX2)
      REAL*4 X(0:*),Y(0:*),SCALE
      REAL*8 AUX1(7),AUX2(0:*)
      INTEGER*4 INIT,INC1X,INC2X,INC1Y,INC2Y,N,M,ISIGN,NAUX1,NAUX2
      EXTERNAL ENOTRM
      CHARACTER*8   S2015,S2030
          CALL EINFO(0)
          CALL ERRSAV(2015,S2015)
          CALL ERRSAV(2030,S2030)
          CALL ERRSET(2015,0,-1,1,ENOTRM,0)
          CALL ERRSET(2030,0,-1,1,ENOTRM,0)
C     SETS NAUX1 AND NAUX2 TO THE MINIMUM VALUES REQUIRED TO USE
C     THE RECOVERABLE INPUT-ARGUMENT ERROR-HANDLING FACILITY
          NAUX1 = 7
          NAUX2 = 0
          CALL SCFT(INIT,X,INC1X,INC2X,Y,INC1Y,INC2Y,
     *              N,M,ISIGN,SCALE,AUX1,NAUX1,AUX2,NAUX2,*10)
          CALL ERRSTR(2015,S2015)
          CALL ERRSTR(2030,S2030)
          RETURN
10        CONTINUE
          CALL ERRSTR(2015,S2015)
          CALL ERRSTR(2030,S2030)
          RETURN 1
          END
```

Figure 7. Fortran Subroutine to Calculate Auxiliary Storage Sizes and Transform Length

Getting the Best Accuracy

This section explains how accuracy of your results can be affected in various situations and what you can do to achieve the best possible accuracy.

What Precisions Do ESSL Subroutines Operate On?

Both short- and long-precision real versions of the subroutines are provided in most areas of ESSL. In some areas, short- and long-precision complex versions are also provided, and, occasionally, an integer version is provided. The subroutine names are distinguished by a one- or two-letter prefix based on the following letters:

S for short-precision real
D for long-precision real
C for short-precision complex
Z for long-precision complex
I for integer

For a description of these data types, see "How Do You Set Up Your Scalar Data?" on page 28. The scalar data types and how you should code them for each programming language are listed under "Coding Your Scalar Data" in each language section in Chapter 4 on page 111.

How Does the Nature of the ESSL Computation Affect Accuracy?

In subroutines performing operations such as copy and swap, the accuracy of data is not affected. In subroutines performing computations involving mathematical operations on array data, the accuracy of the result may be affected by the following:

- The algorithm, which can vary depending on values or array sizes within the computation or the number of threads used.
- The matrix and vector sizes

For this reason, the ESSL subroutines do **not** have a closed formula for the error of computation. In other words, there is no formula with which you can calculate the error of computation in each subroutine.

What Does ESSL Do to Improve Accuracy?

Short-precision subroutines sometimes provide increased accuracy of results by accumulating intermediate results in long precision. This is also noted in the functional description for each subroutine. See the *RS/6000 POWERstation and POWERserver Hardware Technical Reference—General Information* manual for details.

What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?

The data types operated on by the short-precision, long-precision, and integer versions of the subroutines are ANSI/IEEE 32-bit and 64-bit binary floating-point format, and 32-bit integer. See the *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985* for more detail.

There are ESSL-specific rules that apply to the results of computations using the ANSI/IEEE standards. When running your program, the result of a multiplication of NaN ("Not-a-Number") by a scalar zero, under certain circumstances, may differ in the ESSL subroutines from the result you expect.

Usually, when NaN is multiplied by a scalar zero, the result is NaN; however, in some ESSL subroutines where scaling is performed, the result may be zero. For example, in computing $\alpha \boldsymbol{A}$, where α is a scalar and $\boldsymbol{A}$ is a matrix, if α is zero and one (or more) of the elements of $\boldsymbol{A}$ is NaN, the scaled result, using that element, may be a zero, rather than NaN. To avoid problems, you should consider this when designing your program.

What Does ESSL Do to Improve Accuracy?

For the RS/6000 POWER and POWER2, the short-precision, floating-point operands are stored by the hardware in the floating-point registers as long-precision values, and, as a result, all arithmetic operations are performed in long-precision. Where applicable, the ESSL subroutines use the Multiply-Add instructions, which combine a Multiply and Add operation without an intermediate rounding operation.

How is Underflow Handled?

ESSL does not mask underflow. If your program incurs a number of unmasked underflows, its overall performance decreases. **For the RS/6000, floating-point exception trapping is disabled by default. Therefore, you do not have to mask underflow unless you have changed the default.**

Where Can You Find More Information on Accuracy?

Information about accuracy can be found in the following places:

- Migration considerations concerning accuracy of results between releases, platforms, and so forth are described in Chapter 6 on page 169.
- Specific information on accuracy for each area of ESSL is given in "Performance and Accuracy Considerations" in each chapter introduction in Part 2.
- The functional description under "Function" for each subroutine explains what you need to know about the accuracy of the computation. Varying implementation techniques are sometimes used to improve performance. To let you know how accuracy is affected, the functional description may explain in general terms the different techniques used in the computation.

Getting the Best Performance

This section describes how you can achieve the best possible performance from the ESSL subroutines.

What General Coding Techniques Can You Use to Improve Performance?

There are many ways in which you can improve the performance of your program. Here are some of them:

- Use the basic linear algebra subprograms and matrix operations in the order of optimum performance: matrix-matrix computations, matrix-vector computations, and vector-scalar computations. When data is presented in matrices or vectors, rather than vectors or scalars, multiple operations can be performed by a single ESSL subroutine.
- Where possible, use subroutines that do multiple computations, such as SNDOT and SNAXPY, rather than individual computations, such as SDOT and SAXPY.
- Use a stride of 1 for the data in your computations. Not having vector elements consecutively accessed in storage can degrade your performance. The closer the

vector elements are to each other in storage, the better your performance. For an explanation of stride, see “How Stride Is Used for Vectors” on page 58.

- Do **not** specify the size of the leading dimension of an array (*lda*) or stride of a vector (*inc*) equal to or near a multiple of:
  - 128 for a long-precision array
  - 256 for a short-precision array
- Do **not** specify the individual sizes of your one-dimensional arrays as multiples of 128. This is especially important when you are passing several one-dimensional arrays to an ESSL subroutine. (The multiplicity can cause a performance problem that otherwise might not occur.)
- For small problems, avoid using a large leading dimension (*lda*) for your matrix.
- In general, align your arrays on doubleword boundaries, regardless of the type of data; however, when running on a POWER2 processor, it is best to align your long-precision arrays on a quadword boundary. For information on how your programming language aligns data, see your programming language manuals.
- One subroutine may do scaling while another does not. If scaling is not necessary for your data, you get better performance by using the subroutine without scaling. SNORM2 and DNORM2 are examples of subroutines that do not do scaling, versus SNRM2 and DNRM2, which do scaling.
- Use the STRIDE subroutine to calculate the optimal stride values for your input or output data when using any of the Fourier transform subroutines, except _RCFT and _CRFT. Using these stride values for your data allows the Fourier transform subroutines to achieve maximum performance. You first obtain the optimal stride values from STRIDE, calling it once for each stride value desired. You then arrange your data using these stride values. After the data is set up, you call the Fourier transform subroutine. For details on the STRIDE subroutine and how to use it for each Fourier transform subroutine, see “STRIDE—Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines” on page 977. For additional information, see “Setting Up Your Data” on page 744.

Where Can You Find More Information on Performance?

Information about performance can be found in the following places:

- Many of the techniques ESSL uses to achieve the best possible performance are described in the “High Performance of ESSL” on page 6.
- Migration considerations concerning performance are described in “Migrating ESSL Version 2 Programs to Version 3” on page 169.
- Specific information on performance for each area of ESSL is given in “Performance and Accuracy Considerations” in each chapter introduction in Part 2.
- Detailed performance information for selected subroutines can be found in reference [30], [41], [42] and on the IBM RS/6000 web site at `http://www.rs6000.ibm.com/software/Apps/essl.html`.

Dealing with Errors when Using ESSL

At run time, you can encounter different types of errors or messages that are related to the use of the ESSL subroutines:

- Program exceptions
- ESSL input-argument errors
- ESSL computational errors
- ESSL resource errors
- ESSL attention messages

This section explains how to handle all these situations.

What Can You Do about Program Exceptions?

The program exceptions you can encounter in ESSL are described in the RS/6000 architecture manuals. For details, see:

- The *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*
- The *RS/6000 POWERstation and POWERserver Hardware Technical Reference—General Information* manual

What Can You Do about ESSL Input-Argument Errors?

This section gives an overview on how you can handle input-argument errors.

All Input-Argument Errors

ESSL checks the validity of most input arguments. If it finds that any are invalid, it issues the appropriate error messages. Also, except for the two recoverable errors described below, it terminates your program. You should use standard programming techniques to diagnose and fix unrecoverable input-argument errors, as described in Chapter 7 on page 173.

You can determine the input-argument errors that can occur in a subroutine by looking under "Error Conditions" in the subroutine description in Part 2 of this book. Error messages for all input-argument errors are listed in "Input-Argument Error Messages" on page 179.

Recoverable Errors 2015 and 2030 Can Return Updated Values in the NAUX and N Arguments

For two input-argument errors, 2015 and 2030, in Fortran, C, C++, and PL/I programs, you have the option to continue running and have an updated value of the input argument returned to your program for subsequent use. These are called recoverable errors. This recoverable error-handling capability gives you flexibility in determining the correct values for the arguments. You can:

- Determine the correct size of an auxiliary work area by using error 2015. For help in deciding whether you want to use this capability and details on how to use it, see "Using Auxiliary Storage in ESSL" on page 31.
- Determine the correct length of a transform by using error 2030. For help in deciding whether you want to use this capability and details on how to use it, see "Providing a Correct Transform Length to ESSL" on page 38.

If you chose to leave errors 2015 and 2030 unrecoverable, you do not need to make any coding changes to your program. The input-argument error message is issued upon termination, containing the updated values you could have specified for the program to run successfully. You then make the necessary corrections in your program and rerun it.

If you choose to make errors 2015 and 2030 recoverable, you call the ERRSET subroutine to set up the ESSL error exit routine, ENOTRM, and then call the ESSL subroutine. When one or more of these errors occurs, the input-argument error message is issued with the updated values. In addition, the updated values are returned to your program in the input arguments named in the error message, along with a nonzero return code and processing continues. Return code values associated with these recoverable errors are described under "Error Conditions" for each ESSL subroutine in Part 2.

For details on how to code the necessary statements in your program to make 2015 and 2030 recoverable, see the following sections:

- "Input-Argument Errors in Fortran" on page 119
- "Input-Argument Errors in C" on page 136
- "Input-Argument Errors in C++" on page 153

What Can You Do about ESSL Computational Errors?

This section gives an overview on how you can handle computational errors.

All Computational Errors

ESSL computational errors are errors occurring in the computational data, such as in your vectors and matrices. You can determine the computational errors that can occur in a subroutine by looking under "Error Conditions" in the subroutine description in Part 2 of this book. These errors cause your program to terminate abnormally unless you take preventive action. A message is also provided in your output, containing information about the error. Messages are listed in "Computational Error Messages" on page 187.

When a computational error occurs, you should assume that the results are unpredictable. The result of the computation is valid only if no errors have occurred. In this case, a zero return code is returned.

Figure 8 on page 49 shows what happens when a computational error occurs.

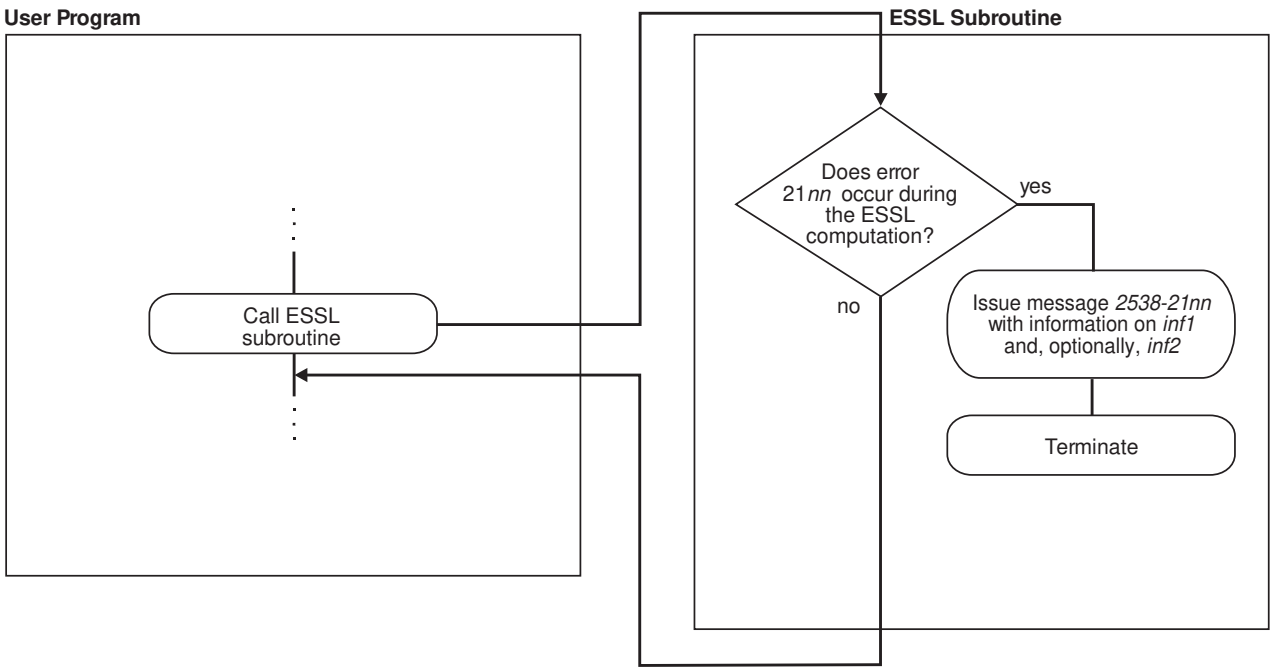


Figure 8. How to Obtain Computational Error Information from an Error Message, but Terminate

Recoverable Computational Errors Can Return Values Through EINFO

In Fortran, C, C++, and PL/I programs, you have the capability to make certain computational errors recoverable and have information returned to your program about the errors. Recoverable computational errors are listed in Table 167 on page 968. First, you call EINFO in the beginning of your program to initialize the ESSL error option table. You then call ERRSET to reset the number of allowable errors for the computational error codes in which you are interested. When a computational error occurs, a nonzero return code is returned for each computational error. Return code values associated with these errors are described under "Error Conditions" in each subroutine description. Based on the return code, your program can branch to an appropriate statement to call the ESSL error information-handler subroutine, EINFO, to obtain specific information about the data involved in the error. This information is returned in the EINFO output arguments, *inf1* and, optionally, *inf2*. You can then check the information returned and continue processing, if you choose. The syntax for EINFO is described under "EINFO—ESSL Error Information-Handler Subroutine" on page 968. You also get a message in your output for each computational error encountered, containing information about the error. The EINFO subroutine provides the same information in the messages as it provides to your program.

For details on how to code the necessary statements in your program to obtain specific information on computational errors, see the following sections:

- "Computational Errors in Fortran" on page 122
- "Computational Errors in C" on page 141
- "Computational Errors in C++" on page 157

Figure 9 on page 50 shows what happens if you make a computational error recoverable.

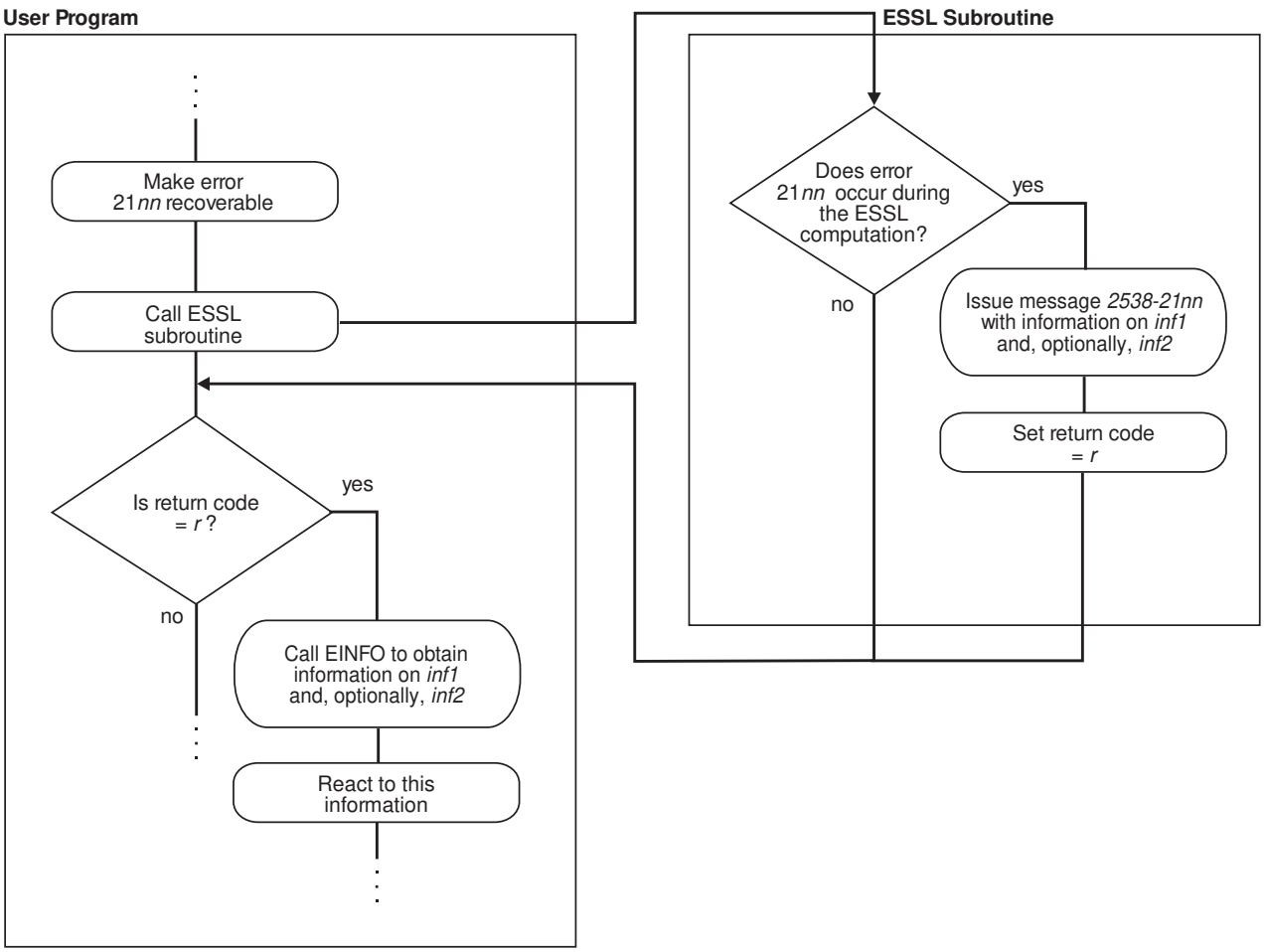


Figure 9. How to Obtain Computational Error Information in an Error Message and in Your Program

What Can You Do about ESSL Resource Errors?

This section gives an overview on how you can handle resource errors.

All Resource Errors

ESSL returns a resource error and terminates your program when an attempt to allocate work area fails. Some ESSL subroutines attempt to allocate work area for their internal use. Other ESSL subroutines attempt to dynamically allocate auxiliary storage when a user requests it through calling sequence arguments, such as *aux* and *naux*. For information on how you could reduce memory constraints on the system or increase the amount of memory available before rerunning the application program, see "ESSL Resource Error Messages" on page 176.

You can determine the resource errors that can occur in a subroutine by looking under "Error Conditions" in the subroutine description in Part 2 of this book. Error messages for all resource errors are listed in "Resource Error Messages" on page 190.

What Can You Do about ESSL Attention Messages?

This section gives an overview on how you can handle attention messages.

All Attention Messages

ESSL returns an attention message to describe a condition that occurred, however, ESSL is able to continue processing. For information on how you could reduce memory constraints on the system or increase the amount of memory available, see "ESSL Resource Error Messages" on page 176.

For example, an attention message may be issued when enough work area was available to continue processing, but was not the amount initially requested. An attention message would be issued to indicate that performance may be degraded.

For a list of subroutines that may generate an attention message, see Table 31 on page 177. For a list of attention messages, see "Informational and Attention Error Messages" on page 190.

How Do You Control Error Handling by Setting Values in the ESSL Error Option Table?

This section explains all aspects of using the ESSL error option table.

What Values Are Set in the ESSL Error Option Table?

The ESSL error option table contains information that tells ESSL what to do every time it encounters an ESSL-generated error. Table 26 shows the default values established in the table when ESSL is installed.

Table 26 (Page 1 of 2). ESSL Error Option Table Default Values

Range of Error Messages (From–To)	Number of Allowable Errors (ALLOW)	Number of Messages Printed (PRINT)	Modifiable Table Entry (MODENT)
2538–2000	Unlimited	255	NO
2538–2001 through 2538–2073	Unlimited	255	YES
2538–2074	Unlimited	5	YES
2538–2075 through 2538–2098	Unlimited	255	YES
2538–2099	1	255	YES
2538–2100 through 2538–2101	1	255	YES
2538–2102	Unlimited	255	YES
2538–2103 through 2538–2113	1	255	YES
2538–2114	Unlimited	255	YES
2538–2115 through 2538–2122	1	255	YES
2538–2123 through 2538–2124	Unlimited	255	YES
2538–2125 through 2538–2126	1	255	YES
2538–2127	Unlimited	255	YES
2538–2128 through 2538–2137	1	255	YES

Table 26 (Page 2 of 2). ESSL Error Option Table Default Values

Range of Error Messages (From–To)	Number of Allowable Errors (ALLOW)	Number of Messages Printed (PRINT)	Modifiable Table Entry (MODENT)
2538–2138 through 2538–2143	Unlimited	255	YES
2538–2144 through 2538–2145	1	255	YES
2538–2146	Unlimited	255	YES
2538–2147 through 2538–2198	1	255	YES
2538–2199	1	255	YES
2538–2400 through 2538–2499	1	255	NO
2538–2600 through 2538–2699	Unlimited	255	NO
2538–2700 through 2538–2799	1	255	NO

How Can You Change the Values in the Error Option Table?

You can change any of the values in the ESSL error option table by calling the ERRSET subroutine in your program. This dynamically changes values at run time. You can also save and restore entries in the table by using the ERRSAV and ERRSTR subroutines, respectively. For a description of the ERRSET, ERRSAV, and ERRSTR subroutines see Chapter 17 on page 965.

When Do You Change the Values in the Error Option Table?

Because you can change the information in the error option table, you can control what happens when any of the ESSL errors occur. There are a number of instances when you may want to do this:

To Customize Your Error-Handling Environment: You may simply want to adjust the number of times an error is allowed to occur before your program terminates. You can use any of the capabilities available in ERRSET.

To Obtain Auxiliary Storage Sizes and Transform Lengths: You may want to make ESSL input-argument error 2015 or 2030 recoverable, so ESSL returns updated auxiliary storage sizes or transform lengths, respectively, to your program. For a more detailed discussion, see "What Can You Do about ESSL Input-Argument Errors?" on page 47. For how to use ERRSET to do this, see the section for your programming language in Chapter 4 on page 111.

To Get More Information About a Computational Error: You may want ESSL to return information about a computational error to your program. For a more detailed discussion, see "What Can You Do about ESSL Computational Errors?" on page 48. For how to do use ERRSET to do this, see the section for your programming language in Chapter 4 on page 111.

To Allow Parts of Your Application to Have Unique Error-Handling Environments: If your program is part of a large application, you may want to dynamically save and restore entries in the error option table that have been altered by ERRSET. This ensures the integrity of the error option table when it is used by multiple programs within an application. For a more detailed discussion, see " How Can You

Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?" on page 53 For how to use ERRSAV and ERRSTR, see the section for your programming language in Chapter 4 on page 111.

How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?

When your program is part of a larger application, you should consider that one of the following can occur:

- If you use ERRSET in your program to reset any of the values in the error option table for any of the ESSL input-argument errors or computational errors, some other program in the application may be adversely affected. It may be expecting its original values.
- If some other program in the application uses ERRSET to reset any of the values in the error option table for any of the ESSL input-argument errors or computational errors, your program may be adversely affected. You may need a certain value in the error option table, and the application may have reset that value.

These situations can be avoided if every program that uses ERRSET, in the large application, also uses the ERRSAV and ERRSTR facilities. For a particular error number, ERRSAV saves an entry from the error option table in an area accessible to your program. ERRSTR then stores the entry back into the error option table from the storage area. You code an ERRSAV and ERRSTR for each input-argument error number and computational error number for which you do an ERRSET to reset the values in the error option table. Call ERRSAV at the beginning of your program after you call EINFO, and then call ERRSTR at the end of your program after all ESSL computations are completed. This saves the original contents of the error option table while your program is running with different values, and then restores it to its original contents when your program is done. For details on how to code these statements in your program, see Chapter 4 on page 111.

How does Error Handling Work in a Threaded Environment?

When your application program or Fortran first creates a thread, ESSL initializes the error option table information to the default settings shown in Table 26 on page 51. You can change the default settings for each thread you created by calling the appropriate error handling subroutines (ERRSET, ERRSAV, or ERRSTR) from each thread. An example of how to initialize the error option table and change the default settings on multiple threads is shown in "Example of Handling Errors in a Multithreaded Application Program" on page 127.

ESSL issues error messages as they occur in a threaded environment. Error messages issued from any of the existing threads are written to standard output in the order in which they occur.

When a terminating condition occurs on any of the the existing threads (for example, the number of allowable errors was exceeded), ESSL terminates your application program.

Where Can You Find More Information on Errors?

Information about errors and how to handle them can be found in the following places:

- How to code your program to use the ESSL error-handling facilities is described in Chapter 4 on page 111.
- All ESSL error messages are listed under "Messages" on page 178.
- The errors and return codes associated with each ESSL subroutine are listed under "Error Conditions" in each subroutine description in Part 2.
- Complete diagnostic procedures for all types of ESSL programming and documentation problems, along with how to collect information and report a problem, are provided in Chapter 7 on page 173.

Chapter 3. Setting Up Your Data Structures

This chapter provides you with information that you need to set up your data structures, consisting of vectors, matrices, and sequences. These techniques apply to programs in all programming languages.

Concepts

Vectors, matrices, and sequences are conceptual data structures contained in arrays. In many cases, ESSL uses stride or leading dimension to select the elements of the vector, matrix, or sequence from an array. In other cases, ESSL uses a specific mapping, or storage layout, that identifies the elements of the vector, matrix, or sequence in an array, sometimes requiring several arrays to help define the mapping. These elements selected from the array(s) make up the conceptual data structure.

When you call an ESSL subroutine, it assumes that the data structure is set up properly in the array(s) you pass to it. If it is not, your results are unpredictable. ESSL also uses these same storage layouts for data structures passed back to your program.

The use of the terms vector, matrix, and sequence in this book is consistent with standard mathematical definitions, and their representations are consistent with conventions used in mathematical texts. Special notations and conventions used in this book for describing vectors, matrices, and sequences are explained in "Special Notations and Conventions" on page xxii.

Overlapping Data Structures: Most of the subroutines do not allow vectors, matrices, or sequences to overlap. If this occurs, results are unpredictable. Where this applies, it is explained in Notes in each subroutine description. This means the elements of the data structure cannot reside in the same storage locations as any of the other data structures. It is possible, however, to have elements of different data structures in the same array, as long as the elements are interleaved through storage using strides greater than 1. For example, using vectors ***x*** and ***y*** with strides of 2, where ***x*** starts at `A(1)` and ***y*** starts at `A(2)`, the elements reside in array `A` in the order x_1, y_1, x_2, y_2, x_3, y_3, ... and so forth.

When you use this technique, you should be careful that you specify different starting locations for each data structure contained in the array.

Vectors

A vector is a one-dimensional, ordered collection of numbers. It can be a column vector, which represents an n by 1 ordered collection, or a row vector, which represents a 1 by n ordered collection.

© Copyright IBM Corp. 1997

The column vector appears symbolically as follows:

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ . \\ . \\ x_n \end{bmatrix}$$

A row vector appears symbolically as follows:

$$\boldsymbol{x} = \begin{bmatrix} x_1 \, x_2 \, x_3 \, \ldots \, x_n \end{bmatrix}$$

Vectors can contain either real or complex numbers. When they contain real numbers, they are sometimes called real vectors. When they contain complex numbers, they are called complex vectors.

Transpose of a Vector

The transpose of a vector changes a column vector to a row vector, or vice versa:

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ . \\ . \\ x_n \end{bmatrix} \qquad \boldsymbol{x}^{\mathrm{T}} = \begin{bmatrix} x_1 \; x_2 \; x_3 \ldots x_n \end{bmatrix} \qquad (\boldsymbol{x}^{\mathrm{T}})^{\mathrm{T}} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ . \\ . \\ x_n \end{bmatrix}$$

The ESSL subroutines use the vector as it is intended in the computation, as either a column vector or a row vector; therefore, no movement of data is necessary.

In the examples provided with the subroutine descriptions in Part 2 of this book, both types of vectors are represented in the same way, showing the elements of the array that make up the vector ***x***, as follows:

```
(1.0, 2.0, 3.0, 4.0, 5.0, 6.0)
```

Conjugate Transpose of a Vector

The conjugate transpose of a vector $\boldsymbol{x}$, containing complex numbers, is denoted by $\boldsymbol{x}^{\mathrm{H}}$ and is expressed as follows:

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ . \\ . \\ x_n \end{bmatrix} \qquad \boldsymbol{x}^{\mathrm{H}} = \begin{bmatrix} \bar{x}_1 & \bar{x}_2 & \bar{x}_3 \ldots \bar{x}_n \end{bmatrix}$$

Just as for the transpose of a vector, no movement of data is necessary for the conjugate transpose of a vector.

In Storage

A vector is usually stored within a one- or two-dimensional array. Its elements are stored sequentially in the array, but not necessarily contiguously.

The **location** of the vector in the array is specified by the argument for the vector in the ESSL calling sequence. It can be specified in a number of ways. For example, if A is an array of length 12, and you want to specify vector $\boldsymbol{x}$ as starting at the first element of array A, specify A as the argument, such as in:

```
X = SASUM (4,A,2)
```

where the number of elements to be summed in the vector is 4, the location of the vector is A, and the stride is 2.

If you want to specify vector $\boldsymbol{x}$ as starting at element 3 in array A, which is declared as A(1:12), specify:

```
X = SASUM (4,A(3),2)
```

If A is declared as A(-1:8), specify the following for element 3:

```
X = SASUM (4,A(1),2)
```

If A is a two-dimensional array and declared as A(1:4,1:10), and you want vector $\boldsymbol{x}$ to start at the second row and third column of A, specify the following:

```
X = SASUM (4,A(2,3),2)
```

The **stride** specified in the ESSL calling sequence is used to step through the array to select the vector elements. The direction in which the vector elements are selected from the array—that is, front to back or back to front—is indicated by the sign (+ or –) of the stride. The absolute value of the stride gives the spacing between each element selected from the array.

To calculate the total number of elements needed in an array for a vector, you can use the following formula, which takes into account the number of elements, *n*, in the array and the stride, *inc*, specified for the vector:

$$1+(n-1)|inc|$$

An array can be much larger than the vector that it contains; that is, there can be many elements following the vector in the array, as well as elements preceding the vector.

For a complete description of how vectors are stored within arrays, see "How Stride Is Used for Vectors."

For a complex vector, a special storage arrangement is used to accommodate the two parts, *a* and *b*, of each complex number (*a*+*bi*) in the array. For each complex number, two sequential storage locations are required in the array. Therefore, exactly twice as much storage is required for complex vectors and matrices as for real vectors and matrices of the same precision. See "How Do You Set Up Your Scalar Data?" on page 28 for a description of real and complex numbers, and "How Do You Set Up Your Arrays?" on page 29 for a description of how real and complex data is stored in arrays.

How Stride Is Used for Vectors

The stride for a vector is an increment that is used to step through array storage to select the vector elements from an array. To define exactly which elements become the conceptual vector in the array, the following items are used together:

- The location of the vector within the array
- The stride for the vector
- The number of elements, *n*, to be processed

The stride can be positive, negative, or 0. For positive and negative strides, if you specify vector elements beyond the range of the array, your results are be unpredictable, and you may get program errors.

This section explains how each of the three types of stride is used to select the vector elements from the array.

Positive Stride

When a positive stride is specified for a vector, the location specified by the argument for the vector is the location of the first element in the vector, element x_1. The vector is in forward order in the array: (x_1, x_2, ..., x_n). For example, if you specify `X(1)` for vector ***x***, where `X` is declared as `X(0:12)` and defined as:

```
X = (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0)
```

then processing begins at the second element in `X`, which is 2.0.

To find each successive element, the stride is added cumulatively to the starting point of vector ***x*** in the array. In this case, the starting point is `X(1)`. If the stride specified for vector ***x*** is 3 and the number of elements to be processed is 4, then the resulting elements selected from `X` for vector ***x*** are: `X(1)`,`X(4)`,`X(7)`, and `X(10)`.

Vector ***x*** is then:

```
(2.0, 5.0, 8.0, 11.0)
```

As shown in this example, a vector does not have to extend to the end of the array. Elements are selected from the second to the eleventh element of the array, and the array elements after that are not used.

This element selection can be expressed in general terms. Using BEGIN as the starting point in an array X and *inc* as the stride, this results in the following elements being selected from the array:

X(BEGIN)
X(BEGIN+*inc*)
X(BEGIN+(2)*inc*)
X(BEGIN+(3)*inc*)
.
.
.
X(BEGIN+(*n*−1)*inc*)

The following general formula can be used to calculate each vector element position in a one-dimensional array:

x_i = X(BEGIN + (*i*−1)(*inc*)) for *i* = 1, *n*

When using an array with more than one dimension, you should understand how the array elements are stored to ensure that elements are selected properly. For a description of array storage, see "Setting Up Arrays in Fortran" on page 112. You should remember that the elements of an array are selected as they are arranged in storage, regardless of the number of dimensions defined in the array. Stride is used to step through array storage until *n* elements are selected. ESSL processing stops at that point. For example, given the following two-dimensional array, declared as A(1:7,1:4).

Matrix A is:

```
⎡                          ⎤
| 1.0   8.0  15.0  22.0 |
| 2.0   9.0  16.0  23.0 |
| 3.0  10.0  17.0  24.0 |
| 4.0  11.0  18.0  25.0 |
| 5.0  12.0  19.0  26.0 |
| 6.0  13.0  20.0  27.0 |
| 7.0  14.0  21.0  28.0 |
⎣                          ⎦
```

with A(3,1) specified for vector ***x***, a stride of 2, and the number of elements to be processed as 12, the resulting vector ***x*** is:

(3.0, 5.0, 7.0, 9.0, 11.0, 13.0, 15.0, 17.0, 19.0, 21.0, 23.0, 25.0)

This is not a conventional use of arrays, and you should be very careful when using this technique.

Zero Stride

When a zero stride is specified for a vector, the starting point for the vector is the only element used in the computation. The starting point for the vector is at the location specified by the argument for the vector, just as though you had specified a positive stride. For example, if you specify X for vector ***x***, where X is defined as:

X = (5.0, 4.0, 3.0, 2.0, 1.0)

and you specify the number of elements, *n*, to be processed as 6, then processing begins at the first element, which is 5.0. This element is used for each of the six elements in vector ***x***.

This makes the conceptual vector ***x*** appear as:

```
(5.0, 5.0, 5.0, 5.0, 5.0, 5.0)
```

The following general formula shows how to calculate each vector position in a one-dimensional array:

x_i = X(BEGIN) for $i = 1, n$

Negative Stride

When a negative stride is specified for a vector, the location specified for the vector is actually the location of the last element in the vector. In other words, the vector is in reverse order in the array: (x_n, x_{n-1}, ..., x_1). You specify the end of the vector, (x_n). ESSL then calculates where the starting point (x_1) is by using the following arguments:

- The location of the vector in the array
- The stride for the vector, *inc*
- The number of elements, *n*, to be processed

If you specify vector ***x*** at location X(BEGIN) in array X with a negative stride of *inc* and *n* elements to be processed, then the following formula gives the starting point of vector ***x*** in the array:

X(BEGIN + (–*n*+1)(*inc*))

For example, if you specify X(2) for vector ***x***, where X is declared as X(1:9) and defined as:

```
X = (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0)
```

and if you specify a stride of –2, and four elements to be processed, processing begins at the following element in X:

X(2+(–4+1)(–2)) = X(8)

where element X(8) is 8.0.

To find each of the *n* successive element positions in the array, you successively add the stride to the starting point *n*–1 times. Suppose the formula calculated a starting point of X(SP); the elements selected are:

X(SP)
X(SP+*inc*)
X(SP+(2)*inc*)
X(SP+(3)*inc*)
.
.
.
X(SP+(*n*–1)*inc*)

In the above example, the resulting elements selected from X for vector ***x*** are X(8), X(6), X(4), and X(2). This makes the resulting vector ***x*** appear as follows:

```
(8.0, 6.0, 4.0, 2.0)
```

The following general formula can be used to calculate each vector element position in a one-dimensional array:

x_i = X(BEGIN + (–*n*+*i*)(*inc*)) for $i = 1, n$

Sparse Vector

A sparse vector is a vector having a relatively small number of nonzero elements.

Consider the following as an example of a sparse vector ***x*** with *n* elements, where *n* is 11, and vector ***x*** is:

```
(0.0, 0.0, 1.0, 0.0, 2.0, 3.0, 0.0, 4.0, 0.0, 5.0, 0.0)
```

In Storage

There are two storage modes that apply to sparse vectors: full-vector storage mode and compressed-vector storage mode. When a sparse vector is stored in **full-vector storage mode**, all its elements, including its zero elements, are stored in an array.

For example, sparse vector ***x*** is stored in full-vector storage mode in a one-dimensional array `X`, as follows:

```
X  = (0.0, 0.0, 1.0, 0.0, 2.0, 3.0, 0.0, 4.0, 0.0, 5.0, 0.0)
```

When a sparse vector is stored in **compressed-vector storage mode**, it is stored without its zero elements. It consists of two one-dimensional arrays, each with a length of *nz*, where *nz* is the number of nonzero elements in vector ***x***:

- The first array contains the nonzero elements of the sparse vector ***x***, stored contiguously within the array.

  Note: The ESSL subroutines do not check that all elements are nonzero. You do not get an error if any elements are zero.

- The second array contains a sequence of integers indicating the element positions (indices) of the nonzero elements of the sparse vector ***x*** stored in full-vector storage mode. This is referred to as the indices array.

For example, the sparse vector ***x*** shown above might have its five nonzero elements stored in ascending order in array `X` of length 5, as follows:

```
X = (1.0, 2.0, 3.0, 4.0, 5.0)
```

in which case, the array of indices, `INDX`, also of length 5, contains:

```
INDX = (3, 5, 6, 8, 10)
```

If the sparse vector ***x*** has its elements stored in random order in the array `X` as:

```
X  = (5.0, 3.0, 4.0, 1.0, 2.0)
```

then the array `INDX` contains:

```
INDX = (10, 6, 8, 3, 5)
```

In general terms, this storage technique can be expressed as follows:

For each $x_j \neq 0$, for $j = 1, n$
there exists i, where $1 \leq i \leq nz$,
such that $\mathtt{X}(i) = x_j$ and $\mathtt{INDX}(i) = j$.

where:

x_1, ..., x_n are the n elements of sparse vector $\boldsymbol{x}$, stored in full-vector storage mode.
`X` is the array containing the nz nonzero elements of sparse vector $\boldsymbol{x}$; that is, vector $\boldsymbol{x}$ is stored in compressed-vector storage mode.
`INDX` is the array containing the nz indices indicating the element positions.

To avoid an error when using the `INDX` array to access the elements in any other target vector, the length of the target vector must be greater than or equal to max(`INDX`(i)) for $i = 1, nz$.

Matrices

A matrix, also referred to as a general matrix, is an m by n ordered collection of numbers. It is represented symbolically as:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & \cdot & \cdot & \cdot & a_{1n} \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ a_{m1} & \cdot & \cdot & \cdot & a_{mn} \end{bmatrix}$$

where the matrix is named $\boldsymbol{A}$ and has m rows and n columns. The elements of the matrix are a_{ij}, where $i = 1, m$ and $j = 1, n$.

Matrices can contain either real or complex numbers. Those containing real numbers are called real matrices; those containing complex numbers are called complex matrices.

Transpose of a Matrix

The transpose of a matrix $\boldsymbol{A}$ is a matrix formed from $\boldsymbol{A}$ by interchanging the rows and columns such that row i of matrix $\boldsymbol{A}$ becomes column i of the transposed matrix. The transpose of $\boldsymbol{A}$ is denoted by $\boldsymbol{A}^{\mathrm{T}}$. Each element a_{ij} in $\boldsymbol{A}$ becomes element a_{ji} in $\boldsymbol{A}^{\mathrm{T}}$. If $\boldsymbol{A}$ is an m by n matrix, then $\boldsymbol{A}^{\mathrm{T}}$ is an n by m matrix. The following represents a matrix and its transpose:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & \cdot & \cdot & \cdot & a_{1n} \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ a_{m1} & \cdot & \cdot & \cdot & a_{mn} \end{bmatrix} \qquad \boldsymbol{A}^{\mathrm{T}} = \begin{bmatrix} a_{11} & \cdot & \cdot & \cdot & a_{m1} \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ a_{1n} & \cdot & \cdot & \cdot & a_{mn} \end{bmatrix}$$

ESSL assumes that all matrices are stored in untransformed format, such as matrix $\boldsymbol{A}$ shown above. No movement of data is necessary in your application program when you are processing transposed matrices. The ESSL subroutines adjust their selection of elements from the matrix when an argument in the calling sequence indicates that the transposed matrix is to be used in the computation. Examples of this are the *transa* and *transb* arguments specified for SGEADD, matrix addition.

Conjugate Transpose of a Matrix

The conjugate transpose of a matrix ***A***, containing complex numbers, is denoted by $\boldsymbol{A}^{\mathrm{H}}$ and is expressed as follows:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \qquad \boldsymbol{A}^{\mathrm{H}} = \begin{bmatrix} \overline{a}_{11} & \cdots & \overline{a}_{m1} \\ \vdots & & \vdots \\ \overline{a}_{1n} & \cdots & \overline{a}_{mn} \end{bmatrix}$$

Just as for the transpose of a matrix, the conjugate transpose of a matrix is stored in untransformed format. No movement of data is necessary in your program.

In Storage

A matrix is usually stored in a two-dimensional array. Its elements are stored successively within the array. Each column of the matrix is stored successively in the array. The leading dimension argument is used to select the matrix elements from each successive column of the array. The starting point of the matrix in the array is specified as the argument for the matrix in the ESSL calling sequence. For example, if matrix ***A*** is contained in array `A` and starts at the first element in the first row and first column of `A`, you should specify `A` as the argument for matrix ***A***, such as in:

```
CALL  SGEMX (5,2,1.0,A,6,X,1,Y,1)
```

where, in the matrix-vector product, the number of rows in matrix ***A*** is 5, the number of columns in matrix ***A*** is 2, the scaling constant is 1.0, the location of the matrix is `A`, the leading dimension is 6, the vectors used in the matrix-vector product are `X` and `Y`, and their strides are 1.

If matrix ***A*** is contained in the array `BIG`, declared as `BIG(1:20,1:30)`, and starts at the second row and third column of `BIG`, you should specify `BIG(2,3)` as the argument for matrix ***A***, such as in:

```
CALL  SGEMX (5,2,1.0,BIG(2,3),6,X,1,Y,1)
```

See “How Leading Dimension Is Used for Matrices” on page 64 for a complete description of how matrices are stored within arrays.

For a complex matrix, a special storage arrangement is used to accommodate the two parts, *a* and *b*, of each complex number (*a*+*bi*) in the array. For each complex number, two sequential storage locations are required in the array. Therefore, exactly twice as much storage is required for complex matrices as for real matrices of the same precision. See “How Do You Set Up Your Scalar Data?” on page 28 for a description of real and complex numbers, and “How Do You Set Up Your Arrays?” on page 29 for a description of how real and complex data is stored in arrays.

How Leading Dimension Is Used for Matrices

The leading dimension for a two-dimensional array is an increment that is used to find the starting point for the matrix elements in each successive column of the array. To define exactly which elements become the conceptual matrix in the array, the following items are used together:

- The location of the matrix within the array
- The leading dimension
- The number of rows, *m*, to be processed in the array
- The number of columns, *n*, to be processed in the array

The leading dimension must always be positive. It must always be greater than or equal to *m*, the number of rows in the matrix to be processed. For an array, `A`, declared as `A(E1:E2,F1:F2)`, the leading dimension is equal to:

```
(E2-E1+1)
```

The starting point for selecting the matrix elements from the array is at the location specified by the argument for the matrix in the ESSL calling sequence. For example, if you specify `A(3,0)` for a 4 by 4 matrix ***A***, where `A` is declared as `A(1:7,0:4)`:

```
[                                  ]
| 1.0   8.0  15.0  22.0  29.0 |
| 2.0   9.0  16.0  23.0  30.0 |
| 3.0  10.0  17.0  24.0  31.0 |
| 4.0  11.0  18.0  25.0  32.0 |
| 5.0  12.0  19.0  26.0  33.0 |
| 6.0  13.0  20.0  27.0  34.0 |
| 7.0  14.0  21.0  28.0  35.0 |
[                                  ]
```

then processing begins at the element at row 3 and column 0 in array `A`, which is 3.0.

The leading dimension is used to find the starting point for the matrix elements in each of the *n* successive columns in the array. ESSL subroutines assume that the arrays are stored in column-major order, as described under "How Do You Set Up Your Arrays?" on page 29, and they add the leading dimension (times the size of the element in bytes) to the starting point. They do this *n*−1 times. This finds the starting point in each of the *n* columns of the array.

In the above example, the leading dimension is:

```
E2-E1+1 = 7-1+1 = 7
```

If the number of columns, *n*, to be processed is 4, the starting points are: `A(3,0)`, `A(3,1)`, `A(3,2)`, and `A(3,3)`. These are elements 3.0, 10.0, 17.0, and 24.0 for a_{11}, a_{12}, a_{13}, and a_{14}, respectively.

In general terms, this results in the following starting positions of each column in the matrix being calculated as follows:

```
A(BEGINI, BEGINJ)
A(BEGINI, BEGINJ+1)
A(BEGINI, BEGINJ+2)
.
.
.
```

A(BEGINI, BEGINJ+*n*−1)

To find the elements in each column of the array, 1 is added successively to the starting point in the column until *m* elements are selected. This is why the leading dimension must be greater than or equal to *m*; otherwise, you go past the end of each dimension of the array. In the above example, if the number of elements, *m*, to be processed in each column is 4, the following elements are selected from array A for the first column of the matrix: A(3,0), A(4,0), A(5,0), and A(6,0). These are elements 3.0, 4.0, 5.0, and 6.0, corresponding to the matrix elements a_{11}, a_{21}, a_{31}, and a_{41}, respectively.

Column element selection can also be expressed in general terms. Using A(BEGINI,BEGINJ) as the starting point in the array, this results in the following elements being selected from each column in the array:

A(BEGINI, BEGINJ)
A(BEGINI+1, BEGINJ)
A(BEGINI+2, BEGINJ)
.
.
.
A(BEGINI+*m*−1, BEGINJ)

Combining this with the technique already described for finding the starting point in each column of the array, the resulting matrix in the example is:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & \cdots & a_{14} \\ \vdots & & \vdots \\ a_{41} & \cdots & a_{44} \end{bmatrix} = \begin{bmatrix} 3.0 & 10.0 & 17.0 & 24.0 \\ 4.0 & 11.0 & 18.0 & 25.0 \\ 5.0 & 12.0 & 19.0 & 26.0 \\ 6.0 & 13.0 & 20.0 & 27.0 \end{bmatrix}$$

As shown in this example, a matrix does not have to include all columns and rows of an array. The elements of matrix ***A*** are selected from rows 3 through 6 and columns 0 through 3 of the array. Rows 1, 2, and 7 and column 4 of the array are not used.

Symmetric Matrix

The matrix ***A*** is symmetric if it has the property $\boldsymbol{A} = \boldsymbol{A}^{\mathrm{T}}$, which means:

- It has the same number of rows as it has columns; that is, it has *n* rows and *n* columns.
- The value of every element a_{ij} on one side of the main diagonal equals its mirror image a_{ji} on the other side: $a_{ij} = a_{ji}$ for $1 \le i \le n$ and $1 \le j \le n$.

The following matrix illustrates a symmetric matrix of order *n*; that is, it has *n* rows and *n* columns. The subscripts on each side of the diagonal appear the same to show which elements are equal:

$$A = \begin{bmatrix} a_{11} & a_{21} & a_{31} & . & . & . & a_{n1} \\ a_{21} & a_{22} & a_{32} & & & & . \\ a_{31} & a_{32} & a_{33} & & & & . \\ . & & & . & & & . \\ . & & & & . & & . \\ . & & & & & . & . \\ a_{n1} & . & . & . & . & . & a_{nn} \end{bmatrix}$$

In Storage

The four storage modes used for storing symmetric matrices are described in the following sections:

- "Lower-Packed Storage Mode"
- "Upper-Packed Storage Mode" on page 67
- "Lower Storage Mode" on page 68
- "Upper Storage Mode" on page 69

The storage technique you should use depends on the ESSL subroutine you are using and is given under Notes in each subroutine description.

Lower-Packed Storage Mode: When a symmetric matrix is stored in lower-packed storage mode, the lower triangular part of the symmetric matrix is stored, including the diagonal, in a one-dimensional array. The lower triangle is packed by columns. (This is equivalent to packing the upper triangle by rows.) The matrix is packed sequentially column by column in $n(n+1)/2$ elements of a one-dimensional array. To calculate the location of each element a_{ij} of matrix $\boldsymbol{A}$ in an array, AP, using the lower triangular packed technique, use the following formula:

$$\mathtt{AP}(i + ((2n-j)(j-1)/2)) = a_{ij} \qquad \text{where } i \geq j$$

This results in the following storage arrangement for the elements of a symmetric matrix $\boldsymbol{A}$ in an array AP:

AP(1)	$= a_{11}$ (start the first column)
AP(2)	$= a_{21}$
AP(3)	$= a_{31}$
.	.
.	.
.	.
AP(n)	$= a_{n1}$
AP(n+1)	$= a_{22}$ (start the second column)
AP(n+2)	$= a_{32}$
.	.
.	.
.	.
AP(2n−1)	$= a_{n2}$
AP(2n)	$= a_{33}$ (start the third column and so forth)
AP(2n+1)	$= a_{43}$
.	.
.	.
.	.
AP(n(n+1)/2)	$= a_{nn}$

Following is an example of a symmetric matrix that uses the element values to show the order in which the matrix elements are stored in the array.

Given the following matrix ***A***:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 6 & 7 & 8 & 9 \\ 3 & 7 & 10 & 11 & 12 \\ 4 & 8 & 11 & 13 & 14 \\ 5 & 9 & 12 & 14 & 15 \end{bmatrix}$$

the array is:

```
AP = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
```

Note: Additional work storage is required in the array for some ESSL subroutines; for example, in the simultaneous linear algebraic equation subroutines SPPF, DPPF, SPPS, and DPPS. See the description of those subroutines in Part 2 for details.

Following is an example of how to transform your symmetric matrix to lower-packed storage mode:

```
      K = 0
      DO 1 J=1,N
         DO 2 I=J,N
            K = K+1
            AP(K)=A(I,J)
2        CONTINUE
1     CONTINUE
```

Upper-Packed Storage Mode: When a symmetric matrix is stored in upper-packed storage mode, the upper triangular part of the symmetric matrix is stored, including the diagonal, in a one-dimensional array. The upper triangle is packed by columns. (This is equivalent to packing the lower triangle by rows.) The matrix is packed sequentially column by column in $n(n+1)/2$ elements of a one-dimensional array. To calculate the location of each element a_{ij} of matrix ***A*** in an array AP using the upper triangular packed technique, use the following formula:

$\mathtt{AP}(i+(j(j-1)/2)) = a_{ij}$ where $j \geq i$

This results in the following storage arrangement for the elements of a symmetric matrix ***A*** in an array AP:

AP(1)	$= a_{11}$ (start the first column)
AP(2)	$= a_{12}$ (start the second column)
AP(3)	$= a_{22}$
AP(4)	$= a_{13}$ (start the third column)
AP(5)	$= a_{23}$
AP(6)	$= a_{33}$
AP(7)	$= a_{14}$ (start the fourth column)
.	.
.	.
.	.
AP($j(j-1)/2+1$)	$= a_{1j}$ (start the j-th column)
AP($j(j-1)/2+2$)	$= a_{2j}$
AP($j(j-1)/2+3$)	$= a_{3j}$

.	.
.	.
.	.
AP($j(j-1)/2+j$)	= a_{jj} (end of the j-th column)
.	.
.	.
.	.
AP($n(n+1)/2$)	= a_{nn}

Following is an example of a symmetric matrix that uses the element values to show the order in which the matrix elements are stored in the array. Given the following matrix ***A***:

$$\begin{bmatrix} 1 & 2 & 4 & 7 & 11 \\ 2 & 3 & 5 & 8 & 12 \\ 4 & 5 & 6 & 9 & 13 \\ 7 & 8 & 9 & 10 & 14 \\ 11 & 12 & 13 & 14 & 15 \end{bmatrix}$$

the array is:

```
AP = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
```

Following is an example of how to transform your symmetric matrix to upper-packed storage mode:

```
      K = 0
      DO 1 J=1,N
         DO 2 I=1,J
            K = K+1
            AP(K)=A(I,J)
2        CONTINUE
1     CONTINUE
```

Lower Storage Mode: When a symmetric matrix is stored in lower storage mode, the lower triangular part of the symmetric matrix is stored, including the diagonal, in a two-dimensional array. These elements are stored in the array in the same way they appear in the matrix. The upper part of the matrix is not required to be stored in the array.

Following is an example of a symmetric matrix ***A*** of order 5 and how it is stored in an array AL.

Given the following matrix ***A***:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 6 & 7 & 8 & 9 \\ 3 & 7 & 10 & 11 & 12 \\ 4 & 8 & 11 & 13 & 14 \\ 5 & 9 & 12 & 14 & 15 \end{bmatrix}$$

the array is:

$$AL = \begin{bmatrix} 1 & * & * & * & * \\ 2 & 6 & * & * & * \\ 3 & 7 & 10 & * & * \\ 4 & 8 & 11 & 13 & * \\ 5 & 9 & 12 & 14 & 15 \end{bmatrix}$$

where "*" means you do not have to store a value in that position in the array. However, these storage positions are required.

Upper Storage Mode: When a symmetric matrix is stored in upper storage mode, the upper triangular part of the symmetric matrix is stored, including the diagonal, in a two-dimensional array. These elements are stored in the array in the same way they appear in the matrix. The lower part of the matrix is not required to be stored in the array.

Following is an example of a symmetric matrix ***A*** of order 5 and how it is stored in an array AU.

Given the following matrix ***A***:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 6 & 7 & 8 & 9 \\ 3 & 7 & 10 & 11 & 12 \\ 4 & 8 & 11 & 13 & 14 \\ 5 & 9 & 12 & 14 & 15 \end{bmatrix}$$

the array is:

$$AU = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ * & 6 & 7 & 8 & 9 \\ * & * & 10 & 11 & 12 \\ * & * & * & 13 & 14 \\ * & * & * & * & 15 \end{bmatrix}$$

where "*" means you do not have to store a value in that position in the array. However, these storage positions are required.

Positive Definite or Negative Definite Symmetric Matrix

A real symmetric matrix ***A*** is positive definite if and only if $\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x}$ is positive for all nonzero vectors ***x***.

A real symmetric matrix ***A*** is negative definite if and only if $\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x}$ is negative for all nonzero vectors ***x***.

In Storage

The positive definite or negative definite symmetric matrix is stored in the same way the symmetric matrix is stored. For a description of this storage technique, see "Symmetric Matrix" on page 65.

Complex Hermitian Matrix

A complex matrix is Hermitian if it is equal to its conjugate transpose:

$$\boldsymbol{H} = \boldsymbol{H}^{\mathrm{H}}$$

In Storage

The complex Hermitian matrix is stored using the same four techniques used for symmetric matrices:

- Lower-packed storage mode, as described in "Lower-Packed Storage Mode" on page 66. (The complex Hermitian matrix is not symmetric; therefore, lower-packed storage mode is not equivalent to packing the upper triangle by rows, as it is for a symmetric matrix.)
- Upper-packed storage mode, as described in "Upper-Packed Storage Mode" on page 67. (The complex Hermitian matrix is not symmetric; therefore, upper-packed storage mode is not equivalent to packing the lower triangle by rows, as it is for a symmetric matrix.)
- Lower storage mode, as described in "Lower Storage Mode" on page 68.
- Upper storage mode, as described in "Upper Storage Mode" on page 69.

Following is an example of a complex Hermitian matrix $\boldsymbol{H}$ of order 5.

Given the following matrix $\boldsymbol{H}$:

```
[                                                  ]
| (11,  0) (21, -1) (31,  1) (41, -1) (51, -1) |
| (21,  1) (22,  0) (32, -1) (42, -1) (52,  1) |
| (31, -1) (32,  1) (33,  0) (43, -1) (53, -1) |
| (41,  1) (42,  1) (43,  1) (44,  0) (54, -1) |
| (51,  1) (52, -1) (53,  1) (54,  1) (55,  0) |
[                                                  ]
```

it is stored in a one-dimensional array, `HP`, in $n(n+1)/2 = 15$ elements as follows:

- In lower-packed storage mode:

```
HP = ((11, *), (21, 1), (31, -1), (41, 1), (51, 1),
      (22, *), (32, 1), (42, 1), (52, -1), (33, *),
      (43, 1), (53, 1), (44, *), (54, 1), (55, *))
```

- In upper-packed storage mode:

```
HP = ((11, *), (21, -1), (22, *), (31, 1), (32, -1),
      (33, *), (41, -1), (42, -1), 43, -1), (44, *),
      (51, -1), (52, 1), (53, -1), (54, -1), (55, *))
```

or it is stored in a two-dimensional array, `HP`, as follows:

- In lower storage mode:

```
        ⎡                                                  ⎤
        ⎢ (11,  *)   *        *       *       *       ⎥
        ⎢ (21,  1) (22,  *)   *       *       *       ⎥
HP =    ⎢ (31, -1) (32,  1) (33, *)   *       *       ⎥
        ⎢ (41,  1) (42,  1) (43, 1) (44, *)   *       ⎥
        ⎢ (51,  1) (52, -1) (53, 1) (54, 1) (55, *)   ⎥
        ⎣                                                  ⎦
```

- In upper storage mode

```
        ⎡                                                  ⎤
        ⎢ (11, *) (21, -1) (31,  1) (41, -1) (51, -1)   ⎥
        ⎢    *    (22,  *) (32, -1) (42, -1) (52,  1)   ⎥
HP =    ⎢    *       *     (33,  *) (43, -1) (53, -1)   ⎥
        ⎢    *       *        *     (44,  *) (54, -1)   ⎥
        ⎢    *       *        *        *     (55,  *)   ⎥
        ⎣                                                  ⎦
```

where "*" means you do not have to store a value in that position in the array. The imaginary parts of the diagonal elements of a complex Hermitian matrix are always 0, so you do not need to set these values. The ESSL subroutines always assume that the values in these positions are 0.

Positive Definite or Negative Definite Complex Hermitian Matrix

A complex Hermitian matrix $\boldsymbol{A}$ is positive definite if and only if $\boldsymbol{x}^{\mathrm{H}}\boldsymbol{A}\boldsymbol{x}$ is positive for all nonzero vectors $\boldsymbol{x}$.

A complex Hermitian matrix $\boldsymbol{A}$ is negative definite if and only if $\boldsymbol{x}^{\mathrm{H}}\boldsymbol{A}\boldsymbol{x}$ is negative for all nonzero vectors $\boldsymbol{x}$.

In Storage

The positive definite or negative definite complex Hermitian matrix is stored in the same way the complex Hermitian matrix is stored. For a description of this storage technique, see "Complex Hermitian Matrix" on page 70.

Positive Definite or Negative Definite Symmetric Toeplitz Matrix

A positive definite or negative definite symmetric matrix $\boldsymbol{A}$ of order n is also a Toeplitz matrix if and only if:

$$a_{ij} = a_{i-1,j-1} \qquad \text{for } i = 2, n \text{ and } j = 2, n$$

The elements on each diagonal of the Toeplitz matrix have a constant value. For the definition of a positive definite or negative definite symmetric matrix, see "Positive Definite or Negative Definite Symmetric Matrix" on page 69.

The following matrix illustrates a symmetric Toeplitz matrix of order n; that is, it has n rows and n columns:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{21} & a_{31} & . & . & . & a_{n1} \\ a_{21} & a_{11} & a_{21} & . & & & . \\ a_{31} & a_{21} & a_{11} & & . & & . \\ . & . & & . & & . & . \\ . & & . & & . & & a_{31} \\ . & & & . & & . & a_{21} \\ a_{n1} & . & . & . & a_{31} & a_{21} & a_{11} \end{bmatrix}$$

A symmetric Toeplitz matrix of order n is represented by a vector $\boldsymbol{x}$ of length n containing the elements of the first column of the matrix (or the elements of the first row), such that $x_i = a_{i1}$ for $i = 1, n$.

The following vector represents the matrix $\boldsymbol{A}$ shown above:

$$\boldsymbol{x} = \begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \\ . \\ . \\ a_{n1} \end{bmatrix}$$

In Storage

The elements of the vector $\boldsymbol{x}$, which represent a positive definite symmetric Toeplitz matrix, are stored sequentially in an array. This is called packed-symmetric-Toeplitz storage mode. Following is an example of a positive definite symmetric Toeplitz matrix $\boldsymbol{A}$ and how it is stored in an array `X`.

Given the following matrix $\boldsymbol{A}$:

$$\begin{bmatrix} 99 & 12 & 13 & 14 & 15 & 16 \\ 12 & 99 & 12 & 13 & 14 & 15 \\ 13 & 12 & 99 & 12 & 13 & 14 \\ 14 & 13 & 12 & 99 & 12 & 13 \\ 15 & 14 & 13 & 12 & 99 & 12 \\ 16 & 15 & 14 & 13 & 12 & 99 \end{bmatrix}$$

the array is:

```
X = (99, 12, 13, 14, 15, 16)
```

Positive Definite or Negative Definite Complex Hermitian Toeplitz Matrix

A positive definite or negative definite complex Hermitian matrix $\boldsymbol{A}$ of order n is also a Toeplitz matrix if and only if:

$a_{ij} = a_{i-1,j-1}$ for $i = 2, n$ and $j = 2, n$

The real part of the diagonal elements of the Toeplitz matrix must have a constant value. The imaginary part of the diagonal elements must be zero.

For the definition of a positive definite of negative definite complex Hermitian matrix, see "Positive Definite or Negative Definite Complex Hermitian Matrix" on page 71.

The following matrix illustrates a complex Hermitian Toeplitz matrix of order n; that is, it has n rows and n columns:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & . & . & . & a_{1n} \\ \bar{a}_{12} & a_{11} & a_{12} & . & & & . \\ \bar{a}_{13} & \bar{a}_{12} & a_{11} & & . & & . \\ . & . & & . & & . & . \\ . & & . & & . & & a_{13} \\ . & & & . & & . & a_{12} \\ \bar{a}_{1n} & . & . & . & \bar{a}_{13} & \bar{a}_{12} & a_{11} \end{bmatrix}$$

A complex Hermitian Toeplitz matrix of order n is represented by a vector $\boldsymbol{x}$ of length n containing the elements of the first row of the matrix.

The following vector represents the matrix $\boldsymbol{A}$ shown above.

$$\boldsymbol{x} = \begin{bmatrix} a_{11} \\ a_{12} \\ a_{13} \\ . \\ . \\ a_{1n} \end{bmatrix}$$

In Storage

The elements of the vector $\boldsymbol{x}$, which represent a positive definite complex Hermitian Toeplitz matrix, are stored sequentially in an array. This is called packed-Hermitian-Toeplitz storage mode. Following is an example of a positive definite complex Hermitian Toeplitz matrix $\boldsymbol{A}$ and how it is stored in an array `X`.

Given the following matrix $\boldsymbol{A}$:

```
[                                                                  ]
|  (10.0, 0.0)   (2.0, -3.0)  (-3.0, 1.0)   (1.0, 1.0) |
|   (2.0, 3.0)   (10.0, 0.0)  (2.0, -3.0)  (-3.0, 1.0) |
| (-3.0, -1.0)    (2.0, 3.0)  (10.0, 0.0)  (2.0, -3.0) |
|  (1.0, -1.0)  (-3.0, -1.0)   (2.0, 3.0)  (10.0, 0.0) |
[                                                                  ]
```

the array is:

```
X = ((10.0, 0.0), (2.0, -3.0), (-3.0, 1.0), (1.0, 1.0))
```

Triangular Matrix

There are two types of triangular matrices: upper triangular matrix and lower triangular matrix. Triangular matrices have the same number of rows as they have columns; that is, they have *n* rows and *n* columns.

A matrix ***U*** is an upper triangular matrix if its nonzero elements are found only in the upper triangle of the matrix, including the main diagonal; that is:

$$u_{ij} = 0 \quad \text{if } i > j$$

A matrix ***L*** is an lower triangular matrix if its nonzero elements are found only in the lower triangle of the matrix, including the main diagonal; that is:

$$l_{ij} = 0 \quad \text{if } i < j$$

The following matrices, ***U*** and ***L***, illustrate upper and lower triangular matrices of order *n*, respectively:

$$\boldsymbol{U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & & \vdots \\ 0 & 0 & u_{33} & & \vdots \\ \vdots & & & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & u_{nn} \end{bmatrix} \qquad \boldsymbol{L} = \begin{bmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & & \vdots \\ l_{31} & l_{32} & l_{33} & & \vdots \\ \vdots & & & \ddots & \vdots \\ l_{n1} & \cdots & \cdots & \cdots & l_{nn} \end{bmatrix}$$

A unit triangular matrix is a triangular matrix in which all the diagonal elements have a value of one; that is:

- For an upper triangular matrix, $u_{ij} = 1$ if $i = j$.
- For an lower triangular matrix, $l_{ij} = 1$ if $i = j$.

The following matrices, ***U*** and ***L***, illustrate upper and lower unit real triangular matrices of order *n*, respectively:

$$\boldsymbol{U} = \begin{bmatrix} 1 & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & 1 & u_{23} & & \vdots \\ 0 & 0 & 1 & & \vdots \\ \vdots & & & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & 1 \end{bmatrix} \qquad \boldsymbol{L} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ l_{21} & 1 & 0 & & \vdots \\ l_{31} & l_{32} & 1 & & \vdots \\ \vdots & & & \ddots & \vdots \\ l_{n1} & \cdots & \cdots & \cdots & 1 \end{bmatrix}$$

In Storage

The four storage modes used for storing triangular matrices are described in the following sections:

- "Upper-Triangular-Packed Storage Mode" on page 75
- "Lower-Triangular-Packed Storage Mode" on page 75
- "Upper-Triangular Storage Mode" on page 76

- "Lower-Triangular Storage Mode" on page 76

It is important to note that because the diagonal elements of a unit triangular matrix are always one, you do not need to set these values in the array for these four storage modes. ESSL always assumes that the values in these positions are one.

Upper-Triangular-Packed Storage Mode: When an upper-triangular matrix is stored in upper-triangular-packed storage mode, the upper triangle of the matrix is stored, including the diagonal, in a one-dimensional array. The upper triangle is packed by columns. The elements are packed sequentially, column by column, in $n(n+1)/2$ elements of a one-dimensional array. To calculate the location of each element of the triangular matrix in the array, use the technique described in "Upper-Packed Storage Mode" on page 67.

Following is an example of an upper triangular matrix ***U*** of order 5 and how it is stored in array UP. It uses the element values to show the order in which the elements are stored in the one-dimensional array.

Given the following matrix ***U***:

$$\begin{bmatrix} 1 & 2 & 4 & 7 & 11 \\ 0 & 3 & 5 & 8 & 12 \\ 0 & 0 & 6 & 9 & 13 \\ 0 & 0 & 0 & 10 & 14 \\ 0 & 0 & 0 & 0 & 15 \end{bmatrix}$$

the array is:

```
UP = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
```

Lower-Triangular-Packed Storage Mode: When a lower-triangular matrix is stored in lower-triangular-packed storage mode, the lower triangle of the matrix is stored, including the diagonal, in a one-dimensional array. The lower triangle is packed by columns. The elements are packed sequentially, column by column, in $n(n+1)/2$ elements of a one-dimensional array. To calculate the location of each element of the triangular matrix in the array, use the technique described in "Lower-Packed Storage Mode" on page 66.

Following is an example of a lower triangular matrix ***L*** of order 5 and how it is stored in array LP. It uses the element values to show the order in which the elements are stored in the one-dimensional array.

Given the following matrix ***L***:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 2 & 6 & 0 & 0 & 0 \\ 3 & 7 & 10 & 0 & 0 \\ 4 & 8 & 11 & 13 & 0 \\ 5 & 9 & 12 & 14 & 15 \end{bmatrix}$$

the array is:

```
LP = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
```

Upper-Triangular Storage Mode: A triangular matrix is stored in upper-triangular storage mode in a two-dimensional array. Only the elements in the upper triangle of the matrix, including the diagonal, are stored in the upper triangle of the array.

Following is an example of an upper triangular matrix ***U*** of order 5 and how it is stored in array UTA.

Given the following matrix ***U***:

$$\begin{bmatrix} 11 & 12 & 13 & 14 & 15 \\ 0 & 22 & 23 & 24 & 25 \\ 0 & 0 & 33 & 34 & 35 \\ 0 & 0 & 0 & 44 & 45 \\ 0 & 0 & 0 & 0 & 55 \end{bmatrix}$$

the array is:

$$\mathrm{UTA} = \begin{bmatrix} 11 & 12 & 13 & 14 & 15 \\ * & 22 & 23 & 24 & 25 \\ * & * & 33 & 34 & 35 \\ * & * & * & 44 & 45 \\ * & * & * & * & 55 \end{bmatrix}$$

where "*" means you do not have to store a value in that position in the array.

Lower-Triangular Storage Mode: A triangular matrix is stored in lower-triangular storage mode in a two-dimensional array. Only the elements in the lower triangle of the matrix, including the diagonal, are stored in the lower triangle of the array.

Following is an example of a lower triangular matrix ***L*** of order 5 and how it is stored in array LTA.

Given the following matrix ***L***:

$$\begin{bmatrix} 11 & 0 & 0 & 0 & 0 \\ 21 & 22 & 0 & 0 & 0 \\ 31 & 32 & 33 & 0 & 0 \\ 41 & 42 & 43 & 44 & 0 \\ 51 & 52 & 53 & 54 & 55 \end{bmatrix}$$

the array is:

$$\mathrm{LTA} = \begin{bmatrix} 11 & * & * & * & * \\ 21 & 22 & * & * & * \\ 31 & 32 & 33 & * & * \\ 41 & 42 & 43 & 44 & * \\ 51 & 52 & 53 & 54 & 55 \end{bmatrix}$$

where "*" means you do not have to store a value in that position in the array.

General Band Matrix

A general band matrix has its nonzero elements arranged uniformly near the diagonal, such that:

$$a_{ij} = 0 \quad \text{if } (i-j) > ml \text{ or } (j-i) > mu$$

where *ml* and *mu* are the lower and upper band widths, respectively, and *ml*+*mu*+1 is the total band width.

The following matrix illustrates a **square** general band matrix of order *n*, where the band widths are $ml = q-1$ and $mu = p-1$:

$$A = \begin{array}{c} \\ \\ - \\ \uparrow \\ ml \\ \downarrow \\ - \end{array} \begin{array}{c} \overset{\left| \leftarrow mu \rightarrow \right|}{} \\ \begin{bmatrix} a_{11} & a_{12} & a_{13} & . & . & a_{1p} & 0 & . & . & 0 \\ a_{21} & a_{22} & a_{23} & & & & . & 0 & & . \\ a_{31} & a_{32} & a_{33} & & & & & . & 0 & . \\ . & & & . & & & & & . & 0 \\ . & & & & . & & & & & . \\ a_{q1} & & & & & . & & & & . \\ 0 & . & & & & & . & & & . \\ . & 0 & . & & & & & . & & . \\ . & & 0 & . & & & & & . & \\ 0 & . & . & 0 & . & . & . & & & a_{nn} \end{bmatrix} \end{array}$$

Some special types of band matrices are:

- Tridiagonal matrix: $ml = mu = 1$
- 9-diagonal matrix: $ml = mu = 4$

The following two matrices illustrate *m* by *n* **rectangular** general band matrices, where the band widths are $ml = q-1$ and $mu = p-1$. For both matrices, the leading diagonal is $a_{11}, a_{22}, a_{33}, ..., a_{nn}$. Following is a general band matrix with $m > n$:

$$A = \begin{array}{c} \\ - \\ \uparrow \\ ml \\ \downarrow \\ - \end{array} \begin{array}{c} \overset{\left| \leftarrow mu \rightarrow \right|}{} \\ \begin{bmatrix} a_{11} & a_{12} & a_{13} & . & . & a_{1p} & 0 & . & . & 0 \\ a_{21} & a_{22} & a_{23} & & & & . & 0 & & . \\ a_{31} & a_{32} & a_{33} & & & & & . & 0 & . \\ . & & & . & & & & & . & 0 \\ . & & & & . & & & & & . \\ a_{q1} & & & & & . & & & & . \\ 0 & . & & & & & . & & & . \\ . & 0 & . & & & & & . & & . \\ . & & 0 & . & & & & & . & \\ . & & & 0 & . & & & & & a_{nn} \\ . & & & & 0 & . & & & & . \\ . & & & & & 0 & . & & & . \\ 0 & . & . & . & . & . & 0 & . & . & a_{mn} \end{bmatrix} \end{array}$$

Following is a general band matrix with $m < n$:

$$
\begin{array}{cc}
 & \left| \leftarrow mu \rightarrow \right| \\
\begin{array}{c} - \\ \uparrow \\ ml \\ \downarrow \\ - \end{array} & \begin{bmatrix}
a_{11} & a_{12} & a_{13} & . & . & a_{1p} & 0 & . & . & . & . & . & 0 \\
a_{21} & a_{22} & a_{23} & & & & . & 0 & & & & & . \\
a_{31} & a_{32} & a_{33} & & & & & . & 0 & & & & . \\
. & & & . & & & & & . & 0 & & & . \\
. & & & & . & & & & & . & 0 & & . \\
a_{q1} & & & & & . & & & & & . & 0 & . \\
0 & . & & & & & . & & & & & . & 0 \\
. & 0 & . & & & & & . & & & & & . \\
. & & 0 & . & & & & & . & & & & . \\
0 & . & . & 0 & . & . & . & . & . & a_{nn} & . & . & a_{mn}
\end{bmatrix}
\end{array}
$$

In Storage

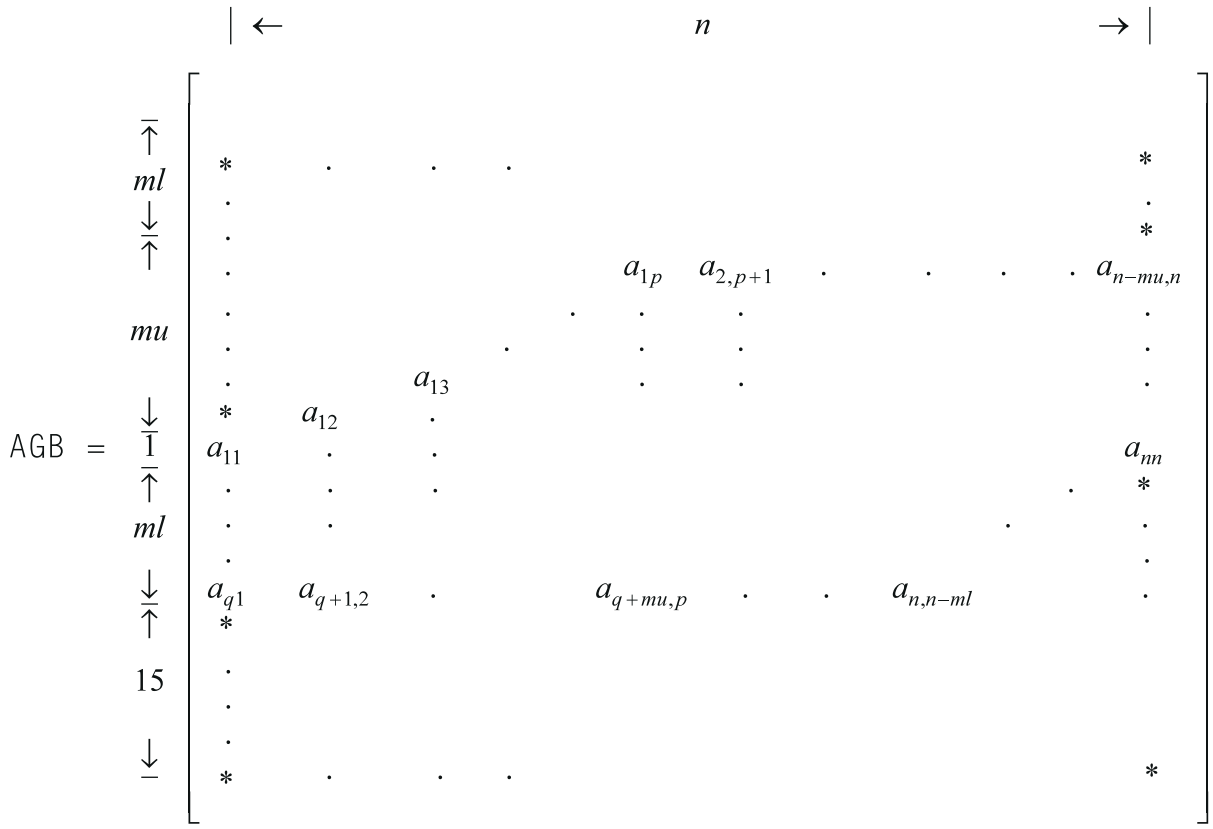
The two storage modes used for storing general band matrices are described in the following sections:

- "General-Band Storage Mode"
- "BLAS-General-Band Storage Mode" on page 80

General-Band Storage Mode: (This storage mode is used only for square matrices.) Only the band elements of a general band matrix are stored for general-band storage mode. Additional storage must also be provided for fill- in. General-band storage mode packs the matrix elements by columns into a two-dimensional array, such that each diagonal of the matrix appears as a row in the packed array.

For a matrix $\boldsymbol{A}$ of order n with band widths ml and mu, the array must have a leading dimension, lda, greater than or equal to $2ml+mu+16$. The size of the second dimension must be (at least) n, the number of columns in the matrix.

Using array `AGB`, which is declared as `AGB`($2ml+mu+16$, n), the columns of elements in matrix $\boldsymbol{A}$ are stored in each column in array `AGB` as follows, where a_{11} is stored at `AGB`($ml+mu+1$, 1):



where "*" means you do not store an element in that position in the array.

In the ESSL subroutine computation, some of the positions in the array indicated by an "*" are used for fill- in. Other positions may not be accessed at all.

Following is an example of a band matrix ***A*** of order 9 and band widths of $ml = 2$ and $mu = 3$.

Given the following matrix ***A***:

$$\begin{bmatrix} 11 & 12 & 13 & 14 & 0 & 0 & 0 & 0 & 0 \\ 21 & 22 & 23 & 24 & 25 & 0 & 0 & 0 & 0 \\ 31 & 32 & 33 & 34 & 35 & 36 & 0 & 0 & 0 \\ 0 & 42 & 43 & 44 & 45 & 46 & 47 & 0 & 0 \\ 0 & 0 & 53 & 54 & 55 & 56 & 57 & 58 & 0 \\ 0 & 0 & 0 & 64 & 65 & 66 & 67 & 68 & 69 \\ 0 & 0 & 0 & 0 & 75 & 76 & 77 & 78 & 79 \\ 0 & 0 & 0 & 0 & 0 & 86 & 87 & 88 & 89 \\ 0 & 0 & 0 & 0 & 0 & 0 & 97 & 98 & 99 \end{bmatrix}$$

you store it in general-band storage mode in a 23 by 9 array `AGB` as follows, where a_{11} is stored in `AGB(6,1)`:

```
        [                                                  ]
        |  *   *   *   *   *   *   *   *   *  |
        |  *   *   *   *   *   *   *   *   *  |
        |  *   *   *  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   *   *  |
        |  *   *   *   *   *   *   *   *   *  |
        |  *   *   *   *   *   *   *   *   *  |
        |  *   *   *   *   *   *   *   *   *  |
AGB =   |  *   *   *   *   *   *   *   *   *  |
        |  *   *   *   *   *   *   *   *   *  |
        |  *   *   *   *   *   *   *   *   *  |
        |  *   *   *   *   *   *   *   *   *  |
        |  *   *   *   *   *   *   *   *   *  |
        |  *   *   *   *   *   *   *   *   *  |
        |  *   *   *   *   *   *   *   *   *  |
        |  *   *   *   *   *   *   *   *   *  |
        |  *   *   *   *   *   *   *   *   *  |
        |  *   *   *   *   *   *   *   *   *  |
        |  *   *   *   *   *   *   *   *   *  |
        |  *   *   *   *   *   *   *   *   *  |
        [                                                  ]
```

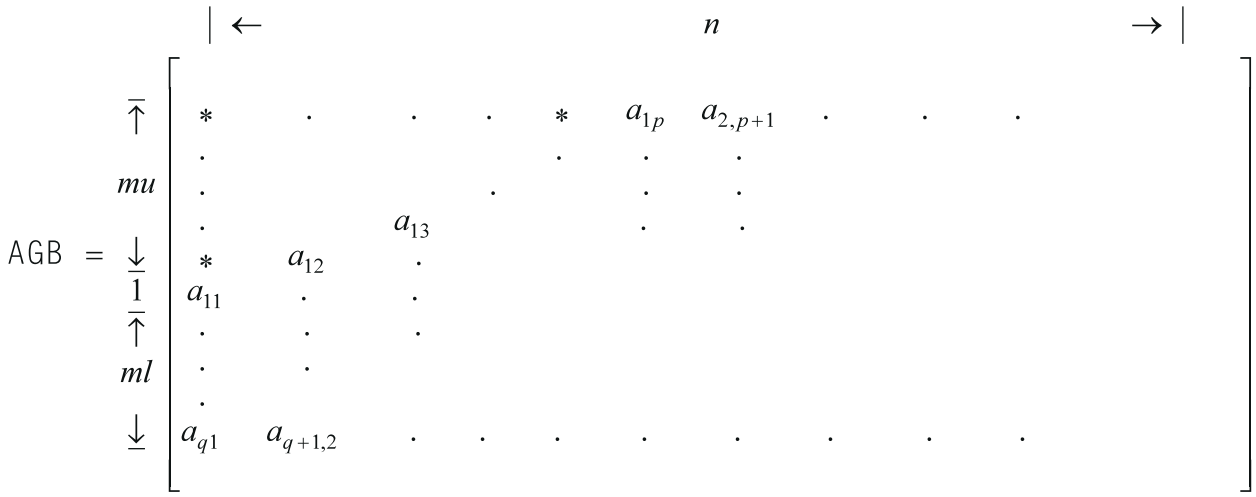
Following is an example of how to transform your general band matrix, of order *n*, to general-band storage mode:

```
      MD=ML+MU+1
      DO 1 J=1,N
         DO 1 I=MAX(J-MU,1),MIN(J+ML,N)
            AGB(I-J+MD,J)=A(I,J)
1     CONTINUE
```

BLAS-General-Band Storage Mode: (This storage mode is used for both square and rectangular matrices.) Only the band elements of a general band matrix are stored for BLAS-general-band storage mode. The storage mode packs the matrix elements by columns into a two-dimensional array, such that each diagonal of the matrix appears as a row in the packed array.

For an *m* by *n* matrix ***A*** with band widths *ml* and *mu*, the array `AGB` must have a leading dimension, *lda*, greater than or equal to *ml*+*mu*+1. The size of the second dimension must be (at least) *n*, the number of columns in the matrix.

Using the array `AGB`, which is declared as `AGB`(*ml*+*mu*+1, *n*), the columns of elements in matrix ***A*** are stored in each column in array `AGB` as follows, where a_{11} is stored at `AGB`(*mu*+1, 1):



where "*" means you do not store an element in that position in the array. These positions are not accessed by ESSL. **Unused positions in the array always occur in the upper left triangle of the array, but may not occur in the lower right triangle of the array, as you can see from the examples given here.**

Following is an example where $m > n$, and general band matrix ***A*** is 9 by 8 with band widths of $ml = 2$ and $mu = 3$.

Given the following matrix ***A***:

```
| 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 |
```

you store it in array `AGB`, declared as `AGB(6,8)`, as follows, where a_{11} is stored in `AGB(4,1)`:

```
AGB =
|  *   *   *  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   * |
```

Following is an example where $m < n$, and general band matrix ***A*** is 7 by 9 with band widths of $ml = 2$ and $mu = 3$.

Given the following matrix ***A***:

$$\begin{bmatrix} 11 & 12 & 13 & 14 & 0 & 0 & 0 & 0 & 0 \\ 21 & 22 & 23 & 24 & 25 & 0 & 0 & 0 & 0 \\ 31 & 32 & 33 & 34 & 35 & 36 & 0 & 0 & 0 \\ 0 & 42 & 43 & 44 & 45 & 46 & 47 & 0 & 0 \\ 0 & 0 & 53 & 54 & 55 & 56 & 57 & 58 & 0 \\ 0 & 0 & 0 & 64 & 65 & 66 & 67 & 68 & 69 \\ 0 & 0 & 0 & 0 & 75 & 76 & 77 & 78 & 79 \end{bmatrix}$$

you store it in array AGB, declared as AGB(6,9), as follows, where a_{11} is stored in AGB(4,1) and the leading diagonal does not fill up the whole row:

$$AGB = \begin{bmatrix} * & * & * & 14 & 25 & 36 & 47 & 58 & 69 \\ * & * & 13 & 24 & 35 & 46 & 57 & 68 & 79 \\ * & 12 & 23 & 34 & 45 & 56 & 67 & 78 & * \\ 11 & 22 & 33 & 44 & 55 & 66 & 77 & * & * \\ 21 & 32 & 43 & 54 & 65 & 76 & * & * & * \\ 31 & 42 & 53 & 64 & 75 & * & * & * & * \end{bmatrix}$$

and where "*" means you do not store an element in that position in the array.

Following is an example of how to transform your general band matrix, for all values of m and n, to BLAS-general-band storage mode:

```
      DO 20 J=1,N
         K=MU+1-J
         DO 10 I=MAX(1,J-MU),MIN(M,J+ML)
            AGB(K+I,J)=A(I,J)
10       CONTINUE
20    CONTINUE
```

Symmetric Band Matrix

A symmetric band matrix is a symmetric matrix whose nonzero elements are arranged uniformly near the diagonal, such that:

$$a_{ij} = 0 \quad \text{if } |i-j| > k$$

where k is the half band width.

The following matrix illustrates a symmetric band matrix of order n, where the half band width $k = q-1$:

$$
\begin{array}{cc}
 & \left| \leftarrow \quad k \quad \rightarrow \right| \\
\boldsymbol{A} = & \begin{bmatrix}
a_{11} & a_{21} & a_{31} & . & . & a_{q1} & 0 & . & . & 0 \\
a_{21} & a_{22} & a_{32} & & & & & 0 & & . \\
a_{31} & a_{32} & a_{33} & & & & & & 0 & . \\
. & & & . & & & & & & 0 \\
. & & & & . & & & & & . \\
a_{q1} & & & & & . & & & & . \\
0 & & & & & & . & & & . \\
. & 0 & & & & & & . & & . \\
. & & 0 & & & & & & . & \\
0 & . & . & 0 & . & . & . & & & a_{nn}
\end{bmatrix}
\end{array}
$$

In Storage

The two storage modes used for storing symmetric band matrices are described in the following sections:

- "Upper-Band-Packed Storage Mode"
- "Lower-Band-Packed Storage Mode" on page 84

Upper-Band-Packed Storage Mode: Only the band elements of the upper triangular part of a symmetric band matrix, including the main diagonal, are stored for upper-band-packed storage mode.

For a matrix $\boldsymbol{A}$ of order n and a half band width of k, the array must have a leading dimension, *lda*, greater than or equal to k+1, and the size of the second dimension must be (at least) n.

Using array `ASB`, which is declared as `ASB`(*lda*,*n*), where $p = lda = k+1$, the elements of a symmetric band matrix are stored as follows:

$$
\mathtt{ASB} = \begin{bmatrix}
* & . & . & . & * & a_{1p} & a_{2,p+1} & . & . & . & a_{n-k,n} \\
. & & & & . & & & & & & \\
. & & & . & & & & & & & \\
. & & . & & & & & & & & . \\
 & * & a_{13} & a_{24} & . & . & . & & & & . \\
* & a_{12} & a_{23} & . & . & . & & & & & . \\
a_{11} & a_{22} & . & . & . & & & & & & a_{nn}
\end{bmatrix}
$$

where "*" means you do not store an element in that position in the array.

Following is an example of a symmetric band matrix $\boldsymbol{A}$ of order 6 and a half band width of 3.

Given the following matrix $\boldsymbol{A}$:

$$\begin{bmatrix} 11 & 12 & 13 & 14 & 0 & 0 \\ 12 & 22 & 23 & 24 & 25 & 0 \\ 13 & 23 & 33 & 34 & 35 & 36 \\ 14 & 24 & 34 & 44 & 45 & 46 \\ 0 & 25 & 35 & 45 & 55 & 56 \\ 0 & 0 & 36 & 46 & 56 & 66 \end{bmatrix}$$

you store it in upper-band-packed storage mode in array ASB, declared as ASB(4,6), as follows.

$$\mathtt{ASB} = \begin{bmatrix} * & * & * & 14 & 25 & 36 \\ * & * & 13 & 24 & 35 & 46 \\ * & 12 & 23 & 34 & 45 & 56 \\ 11 & 22 & 33 & 44 & 55 & 66 \end{bmatrix}$$

Following is an example of how to transform your symmetric band matrix to upper-band-packed storage mode:

```
      DO 20 J=1,N
         M=K+1-J
         DO 10 I=MAX(1,J-K),J
            ASB(M+I,J)=A(I,J)
      10 CONTINUE
   20 CONTINUE
```

Lower-Band-Packed Storage Mode: Only the band elements of the lower triangular part of a symmetric band matrix, including the main diagonal, are stored for lower-band-packed storage mode.

For a matrix ***A*** of order *n* and a half band width of *k*, the array must have a leading dimension, *lda*, greater than or equal to *k*+1, and the size of the second dimension must be (at least) *n*.

Using array ASB, which is declared as ASB(*lda*,*n*), where $q = lda = k+1$, the elements of a symmetric band matrix are stored as follows:

$$\mathtt{ASB} = \begin{bmatrix} a_{11} & a_{22} & \cdots & & a_{nn} \\ a_{21} & a_{32} & & & * \\ a_{31} & a_{42} & & & \cdot \\ \cdot & \cdot & & & \cdot \\ \cdot & \cdot & & & \cdot \\ \cdot & \cdot & & & \\ a_{q1} & a_{q+1,2} & \cdots & a_{n,n-k} * \cdots & * \end{bmatrix}$$

where "*" means you do not store an element in that position in the array.

Following is an example of a symmetric band matrix ***A*** of order 6 and a half band width of 2.

Given the following matrix ***A***:

$$\begin{bmatrix} 11 & 21 & 31 & 0 & 0 & 0 \\ 21 & 22 & 32 & 42 & 0 & 0 \\ 31 & 32 & 33 & 43 & 53 & 0 \\ 0 & 42 & 43 & 44 & 54 & 64 \\ 0 & 0 & 53 & 54 & 55 & 65 \\ 0 & 0 & 0 & 64 & 65 & 66 \end{bmatrix}$$

you store it in lower-band-packed storage mode in array ASB, declared as ASB(3,6), as follows:

$$ASB = \begin{bmatrix} 11 & 22 & 33 & 44 & 55 & 66 \\ 21 & 32 & 43 & 54 & 65 & * \\ 31 & 42 & 53 & 64 & * & * \end{bmatrix}$$

Following is an example of how to transform your symmetric band matrix to lower-band-packed storage mode:

```
      DO 20 J=1,N
         DO 10 I=J,MIN(J+K,N)
            ASB(I-J+1,J)=A(I,J)
      10 CONTINUE
   20 CONTINUE
```

Positive Definite Symmetric Band Matrix

A real symmetric band matrix $\boldsymbol{A}$ is positive definite if and only if $\boldsymbol{x}^{T}\boldsymbol{A}\boldsymbol{x}$ is positive for all nonzero vectors $\boldsymbol{x}$.

In Storage

The positive definite symmetric band matrix is stored in the same way a symmetric band matrix is stored. For a description of this storage technique, see “Symmetric Band Matrix” on page 82.

Complex Hermitian Band Matrix

A complex band matrix is Hermitian if it is equal to its conjugate transpose:

$$\boldsymbol{H} = \boldsymbol{H}^{H}$$

In Storage

The complex Hermitian band matrix is stored using the same two techniques used for symmetric band matrices:

- Lower-band-packed storage mode, as described in “Lower-Band-Packed Storage Mode” on page 84
- Upper-band-packed storage mode, as described in “Upper-Band-Packed Storage Mode” on page 83

Following is an example of a complex Hermitian band matrix $\boldsymbol{H}$ of order 5, having a half band width of 2.

Given the following matrix $\boldsymbol{H}$:

```
[                                                  ]
| (11,  0) (21, -1) (31,  1)  (0,  0)  (0,  0) |
| (21,  1) (22,  0) (32, -1) (42, -1)  (0,  0) |
| (31, -1) (32,  1) (33,  0) (43, -1) (53, -1) |
|  (0,  0) (42,  1) (43,  1) (44,  0) (54, -1) |
|  (0,  0)  (0,  0) (53,  1) (54,  1) (55,  0) |
[                                                  ]
```

you store it in a two-dimensional array HP, as follows:

- In lower-band-packed storage mode:

```
      [                                                  ]
      | (11,  *) (22, *) (33, *) (44,  *) (55, *) |
HP =  | (21,  1) (32, 1) (43, 1) (54,  1)    *    |
      | (31, -1) (42, 1) (53, 1)    *        *    |
      [                                                  ]
```

- In upper-band-packed storage mode:

```
      [                                                  ]
      |    *        *     (31,  1) (42, -1) (53, -1) |
HP =  |    *     (21, -1) (32, -1) (43, -1) (54, -1) |
      | (11, *)  (22,  *) (33,  *) (44,  *) (55,  *) |
      [                                                  ]
```

where "*" means you do not have to store a value in that position in the array. The imaginary parts of the diagonal elements of a complex Hermitian band matrix are always 0, so you do not need to set these values. The ESSL subroutines always assume that the values in these positions are 0.

Triangular Band Matrix

There are two types of triangular band matrices: upper triangular band matrix and lower triangular band matrix. Triangular band matrices have the same number of rows as they have columns; that is, they have n rows and n columns. They have an upper or lower band width of k.

A band matrix $\boldsymbol{U}$ is an upper triangular band matrix if its nonzero elements are found only in the upper triangle of the matrix, including the main diagonal; that is:

$$u_{ij} = 0 \quad \text{if } i > j$$

Its band elements are arranged uniformly near the diagonal in the upper triangle of the matrix, such that:

$$u_{ij} = 0 \quad \text{if } j-i > k$$

The following matrix $\boldsymbol{U}$ illustrates an upper triangular band matrix of order n with an upper band width $k = q-1$:

$$
\begin{array}{c}
\left| \leftarrow k \rightarrow \right| \\
\boldsymbol{U} = \begin{bmatrix}
u_{11} & u_{21} & u_{31} & . & . & u_{q1} & 0 & . & . & 0 \\
0 & u_{22} & u_{32} & & & & & 0 & & . \\
. & 0 & u_{33} & & & & & & 0 & . \\
. & & & . & & & & & & 0 \\
. & & & & . & & & & & . \\
. & & & & & . & & & & . \\
. & & & & & & . & & & . \\
 & & & & & & & . & & \\
 & & & & & & & & . & \\
0 & . & . & . & & & & & 0 & u_{nn}
\end{bmatrix}
\end{array}
$$

A band matrix $\boldsymbol{L}$ is a lower triangular band matrix if its nonzero elements are found only in the lower triangle of the matrix, including the main diagonal; that is:

$$l_{ij} = 0 \quad \text{if } i < j$$

Its band elements are arranged uniformly near the diagonal in the lower triangle of the matrix such that:

$$l_{ij} = 0 \quad \text{if } i-j > k$$

The following matrix $\boldsymbol{L}$ illustrates an upper triangular band matrix of order n with a lower band width $k = q-1$:

$$
\boldsymbol{L} = \begin{array}{c}
- \\
\uparrow \\
k \\
\downarrow \\
- \\
\\
\\
\\
\\
\end{array}
\begin{bmatrix}
l_{11} & 0 & . & . & . & & & & & 0 \\
l_{21} & l_{22} & 0 & & & & & & & . \\
l_{31} & l_{32} & l_{33} & & & & & & & . \\
. & & & . & & & & & & . \\
. & & & & . & & & & & . \\
l_{q1} & & & & & . & & & & . \\
0 & & & & & & . & & & \\
. & 0 & & & & & & . & & \\
. & & 0 & & & & & & . & 0 \\
0 & . & . & 0 & . & . & . & & & l_{nn}
\end{bmatrix}
$$

A triangular band matrix can also be a unit triangular band matrix if all the diagonal elements have a value of 1. For an illustration of a unit triangular matrix, see “Triangular Matrix” on page 74.

In Storage

The two storage modes used for storing triangular band matrices are described in the following sections:

- “Upper-Triangular-Band-Packed Storage Mode” on page 88
- “Lower-Triangular-Band-Packed Storage Mode” on page 89

It is important to note that because the diagonal elements of a unit triangular band matrix are always one, you do not need to set these values in the array for these two storage modes. ESSL always assumes that the values in these positions are one.

Upper-Triangular-Band-Packed Storage Mode: Only the band elements of the upper triangular part of an upper triangular band matrix, including the main diagonal, are stored for upper-triangular-band-packed storage mode.

For a matrix ***U*** of order *n* and an upper band width of *k*, the array must have a leading dimension, *lda*, greater than or equal to *k*+1, and the size of the second dimension must be (at least) *n*.

Using array UTB, which is declared as UTB(*lda*,*n*), where $p = lda = k+1$, the elements of an upper triangular band matrix are stored as follows:

$$\mathtt{UTB} = \begin{bmatrix} & & & & & & & & & \\ * & . & . & . & * & u_{1p} & u_{2,p+1} & . & . & . & u_{n-k,n} \\ . & & & & . & & & & & & \\ . & & & . & & & & & & & \\ . & & . & & & & & & & . & \\ & * & u_{13} & u_{24} & . & . & . & & & . & \\ * & u_{12} & u_{23} & . & . & . & & & & . & \\ u_{11} & u_{22} & . & . & . & & & & & u_{nn} & \\ & & & & & & & & & & \end{bmatrix}$$

where "*" means you do not store an element in that position in the array.

Following is an example of an upper triangular band matrix ***U*** of order 6 and an upper band width of 3.

Given the following matrix ***U***:

$$\begin{bmatrix} 11 & 12 & 13 & 14 & 0 & 0 \\ 0 & 22 & 23 & 24 & 25 & 0 \\ 0 & 0 & 33 & 34 & 35 & 36 \\ 0 & 0 & 0 & 44 & 45 & 46 \\ 0 & 0 & 0 & 0 & 55 & 56 \\ 0 & 0 & 0 & 0 & 0 & 66 \end{bmatrix}$$

you store it in upper-triangular-band-packed storage mode in array UTB, declared as UTB(4,6), as follows:

$$\mathtt{UTB} = \begin{bmatrix} * & * & * & 14 & 25 & 36 \\ * & * & 13 & 24 & 35 & 46 \\ * & 12 & 23 & 34 & 45 & 56 \\ 11 & 22 & 33 & 44 & 55 & 66 \end{bmatrix}$$

Following is an example of how to transform your upper triangular band matrix to upper-triangular-band-packed storage mode:

```
      DO 20 J=1,N
         M=K+1-J
         DO 10 I=MAX(1,J-K),J
            UTB(M+I,J)=U(I,J)
      10 CONTINUE
   20 CONTINUE
```

Lower-Triangular-Band-Packed Storage Mode: Only the band elements of the lower triangular part of a lower triangular band matrix, including the main diagonal, are stored for lower-triangular-band-packed storage mode.

Note: As an alternative to this storage mode, you can specify your arguments in your subroutine in a special way so that ESSL selects the matrix elements properly, and you can leave your matrix stored in full-matrix storage mode. For details, see the Notes in the subroutine description in Part 2 of this book.

For a matrix ***L*** of order *n* and a lower band width of *k*, the array must have a leading dimension, *lda*, greater than or equal to *k*+1, and the size of the second dimension must be (at least) *n*.

Using array LTB, which is declared as LTB(*lda*,*n*), where $q = lda = k+1$, the elements of a lower triangular band matrix are stored as follows:

$$\mathrm{LTB} = \begin{bmatrix} l_{11} & l_{22} & \cdots & & l_{nn} \\ l_{21} & l_{32} & & & * \\ l_{31} & l_{42} & & & . \\ . & . & & & . \\ . & . & & & . \\ . & . & & & \\ l_{q1} & l_{q+1,2} & \cdots & l_{n,n-k} * \cdots & * \end{bmatrix}$$

where "*" means you do not store an element in that position in the array.

Following is an example of a lower triangular band matrix ***L*** of order 6 and a lower band width of 2.

Given the following matrix ***L***:

$$\begin{bmatrix} 11 & 0 & 0 & 0 & 0 & 0 \\ 21 & 22 & 0 & 0 & 0 & 0 \\ 31 & 32 & 33 & 0 & 0 & 0 \\ 0 & 42 & 43 & 44 & 0 & 0 \\ 0 & 0 & 53 & 54 & 55 & 0 \\ 0 & 0 & 0 & 64 & 65 & 66 \end{bmatrix}$$

you store it in lower-triangular-band-packed storage mode in array LTB, declared as LTB(3,6), as follows:

$$\text{LTB} = \begin{bmatrix} 11 & 22 & 33 & 44 & 55 & 66 \\ 21 & 32 & 43 & 54 & 65 & * \\ 31 & 42 & 53 & 64 & * & * \end{bmatrix}$$

Following is an example of how to transform your lower triangular band matrix to lower-triangular-band-packed storage mode:

```
      DO 20 J=1,N
         M=1-J
         DO 10 I=J,MIN(N,J+K)
            LTB(M+I,J)=L(I,J)
      10 CONTINUE
   20 CONTINUE
```

General Tridiagonal Matrix

A general tridiagonal matrix is a matrix whose nonzero elements are found only on the diagonal, subdiagonal, and superdiagonal of the matrix; that is:

$$a_{ij} = 0 \quad \text{if } |i-j| > 1$$

The following matrix illustrates a general tridiagonal matrix of order n:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} & 0 & . & . & . & 0 \\ a_{21} & a_{22} & a_{23} & 0 & & & . \\ 0 & a_{32} & a_{33} & a_{34} & 0 & & . \\ . & 0 & a_{43} & a_{44} & . & . & . \\ . & & 0 & . & . & . & . \\ . & & & . & . & . & . \\ & & & & . & . & . \\ 0 & . & . & . & & . & . \; a_{nn} \end{bmatrix}$$

In Storage

Only the diagonal, subdiagonal, and superdiagonal elements of the general tridiagonal matrix are stored. This is called tridiagonal storage mode. The elements of a general tridiagonal matrix, $\boldsymbol{A}$, of order n are stored in three one-dimensional arrays, `C`, `D`, and `E`, each of length n, where array `C` contains the subdiagonal elements, stored as follows:

$$\texttt{C} = (*, a_{21}, a_{32}, a_{43}, ..., a_{n,n-1})$$

and array `D` contains the main diagonal elements, stored as follows:

$$\texttt{D} = (a_{11}, a_{22}, a_{33}, ..., a_{nn})$$

and array `E` contains the superdiagonal elements, stored as follows:

$$\texttt{E} = (a_{12}, a_{23}, a_{34}, ..., a_{n-1,n}, *)$$

where "*" means you do not store an element in that position in the array.

Following is an example of a general tridiagonal matrix $\boldsymbol{A}$ of order 5:

$$\begin{bmatrix} 11 & 12 & 0 & 0 & 0 \\ 21 & 22 & 23 & 0 & 0 \\ 0 & 32 & 33 & 34 & 0 \\ 0 & 0 & 43 & 44 & 45 \\ 0 & 0 & 0 & 54 & 55 \end{bmatrix}$$

which you store in tridiagonal storage mode in arrays C, D, and E, each of length 5, as follows:

```
C = (*, 21, 32, 43, 54)

D = (11, 22, 33, 44, 55)

E = (12, 23, 34, 45, *)
```

Note: Some ESSL subroutines provide an option for specifying at least *n* additional locations at the end of each of the arrays C, D, and E. These additional locations are used for working storage by the ESSL subroutine. The reasons for choosing this option are explained in the subroutine descriptions.

Symmetric Tridiagonal Matrix

A tridiagonal matrix ***A*** is also symmetric if and only if its nonzero elements are found only on the diagonal, subdiagonal, and superdiagonal of the matrix, and its subdiagonal elements and superdiagonal elements are equal; that is:

$$(a_{ij} = 0 \quad \text{if } |i-j| > 1) \qquad \text{and} \qquad (a_{ij} = a_{ji} \quad \text{if } |i-j| = 1)$$

The following matrix illustrates a symmetric tridiagonal matrix of order *n*:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{21} & 0 & . & . & . & 0 \\ a_{21} & a_{22} & a_{32} & 0 & & & . \\ 0 & a_{32} & a_{33} & a_{43} & 0 & & . \\ . & 0 & a_{43} & a_{44} & . & . & . \\ . & & 0 & . & . & . & . \\ . & & & . & . & . & . \\ & & & & . & . & . \\ 0 & . & . & . & & . & a_{nn} \end{bmatrix}$$

In Storage

Only the diagonal and subdiagonal elements of the positive definite symmetric tridiagonal matrix are stored. This is called symmetric-tridiagonal storage mode. The elements of a symmetric tridiagonal matrix ***A*** of order *n* are stored in two one-dimensional arrays C and D, each of length *n*, where array C contains the subdiagonal elements, stored as follows:

C = (*, a_{21}, a_{32}, a_{43}, ..., $a_{n,n-1}$)

where "*" means you do not store an element in that position in the array. Then array D contains the main diagonal elements,stored as follows:

D = (a_{11}, a_{22}, a_{33}, ..., a_{nn})

Following is an example of a symmetric tridiagonal matrix ***A*** of order 5:

$$\begin{bmatrix} 10 & 1 & 0 & 0 & 0 \\ 1 & 20 & 2 & 0 & 0 \\ 0 & 2 & 30 & 3 & 0 \\ 0 & 0 & 3 & 40 & 4 \\ 0 & 0 & 0 & 4 & 50 \end{bmatrix}$$

which you store in symmetric-tridiagonal storage mode in arrays `C` and `D`, each of length 5, as follows:

```
C = (*, 1, 2, 3, 4)

D = (10, 20, 30, 40, 50)
```

Note: Some ESSL subroutines provide an option for specifying at least *n* additional locations at the end of each of the arrays `C` and `D`. These additional locations are used for working storage by the ESSL subroutine. The reasons for choosing this option are explained in the subroutine descriptions.

Positive Definite Symmetric Tridiagonal Matrix

A real symmetric tridiagonal matrix ***A*** is positive definite if and only if $\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x}$ is positive for all nonzero vectors ***x***.

In Storage

The positive definite symmetric tridiagonal matrix is stored in the same way the symmetric tridiagonal matrix is stored. For a description of this storage technique, see "Symmetric Tridiagonal Matrix" on page 91.

Sparse Matrix

A sparse matrix is a matrix having a relatively small number of nonzero elements.

Consider the following as an example of a sparse matrix ***A***:

$$\begin{bmatrix} 11 & 0 & 13 & 0 & 0 & 0 \\ 21 & 22 & 0 & 24 & 0 & 0 \\ 0 & 32 & 33 & 0 & 35 & 0 \\ 0 & 0 & 43 & 44 & 0 & 46 \\ 51 & 0 & 0 & 54 & 55 & 0 \\ 61 & 62 & 0 & 0 & 65 & 66 \end{bmatrix}$$

In Storage

A sparse matrix can be stored in full-matrix storage mode or a packed storage mode. When a sparse matrix is stored in **full-matrix storage mode**, all its elements, including its zero elements, are stored in an array.

The seven packed storage modes used for storing sparse matrices are described in the following sections:

- "Compressed-Matrix Storage Mode" on page 93
- "Compressed-Diagonal Storage Mode" on page 94
- "Storage-by-Indices" on page 97

- "Storage-by-Columns" on page 98
- "Storage-by-Rows" on page 99
- "Diagonal-Out Skyline Storage Mode" on page 100
- "Profile-In Skyline Storage Mode" on page 103

Note: When the elements of a sparse matrix are stored using any of these storage modes, the ESSL subroutines do not check that all elements are nonzero. You do not get an error if any elements are zero.

Compressed-Matrix Storage Mode: The sparse matrix ***A***, stored in compressed-matrix storage mode, uses two two-dimensional arrays to define the sparse matrix storage, AC and KA. See reference [67]. Given the *m* by *n* sparse matrix ***A***, having a maximum of *nz* nonzero elements in each row:

- AC is defined as AC(*lda*,*nz*), where the leading dimension, *lda*, must be greater than or equal to *m*. Each row of array AC contains the nonzero elements of the corresponding row of matrix ***A***. For each row in matrix ***A*** containing less than *nz* nonzero elements, the corresponding row in array AC is padded with zeros. The elements in each row can be stored in any order.
- KA is an integer array defined as KA(*lda*,*nz*), where the leading dimension, *lda*, must be greater than or equal to *m*. It contains the column numbers of the matrix ***A*** elements that are stored in the corresponding positions in array AC. For each row in matrix ***A*** containing less than *nz* nonzero elements, the corresponding row in array KA is padded with any values from 1 to *n*. **Because this array is used by the ESSL subroutines to access other target vectors in the computation, you must adhere to these required values to avoid errors.**

Unless all the rows of sparse matrix *A* contain approximately the same number of nonzero elements, this storage mode requires a large amount of storage. This diminishes the performance you can obtain by using this storage mode.

Consider the following as an example of a 6 by 6 sparse matrix ***A*** with a maximum of four nonzero elements in each row. It shows how matrix ***A*** can be stored in arrays AC and KA.

Given the following matrix ***A***:

$$\begin{bmatrix} 11 & 0 & 13 & 0 & 0 & 0 \\ 21 & 22 & 0 & 24 & 0 & 0 \\ 0 & 32 & 33 & 0 & 35 & 0 \\ 0 & 0 & 43 & 44 & 0 & 46 \\ 51 & 0 & 0 & 54 & 55 & 0 \\ 61 & 62 & 0 & 0 & 65 & 66 \end{bmatrix}$$

the arrays are:

$$AC = \begin{bmatrix} 11 & 13 & 0 & 0 \\ 22 & 21 & 24 & 0 \\ 33 & 32 & 35 & 0 \\ 44 & 43 & 46 & 0 \\ 55 & 51 & 54 & 0 \\ 66 & 61 & 62 & 65 \end{bmatrix}$$

$$KA = \begin{bmatrix} 1 & 3 & * & * \\ 2 & 1 & 4 & * \\ 3 & 2 & 5 & * \\ 4 & 3 & 6 & * \\ 5 & 1 & 4 & * \\ 6 & 1 & 2 & 5 \end{bmatrix}$$

where "*" means you can store any value from 1 to 6 in that position in the array.

Symmetric sparse matrices use the same storage technique as nonsymmetric sparse matrices; that is, all nonzero elements of a symmetric matrix ***A*** must be stored in array AC, not just the elements of the upper triangle and diagonal of matrix ***A***.

In general terms, this storage technique can be expressed as follows:

For each $a_{ij} \neq 0$, for $i = 1, m$ and $j = 1, n$
there exists k, where $1 \leq k \leq nz$,
such that $AC(i,k) = a_{ij}$ and $KA(i,k) = j$.

For all other elements of AC and KA,
$AC(i,k) = 0$ and $1 \leq KA(i,k) \leq n$

where:

- a_{ij} are the elements of the m by n matrix ***A*** that has a maximum of nz nonzero elements in each row.
- Array AC is defined as $AC(lda,nz)$, where $lda \geq m$.
- Array KA is defined as $KA(lda,nz)$, where $lda \geq m$.

Compressed-Diagonal Storage Mode: The storage mode used for square sparse matrices stored in compressed-diagonal storage mode has two variations, depending on whether the matrix is a general sparse matrix or a symmetric sparse matrix. This section explains both of these variations. This section begins, however, by explaining the conventions used for numbering the diagonals in the matrix, which apply to the storage descriptions.

Matrix ***A*** of order n has $2n-1$ diagonals. Because $k = j-i$ is constant for the elements a_{ij} along each diagonal, each diagonal can be assigned a diagonal number, k, having a value from $1-n$ to $n-1$. Then the diagonals can be referred to as $\boldsymbol{d}_k$, where $k = 1-n, n-1$.

The following matrix shows the starting position of each diagonal, $\boldsymbol{d}_k$:

$$
A = \begin{array}{c} \\ \boldsymbol{d}_{-1} \\ \boldsymbol{d}_{-2} \\ . \\ . \\ . \\ \boldsymbol{d}_{1-n} \end{array}
\begin{array}{c}
\begin{array}{ccccccc} \boldsymbol{d}_0 & \boldsymbol{d}_1 & \boldsymbol{d}_2 & . & . & . & \boldsymbol{d}_{n-1} \end{array} \\
\begin{bmatrix}
a_{11} & a_{12} & a_{13} & . & . & . & a_{1n} \\
a_{21} & a_{22} & a_{23} & & & & . \\
a_{31} & a_{32} & a_{33} & & & & . \\
. & & & . & & & . \\
. & & & & . & & . \\
. & & & & & . & . \\
a_{n1} & . & . & . & . & . & a_{nn}
\end{bmatrix}
\end{array}
$$

For a **general** (square) sparse matrix ***A***, compressed-diagonal storage mode uses two arrays to define the sparse matrix storage, AD and LA. Using the above convention for numbering the diagonals, and given that sparse matrix ***A*** contains *nd* diagonals having nonzero elements, arrays AD and LA are set up as follows:

- AD is defined as AD(*lda*,*nd*), where the leading dimension, *lda*, must be greater than or equal to *n*. Each diagonal of matrix ***A*** that has at least one nonzero element is stored in a column of array AD. All of the elements of the diagonal, including its zero elements, are stored in *n* contiguous locations in the array, in the same order as they appear in the diagonal. Padding with zeros is required as follows to fill the *n* locations in each column of array AD:
  - Each superdiagonal ($k > 0$), which has $n-k$ elements, is padded with *k* trailing zeros.
  - The main diagonal ($k = 0$), which has *n* elements, does not require padding.
  - Each subdiagonal ($k < 0$), which has $n-|k|$ elements, is padded with $|k|$ leading zeros.

  The diagonals can be stored in any columns in array AD.
- LA is a one-dimensional integer array of length *nd*, containing the diagonal numbers *k* for the diagonals stored in each corresponding column in array AD.

Because this storage mode requires entire diagonals to be stored, if the nonzero elements in matrix *A* are not concentrated along a few diagonals, this storage mode requires a large amount of storage. This diminishes the performance you obtain by using this storage mode.

Consider the following as an example of how a 6 by 6 general sparse matrix ***A*** with 5 nonzero diagonals is stored in arrays AD and LA.

Given the following matrix ***A***:

```
| 11   0  13   0   0   0 |
| 21  22   0  24   0   0 |
|  0  32  33   0  35   0 |
|  0   0  43  44   0  46 |
| 51   0   0  54  55   0 |
| 61  62   0   0  65  66 |
```

the arrays are:

$$AD = \begin{bmatrix} 11 & 13 & 0 & 0 & 0 \\ 22 & 24 & 21 & 0 & 0 \\ 33 & 35 & 32 & 0 & 0 \\ 44 & 46 & 43 & 0 & 0 \\ 55 & 0 & 54 & 51 & 0 \\ 66 & 0 & 65 & 62 & 61 \end{bmatrix}$$

$$LA = (0, 2, -1, -4, -5)$$

For a **symmetric** sparse matrix, where each superdiagonal k is equal to subdiagonal $-k$, compressed-diagonal storage mode uses the same storage technique as for the general sparse matrix, except that only the nonzero main diagonal and one diagonal of each couple of nonzero diagonals, k and $-k$, are used in setting up arrays AD and LA. You can store either the upper or the lower diagonal of each couple.

Consider the following as an example of a symmetric sparse matrix of order 6 and how it is stored in arrays AD and LA, using only three nonzero diagonals in the matrix.

Given the following matrix ***A***:

$$\begin{bmatrix} 11 & 0 & 13 & 0 & 51 & 0 \\ 0 & 22 & 0 & 24 & 0 & 62 \\ 13 & 0 & 33 & 0 & 35 & 0 \\ 0 & 24 & 0 & 44 & 0 & 46 \\ 51 & 0 & 35 & 0 & 55 & 0 \\ 0 & 62 & 0 & 46 & 0 & 66 \end{bmatrix}$$

the arrays are:

$$AD = \begin{bmatrix} 11 & 13 & 0 \\ 22 & 24 & 0 \\ 33 & 35 & 0 \\ 44 & 46 & 0 \\ 55 & 0 & 51 \\ 66 & 0 & 62 \end{bmatrix}$$

$$LA = (0, 2, -4)$$

In general terms, this storage technique can be expressed as follows:

For each $\boldsymbol{d}_k \neq (0, ..., 0)$, for $k = 1-n, n-1$
  for **general** square sparse matrices, or
for each unique $\boldsymbol{d}_k \neq (0, ..., 0)$, for $k = 1-n, n-1$
  for **symmetric** sparse matrices,
there exists l, where $1 \leq l \leq nd$,
such that LA(l) $= k$ and column l in array AD contains $\boldsymbol{dp}_k$.

where:

- Array AD is defined as AD(*lda*,*nd*), where $lda \geq n$, and where *nd* is the number of nonzero diagonals, $\boldsymbol{d}_k$ that are stored in array AD.

- Array LA has *nd* elements.
- *k* is the diagonal number of each diagonal, $\boldsymbol{d}_k$, where $k = i-j$.
- $\boldsymbol{dp}_k$ are the diagonals, $\boldsymbol{d}_k$, with padding, which are constructed from the sparse matrix $\boldsymbol{A}$ elements, a_{ij}, for $i, j = 1, n$ as follows:

  For superdiagonals ($k > 0$), $\boldsymbol{dp}_k$ has k trailing zeros: $\boldsymbol{dp}_k = (a_{1,k+1}, a_{2,k+2}, ..., a_{n-k,n}, 0_1, ..., 0_k)$
  For the main diagonal ($k = 0$), $\boldsymbol{dp}_0$ has no padding: $\boldsymbol{dp}_0 = (a_{11}, a_{22}, ..., a_{nn})$
  For subdiagonals ($k < 0$), $\boldsymbol{dp}_k$ has $|k|$ leading zeros: $\boldsymbol{dp}_k = (0_1, ..., 0_{|k|}, a_{|k|+1,1}, a_{|k|+2,2}, ..., a_{n,\, n-|k|})$

Storage-by-Indices: For a sparse matrix $\boldsymbol{A}$, storage-by-indices uses three one-dimensional arrays to define the sparse matrix storage, AR, IA, and JA. Given the *m* by *n* sparse matrix $\boldsymbol{A}$ having *ne* nonzero elements, the arrays are set up as follows:

- AR of (at least) length *ne* contains the *ne* nonzero elements of the sparse matrix $\boldsymbol{A}$, stored contiguously in **any** order.
- IA, an integer array of (at least) length *ne* contains the corresponding row numbers of each nonzero element, a_{ij}, in matrix $\boldsymbol{A}$.
- JA, an integer array of (at least) length *ne* contains the corresponding column numbers of each nonzero element, a_{ij}, in matrix $\boldsymbol{A}$.

Consider the following as an example of a 6 by 6 sparse matrix $\boldsymbol{A}$ and how it can be stored in arrays AR, IA, and JA.:

Given the following matrix $\boldsymbol{A}$:

$$\begin{bmatrix} 11 & 0 & 13 & 0 & 0 & 0 \\ 21 & 22 & 0 & 24 & 0 & 0 \\ 0 & 32 & 33 & 0 & 35 & 0 \\ 0 & 0 & 43 & 44 & 0 & 46 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 61 & 62 & 0 & 0 & 65 & 66 \end{bmatrix}$$

the arrays are:

```
AR = (11, 22, 32, 33, 13, 21, 43, 24, 66, 46, 35, 62, 61, 65, 44)

IA = (1, 2, 3, 3, 1, 2, 4, 2, 6, 4, 3, 6, 6, 6, 4)

JA = (1, 2, 2, 3, 3, 1, 3, 4, 6, 6, 5, 2, 1, 5, 4)
```

In general terms, this storage technique can be expressed as follows:

For each $a_{ij} \neq 0$, for $i = 1, m$ and $j = 1, n$
there exists k, where $1 \leq k \leq ne$, such that:
AR(k) = a_{ij}
IA(k) = i
JA(k) = j

where:

- a_{ij} are the elements of the *m* by *n* sparse matrix $\boldsymbol{A}$.

- Arrays AR, IA, and JA each have *ne* elements.

Storage-by-Columns: For a sparse matrix, ***A***, storage-by-columns uses three one-dimensional arrays to define the sparse matrix storage, AR, IA, and JA. Given the *m* by *n* sparse matrix ***A*** having *ne* nonzero elements, the arrays are set up as follows:

- AR of (at least) length *ne* contains the *ne* nonzero elements of the sparse matrix ***A***, stored contiguously. The columns of matrix ***A*** are stored consecutively from 1 to *n* in AR. The elements in each column of ***A*** are stored in any order in AR.
- IA, an integer array of (at least) length *ne* contains the corresponding row numbers of each nonzero element, a_{ij}, in matrix ***A***.
- JA, an integer array of (at least) length *n*+1 contains the relative starting position of each column of matrix ***A*** in array AR; that is, each element JA(*j*) of the column pointer array indicates where column *j* begins in array AR. If all elements in column *j* are zero, then JA(*j*) = JA(*j*+1). The last element, JA(*n*+1), indicates the position after the last element in array AR, which is *ne*+1.

Consider the following as an example of a 6 by 6 sparse matrix ***A*** and how it can be stored in arrays AR, IA, and JA.

Given the following matrix ***A***:

$$\begin{bmatrix} 11 & 0 & 13 & 0 & 0 & 0 \\ 21 & 22 & 0 & 24 & 0 & 0 \\ 0 & 32 & 33 & 0 & 0 & 0 \\ 0 & 0 & 43 & 44 & 0 & 46 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 61 & 62 & 0 & 0 & 0 & 66 \end{bmatrix}$$

the arrays are:

```
AR = (11, 61, 21, 62, 32, 22, 13, 33, 43, 44, 24, 46, 66)

IA = (1, 6, 2, 6, 3, 2, 1, 3, 4, 4, 2, 4, 6)

JA = (1, 4, 7, 10, 12, 12, 14)
```

In general terms, this storage technique can be expressed as follows:

For each $a_{ij} \neq 0$, for $i = 1, m$ and $j = 1, n$
there exists k, where $1 \leq k \leq ne$, such that
  AR(k) = a_{ij}
  IA(k) = i
And for $j = 1, n$,
  JA(j) = k, where a_{ij}, in AR(k), is the first element stored in AR for column j
  JA(j) = JA(j+1), where all $a_{ij} = 0$ in column j
  JA(n+1) = ne+1

where:

- a_{ij} are the elements of the *m* by *n* sparse matrix ***A***.
- Arrays AR and IA each have *ne* elements.
- Array JA has *n*+1 elements.

Storage-by-Rows: The storage mode used for sparse matrices stored by rows has three variations, depending on whether the matrix is a general sparse matrix or a symmetric sparse matrix. This section explains these variations.

For a **general** sparse matrix ***A***, storage-by-rows uses three one-dimensional arrays to define the sparse matrix storage, AR, IA, and JA. Given the *m* by *n* sparse matrix ***A*** having *ne* nonzero elements, the arrays are set up as follows:

- AR of (at least) length *ne* contains the *ne* nonzero elements of the sparse matrix ***A***, stored contiguously. The rows of matrix ***A*** are stored consecutively from 1 to *m* in AR. The elements in each row of ***A*** are stored in any order in AR.
- IA, an integer array of (at least) length *m*+1 contains the relative starting position of each row of matrix ***A*** in array AR; that is, each element IA(*i*) of the row pointer array indicates where row *i* begins in array AR. If all elements in row *i* are zero, then IA(*i*) = IA(*i*+1). The last element, IA(*m*+1), indicates the position after the last element in array AR, which is *ne*+1.
- JA, an integer array of (at least) length *ne* contains the corresponding column numbers of each nonzero element, a_{ij}, in matrix ***A***.

Consider the following as an example of a 6 by 6 general sparse matrix ***A*** and how it can be stored in arrays AR, IA, and JA.

Given the following matrix ***A***:

$$\begin{bmatrix} 11 & 0 & 13 & 0 & 0 & 0 \\ 21 & 22 & 0 & 24 & 0 & 0 \\ 0 & 32 & 33 & 0 & 0 & 0 \\ 0 & 0 & 43 & 44 & 0 & 46 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 61 & 62 & 0 & 0 & 0 & 66 \end{bmatrix}$$

the arrays are:

```
AR = (11, 13, 24, 22, 21, 32, 33, 44, 43, 46, 61, 62, 66)

IA = (1, 3, 6, 8, 11, 11, 14)

JA = (1, 3, 4, 2, 1, 2, 3, 4, 3, 6, 1, 2, 6)
```

For a **symmetric** sparse matrix of order *m*, storage-by-rows uses the same storage technique as for the general sparse matrix, except that only the upper or lower triangle and diagonal elements are used in setting up arrays AR, IA, and JA.

Consider the following as an example of a symmetric sparse matrix ***A*** of order 6 and how it can be stored in arrays AR, IA, and JA using upper-storage-by-rows, which stores only the upper triangle and diagonal elements.

Given the following matrix A:

$$\begin{bmatrix} 11 & 0 & 13 & 0 & 0 & 0 \\ 0 & 22 & 23 & 24 & 0 & 0 \\ 13 & 23 & 33 & 0 & 35 & 0 \\ 0 & 24 & 0 & 44 & 0 & 46 \\ 0 & 0 & 35 & 0 & 55 & 0 \\ 0 & 0 & 0 & 46 & 0 & 0 \end{bmatrix}$$

the arrays are:

```
AR = (11, 13, 22, 24, 23, 33, 35, 46, 44, 55)

IA = (1, 3, 6, 8, 10, 11, 11)

JA = (1, 3, 2, 3, 4, 3, 5, 4, 6, 5)
```

Using the same symmetric matrix ***A***, consider the following as an example of how it can be stored in arrays AR, IA, and JA using lower-storage-by-rows, which stores only the lower triangle and diagonal elements:

```
AR = (11, 22, 23, 33, 13, 24, 44, 55, 35, 46)

IA = (1, 2, 3, 6, 8, 10, 11 )

JA = (1, 2, 2, 3, 1, 2, 4, 5, 3, 4)
```

In general terms, this storage technique can be expressed as follows:

For each $a_{ij} \neq 0$,
  for $i = 1, m$ and $j = 1, n$ for general sparse matrices
    or
  for $i = 1, m$ and $j = i, m$ for symmetric sparse matrices using the lower triangle
    or
  for $i = 1, m$ and $j = 1, i$ for symmetric sparse matrices using the upper triangle
there exists k, where $1 \leq k \leq ne$, such that
  AR(k) = a_{ij}
  JA(k) = j
And for $i = 1, m$,
  IA(i) = k, where a_{ij}, in AR(k), is the first element stored in AR for row i
  IA(i) = IA(i+1), where all $a_{ij} = 0$ in row i
  IA(m+1) = ne+1

where:

- a_{ij} are the elements of sparse matrix ***A***, which is either an m by n general sparse matrix or a symmetric sparse matrix of order m containing ne nonzero elements.
- Arrays AR and JA each have ne elements.
- Array IA has m+1 elements.

Diagonal-Out Skyline Storage Mode: The diagonal-out skyline storage mode used for sparse matrices has two variations, depending on whether the matrix is a general sparse matrix or a symmetric sparse matrix. Both of these variations are explained here.

For a **general** sparse matrix ***A***, diagonal-out skyline storage mode uses four one-dimensional arrays to define the sparse matrix storage, AU, IDU, AL, and IDL. Given the sparse matrix ***A*** of order *n*, containing *nu*+*nl*–*n* elements under the top and left profiles, the arrays are set up as follows:

- AU of (at least) length *nu* contains the upper triangle of the sparse matrix ***A***, where the columns are stored consecutively from 1 to *n* in AU in the following way. For each column, the elements starting at the diagonal element and ending at the topmost nonzero element in the column are stored contiguously in AU. The elements stored may include zero elements along with the nonzero elements. If all elements in the column to be stored are zero, the diagonal element, a_{ii}, having a value of zero, is stored in AU for that column. A total of *nu* elements are stored for the upper triangle of ***A***.
- IDU, an integer array of (at least) length *n*+1 contains the relative position of each diagonal element of matrix ***A*** in array AU; that is, each element IDU(*i*) of the diagonal pointer array indicates where diagonal element a_{ii} is stored in array AU. One-origin is used, so the first element of IDU is always 1. The last element, IDU(*n*+1), indicates the position after the last element in array AU, which is *nu*+1.
- AL of (at least) length *nl* contains the lower triangle of the sparse matrix ***A***, where the rows are stored consecutively from 1 to *n* in AL in the following way. For each row, the elements starting at the diagonal element and ending at the leftmost nonzero element in the row are stored contiguously in AL. The elements stored may include zero elements along with the nonzero elements. If all elements in the row to be stored are zero, the diagonal element, a_{ii}, having a value of zero, is stored in AL for that row. A total of *nl* elements are stored for the lower triangle of ***A***. The values of the diagonal elements are meaningless, so you can store any values in those positions in AL.
- IDL, an integer array of (at least) length *n*+1 contains the relative position of each diagonal element of matrix ***A*** in array AL; that is, each element IDL(*i*) of the diagonal pointer array indicates where diagonal element a_{ii} is stored in array AL. One-origin is used, so the first element of IDL is always 1. The last element, IDL(*n*+1), indicates the position after the last element in array AL, which is *nl*+1.

Consider the following as an example of a 6 by 6 general sparse matrix ***A*** and how it is stored in arrays AU, IDU, AL, and IDL.

Given the following matrix ***A***:

$$\begin{bmatrix} 0 & 12 & 13 & 0 & 0 & 0 \\ 21 & 22 & 0 & 24 & 0 & 0 \\ 31 & 0 & 33 & 34 & 0 & 36 \\ 41 & 42 & 43 & 44 & 45 & 0 \\ 0 & 0 & 0 & 54 & 55 & 56 \\ 0 & 0 & 63 & 0 & 65 & 66 \end{bmatrix}$$

the arrays are:

```
AU = (0, 22, 12, 33, 0, 13, 44, 34, 24, 55, 45, 66, 56, 0, 36)

IDU = (1, 2, 4, 7, 10, 12, 16) where nu=15

AL = (*, *, 21, *, 0, 31, *, 43, 42, 41, *, 54, *, 65, 0, 63)
```

IDL = (1, 2, 4, 7, 11, 13, 17) where *nl*=16

and where "*" means you do not have to store a value in that position in the array. However, these storage positions are required.

For a **symmetric** sparse matrix of order *n*, diagonal-out skyline storage mode uses the same storage technique as for the upper triangle and diagonal elements of the general sparse matrix; therefore, only the AU and IDU arrays are needed.

Consider the following as an example of a symmetric sparse matrix ***A*** of order 6 and how it is stored in arrays AU and IDU.

Given the following matrix ***A***:

$$\begin{bmatrix} 0 & 12 & 13 & 0 & 0 & 0 \\ 12 & 22 & 0 & 24 & 0 & 0 \\ 13 & 0 & 33 & 34 & 0 & 36 \\ 0 & 24 & 34 & 44 & 45 & 0 \\ 0 & 0 & 0 & 45 & 55 & 56 \\ 0 & 0 & 36 & 0 & 56 & 66 \end{bmatrix}$$

the arrays are:

AU = (0, 22, 12, 33, 0, 13, 44, 34, 24, 55, 45, 66, 56, 0, 36)

IDU = (1, 2, 4, 7, 10, 12, 16) where *nu*=15

In general terms, this storage technique can be expressed as follows:

For general sparse matrices and symmetric sparse matrices:
For each a_{ij} for $j = 1, n$ and $i = j, k$,
where a_{kj} is the topmost $a_{ij} \neq 0$ in each column j,
there exists m, where $1 \leq m \leq nu$, such that
AU(m+j–i) = a_{ij}
IDU(j) = m for each a_{jj}
IDU(n+1) = nu+1

Also, for general sparse matrices:
For each a_{ij} for $i = 1, n$ and $i = j, k$,
where a_{ik} is the leftmost $a_{ij} \neq 0$ in each row i,
there exists m, where $1 \leq m \leq nl$, such that
AL(m+i–j) = a_{ij}
IDL(i) = m for each a_{ii}
IDL(n+1) = nl+1

where:

- a_{ij} are the elements of sparse matrix ***A***, of order n.
- Array AU has nu elements.
- Array AL has nl elements.
- Arrays IDU and IDL each have n+1 elements.

Profile-In Skyline Storage Mode: The profile-in skyline storage mode used for sparse matrices has two variations, depending on whether the matrix is a general sparse matrix or a symmetric sparse matrix. Both of these variations are explained here.

For a **general** sparse matrix ***A***, profile-in skyline storage mode uses four one-dimensional arrays to define the sparse matrix storage, AU, IDU, AL, and IDL. Given the sparse matrix ***A*** of order *n*, containing *nu*+*nl*–*n* elements under the top and left profiles, the arrays are set up as follows:

- AU of (at least) length *nu* contains the upper triangle of the sparse matrix ***A***, where the columns are stored consecutively from 1 to *n* in AU in the following way. For each column, the elements starting at the topmost nonzero element in the column and ending at the diagonal element are stored contiguously in AU. The elements stored may include zero elements along with the nonzero elements. If all elements in the column to be stored are zero, the diagonal element, a_{ii}, having a value of zero, is stored in AU for that column. A total of *nu* elements are stored for the upper triangle of ***A***.
- IDU, an integer array of (at least) length *n*+1 contains the relative position of each diagonal element of matrix ***A*** in array AU; that is, each element IDU(*i*) of the diagonal pointer array indicates where diagonal element a_{ii} is stored in array AU. One-origin is used, so the first element of IDU is always 1. The last element, IDU(*n*+1), indicates the position after the last element in array AU, which is *nu*+1.
- AL of (at least) length *nl* contains the lower triangle of the sparse matrix ***A***, where the rows are stored consecutively from 1 to *n* in AL in the following way. For each row, the elements starting at the leftmost nonzero element in the row and ending at the diagonal element are stored contiguously in AL. The elements stored may include zero elements along with the nonzero elements. If all elements in the row to be stored are zero, the diagonal element, a_{ii}, having a value of zero, is stored in AL for that row. A total of *nl* elements are stored for the lower triangle of ***A***. The values of the diagonal elements are meaningless, so you can store any values in those positions in AL.
- IDL, an integer array of (at least) length *n*+1 contains the relative position of each diagonal element of matrix ***A*** in array AL; that is, each element IDL(*i*) of the diagonal pointer array indicates where diagonal element a_{ii} is stored in array AL. One-origin is used, so the first element of IDL is always 1. The last element, IDL(*n*+1), indicates the position after the last element in array AL, which is *nl*+1.

Consider the following as an example of a 6 by 6 general sparse matrix ***A*** and how it is stored in arrays AU, IDU, AL, and IDL.

Given the following matrix ***A***:

$$\begin{bmatrix} 0 & 12 & 13 & 0 & 0 & 0 \\ 21 & 22 & 0 & 24 & 0 & 0 \\ 31 & 0 & 33 & 34 & 0 & 36 \\ 41 & 42 & 43 & 44 & 45 & 0 \\ 0 & 0 & 0 & 54 & 55 & 56 \\ 0 & 0 & 63 & 0 & 65 & 66 \end{bmatrix}$$

the arrays are:

```
AU = (0, 12, 22, 13, 0, 33, 24, 34, 44, 45, 55, 36, 0, 56, 66)
```

IDU = (1, 3, 6, 9, 11, 15, 16) where *nu*=15

AL = (*, 21, *, 31, 0, *, 41, 42, 43, *, 54, *, 63, 0, 65, *)

IDL = (1, 3, 6, 10, 12, 16, 17) where *nl*=16

and where "*" means you do not have to store a value in that position in the array. However, these storage positions are required.

For a **symmetric** sparse matrix of order *n*, profile-in skyline storage mode uses the same storage technique as for the upper triangle and diagonal elements of the general sparse matrix; therefore, only the AU and IDU arrays are needed.

Consider the following as an example of a symmetric sparse matrix ***A*** of order 6 and how it is stored in arrays AU and IDU.

Given the following matrix ***A***:

$$\begin{bmatrix} 0 & 12 & 13 & 0 & 0 & 0 \\ 12 & 22 & 0 & 24 & 0 & 0 \\ 13 & 0 & 33 & 34 & 0 & 36 \\ 0 & 24 & 34 & 44 & 45 & 0 \\ 0 & 0 & 0 & 45 & 55 & 56 \\ 0 & 0 & 36 & 0 & 56 & 66 \end{bmatrix}$$

the arrays are:

AU = (0, 12, 22, 13, 0, 33, 24, 34, 44, 45, 55, 36, 0, 56, 66)

IDU = (1, 3, 6, 9, 11, 15, 16) where *nu*=15

In general terms, this storage technique can be expressed as follows:

For general sparse matrices and symmetric sparse matrices:
For each a_{ij} for $j = 1, n$ and $i = k, j$,
where a_{kj} is the topmost $a_{ij} \neq 0$ in each column j,
there exists m, where $1 \leq m \leq nu$, such that
AU($m-j+i$) = a_{ij}
IDU(j) = m for each a_{jj}
IDU($n+1$) = $nu+1$

Also, for general sparse matrices:
For each a_{ij} for $i = 1, n$ and $j = k, i$,
where a_{ik} is the leftmost $a_{ij} \neq 0$ in each row i,
there exists m, where $1 \leq m \leq nl$, such that
AL($m-i+j$) = a_{ij}
IDL(i) = m for each a_{ii}
IDL($n+1$) = $nl+1$

where:

- a_{ij} are the elements of sparse matrix ***A***, of order n.
- Array AU has nu elements.
- Array AL has nl elements.

- Arrays IDU and IDL each have n+1 elements.

Sequences

A sequence is an ordered collection of numbers. It can be a one-, two-, or three-dimensional sequence. Sequences are used in the areas of sorting, searching, Fourier transforms, convolutions, and correlations.

Real and Complex Elements in Storage

Sequences can contain either real or complex data. For sequences containing complex data, a special storage arrangement is used to accommodate the two parts, a and b, of each complex number, $a+bi$, in the array. For each complex number, two sequential storage locations are required in the array. Therefore, exactly twice as much storage is required for complex sequences as for real sequences of the same precision. See "How Do You Set Up Your Scalar Data?" on page 28 for a description of real and complex numbers, and "How Do You Set Up Your Arrays?" on page 29 for a description of how real and complex data is stored in arrays.

One-Dimensional Sequences

A one-dimensional sequence appears symbolically as follows, where the subscripts indicate the element positions within the sequence:

$(x_1, x_2, x_3, \ldots x_n)$

In Storage

A one-dimensional sequence is stored in an array using stride in the same way a vector uses stride. For details, see "How Stride Is Used for Vectors" on page 58.

Two-Dimensional Sequences

A two-dimensional sequence appears symbolically as a series of columns of elements. (They are represented in the same way as a matrix without the square brackets.) The two subscripts indicate the element positions in the first and second dimensions, respectively:

$$\begin{matrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{matrix}$$

In Storage

A two-dimensional sequence is stored in an array using the stride for the second dimension in the same way that a matrix uses leading dimension. It uses a stride of 1 for the first dimension. For details, see "How Leading Dimension Is Used for Matrices" on page 64. (In the area of Fourier transforms, a two-dimensional sequence may be stored in transposed form in an array. In this case, the stride for the second dimension is 1, and the stride for the first dimension is the leading dimension of the array.)

Three-Dimensional Sequences

A three-dimensional sequence is represented as a series of blocks of elements. Each block is equivalent to a two-dimensional sequence. The number of blocks indicates the length of the third dimension. The three subscripts indicate the element positions in the first, second, and third dimensions, respectively:

Plane 0:

$$\begin{matrix} a_{0,0,0} & \cdots & a_{0,n-1,0} \\ a_{1,0,0} & \cdots & a_{1,n-1,0} \\ \vdots & & \vdots \\ a_{m-1,0,0} & \cdots & a_{m-1,n-1,0} \end{matrix}$$

Plane 1:

$$\begin{matrix} a_{0,0,1} & \cdots & a_{0,n-1,1} \\ a_{1,0,1} & \cdots & a_{1,n-1,1} \\ \vdots & & \vdots \\ a_{m-1,0,1} & \cdots & a_{m-1,n-1,1} \end{matrix}$$

$$\vdots$$

Plane $(p-1)$:

$$\begin{matrix} a_{0,0,p-1} & \cdots & a_{0,n-1,p-1} \\ a_{1,0,p-1} & \cdots & a_{1,n-1,p-1} \\ \vdots & & \vdots \\ a_{m-1,0,p-1} & \cdots & a_{m-1,n-1,p-1} \end{matrix}$$

In Storage

Each block of elements in a three-dimensional sequence is stored successively in an array. The stride for the third dimension is used to select the elements for each successive block of elements in the array. The starting point of the three-dimensional sequence is specified as the argument for the sequence in the ESSL calling statement. For example, if the three-dimensional sequence is contained in array `BIG`, declared as `BIG(1:20,1:30,1:10)`, and starts at the second element in the first dimension, the third element in the second dimension, and the first element in the third dimension of array `BIG`, you should specify `BIG(2,3,1)` as the argument for the sequence, such as in:

```
CALL  SCFT3 (BIG(2,3,1),20,600,Y,32,2056,16,20,10,1,1.0,AUX,30000)
```

See "How Stride Is Used for Three-Dimensional Sequences" for a detailed description of how three-dimensional sequences are stored within arrays using strides.

How Stride Is Used for Three-Dimensional Sequences

The elements of the three-dimensional sequence can be defined as a_{ijk} for $i = 1, m$, $j = 1, n$, and $k = 1, p$. The first two subscripts, i and j, define the elements in the first two dimensions of the sequence, and the third subscript, k, defines the elements in the third dimension. Using this definition of three-dimensional sequences, this section explains how these elements are mapped into an array using the concepts of stride. (Remember that the elements a_{ijk} are the elements of the conceptual data structure, the three-dimensional sequence to be processed by ESSL. The sequence does not have to include all the elements in the array. Strides are used by the ESSL subroutines to select the desired elements to be processed in the array.)

The sequence elements in the first two dimensions are mapped into an array in the same way a matrix or two-dimensional sequence is mapped into an array. It uses all the items listed in "How Leading Dimension Is Used for Matrices" on page 64, such as the starting point, the number of rows and columns, and the leading dimension. The stride for the first dimension, *inc1*, of a three-dimensional sequence is assumed to be 1, as for matrices. The stride for the second dimension, *inc2*, of a three-dimensional sequence is equivalent to the leading dimension for a matrix.

The stride for the third dimension, *inc3*, is used to define the array elements that make up the third dimension of the three-dimensional sequence. The stride for the third dimension is used as an increment to step through the array to find the starting point for each of the p successive blocks of elements in the array. The stride, *inc3*, must always be positive. It must always be greater than or equal to the number of elements to be processed in the first two dimensions; that is, $inc3 \geq (inc2)(n)$.

A three-dimensional sequence is usually stored in a one-, two-, or three-dimensional array; however, for the sake of this discussion, a three-dimensional array is used here. For an array, `A`, declared as `A(E1:E2,F1:F2,G1:G2)`, the strides in the first, second, and third dimensions are:

inc1 = 1
inc2 = (E2–E1+1)
inc3 = (E2–E1+1)(F2–F1+1)

Given an array `A`, declared as `A(1:7,1:3,0:3)`, where the lengths of the first, second, and third dimensions are 7, 3, and 4, respectively, the resulting strides are *inc1* = 1, *inc2* = 7, and *inc3* = 21.

The starting point for a three-dimensional sequence in an array is at the location specified by the argument for the sequence in the ESSL calling statement. Using the array `A`, described above, if you specify `A(2,2,1)` for a three-dimensional sequence, where `A` is defined as follows, in four blocks, for planes 0 - 3, respectively:

```
1.0  8.0 15.0    22.0 29.0 36.0    43.0 50.0 57.0    64.0 71.0 78.0
2.0  9.0 16.0    23.0 30.0 37.0    44.0 51.0 58.0    65.0 72.0 79.0
3.0 10.0 17.0    24.0 31.0 38.0    45.0 52.0 59.0    66.0 73.0 80.0
4.0 11.0 18.0    25.0 32.0 39.0    46.0 53.0 60.0    67.0 74.0 81.0
5.0 12.0 19.0    26.0 33.0 40.0    47.0 54.0 61.0    68.0 75.0 82.0
6.0 13.0 20.0    27.0 34.0 41.0    48.0 55.0 62.0    69.0 76.0 83.0
7.0 14.0 21.0    28.0 35.0 42.0    49.0 56.0 63.0    70.0 77.0 84.0
```

then processing begins in the second block of elements at row 2 and column 2 in array `A`, which is 30.0. The stride in the third dimension is then used to find the starting point for each of the next $p-1$ successive blocks of elements in the array. The stride, *inc3*, is added to the starting point $p-1$ times. In this example, the stride for the third dimension is 21, and the number of blocks of elements, p, to be processed is 3, so the starting points in array `A` are `A(2,2,1)`, `A(2,2,2)`, and `A(2,2,3)`. These are elements 30.0, 51.0, and 72.0. These array elements then correspond to the sequence elements a_{111}, a_{112}, and a_{113}, respectively.

In general terms, this results in the following starting positions for the blocks of elements in the array:

```
A(BEGINI, BEGINJ, BEGINK)
A(BEGINI, BEGINJ, BEGINK+1)
A(BEGINI, BEGINJ, BEGINK+2)
.
.
A(BEGINI, BEGINJ, BEGINK+p-1)
```

Using $m = 4$, $n = 2$, and $p = 3$ to define the elements of the three-dimensional data structure in this example, the resulting three-dimensional sequence is defined as follows, in three blocks, for planes 0 - 2, respectively:

Plane 0:		**Plane 1:**		**Plane 2:**	
a_{000}	a_{010}	a_{001}	a_{011}	a_{002}	a_{012}
a_{100}	a_{110}	a_{101}	a_{111}	a_{102}	a_{112}
a_{200}	a_{210}	a_{201}	a_{211}	a_{202}	a_{212}
a_{300}	a_{310}	a_{301}	a_{311}	a_{302}	a_{312}

=

Plane 0:		**Plane 1:**		**Plane 2:**	
30.0	37.0	51.0	58.0	72.0	79.0
31.0	38.0	52.0	59.0	73.0	80.0
32.0	39.0	53.0	60.0	74.0	81.0
33.0	40.0	54.0	61.0	75.0	82.0

As shown in this example, the three-dimensional sequence does not have to include all the blocks of elements in the array. In this case, the three-dimensional sequence includes only the second through the fourth block of elements in the array. The first block is not used. Elements of an array are selected as they are arranged in storage, regardless of the number of dimensions defined in the array. Therefore, when using a one- or two-dimensional array to store your three-dimensional sequence, you should understand how your array elements are stored to ensure that elements are selected properly. See "Setting Up Arrays in Fortran" on page 112 for a description of array storage.

Note: Three-dimensional sequences are used by the three-dimensional Fourier transform subroutines. By specifying certain stride values for *inc2* and *inc3* and declaring your arrays to have certain number of dimensions, you achieve optimal performance in these subroutines. For details, see "Setting Up Your Data" on page 744 and the Notes section for each subroutine.

Chapter 4. Coding Your Program

This chapter provides you with information you need to code your Fortran, C, C++, and PL/I programs using ESSL.

Fortran Programs

This section describes how to code your Fortran program using any of the ESSL run-time libraries.

Calling ESSL Subroutines and Functions in Fortran

In Fortran programs, most ESSL subroutines are invoked with the CALL statement:

```
CALL  subroutine-name  (argument-1, . . . , argument-n)
```

An example of a calling sequence for the SAXPY subroutine might be:

```
CALL SAXPY (5,A,X,J+INC,Y,1)
```

The remaining ESSL subroutines are invoked as functions by coding a function reference. You first declare the type of value returned by the function: short- or long-precision real, short- or long-precision complex, or integer. Then you code the function reference as part of an expression in a statement. An example of declaring and invoking the DASUM function might be:

```
DOUBLE PRECISION DASUM,SUM,X
     .
     .
     .
SUM = DASUM (N,X,INCX)
```

Values are returned differently for ESSL subroutines and functions. For subroutines, the results of the computation are returned in an argument specified in the calling sequence. In the CALL statement above, the result is returned in argument `Y`. For functions, the result is returned as the value of the function. In the assignment statement above, the result is assigned to `SUM`.

See the following Fortran publications for details on how to code the CALL statement and a function reference:

- *IBM XL Fortran for AIX User's Guide Version 5*
- *IBM XL Fortran for AIX Language Reference Version 5*

Setting Up a User-Supplied Subroutine for ESSL in Fortran

Some ESSL numerical quadrature subroutines call a user-supplied subroutine, *subf*, identified in the ESSL calling sequence. If your program that calls the numerical quadrature subroutines is coded in Fortran, there are some coding rules you must follow:

- You must declare *subf* as EXTERNAL in your program.

© Copyright IBM Corp. 1997

- You should code the *subf* subroutine to the specifications given in " Programming Considerations for the SUBF Subroutine" on page 926. For examples of coding a *subf* subroutine in Fortran, see the subroutine descriptions in that chapter.

Setting Up Scalar Data in Fortran

Table 27 lists the scalar data types in Fortran that are used for ESSL. Only those types and lengths used by ESSL are listed.

Table 27. Scalar Data Types in Fortran Programs

Terminology Used by ESSL	**Fortran Equivalent**
Character item[1] 'N', 'T', 'C' or 'n', 't', 'c'	CHARACTER*1 'N', 'T', 'C'
Logical item .TRUE., .FALSE.	LOGICAL .TRUE., .FALSE.
Fullword integer 12345, –12345	INTEGER or INTEGER*4 12345, –12345
Short-precision real number[2] 12.345	REAL or REAL*4 0.12345E2
Long-precision real number[2] 12.345	DOUBLE PRECISION or REAL*8 0.12345D2
Short-precision complex number[2] (123.45, –54321.0)	COMPLEX or COMPLEX*8 (123.45E0, –543.21E2)
Long-precision complex number[2] (123.45, –54321.0)	COMPLEX*16 (123.45D0, –543.21D2)

[1] ESSL accepts character data in either upper- or lowercase in its calling sequences.

[2] Short- and long-precision numbers look the same in this book.

Setting Up Arrays in Fortran

Arrays are declared in Fortran by specifying the array name, the number of dimensions, and the range of each dimension in a DIMENSION statement or an explicit data type statement, such as REAL, DOUBLE PRECISION, and so forth.

Real and Complex Array Elements

Each array element can be either a real or complex data item of short or long precision. The type of the array determines the size of the element storage locations. Short-precision data requires 4 bytes, and long-precision data requires 8 bytes. Complex data requires two storage locations of either 4 or 8 bytes each, for short or long precision, respectively, to accommodate the two parts of the complex number: $c = a+bi$. Therefore, exactly twice as much storage is required for complex data as for real data of the same precision. See "How Do You Set Up Your Scalar Data?" on page 28 for a description of real and complex numbers.

Even though complex data items require two storage locations, the same number of elements exist in the array as for real data. A reference to an element—for example, C(3)—in an array containing complex data gives you the whole complex number; that is, it contains both *a* and *b*, where the complex number is expressed as follows:

C(I) ← (a_i, b_i) for a one-dimensional array
C(I,J) ← (a_{ij}, b_{ij}) for a two-dimensional array
C(I,J,K) ← (a_{ijk}, b_{ijk}) for a three-dimensional arra y

One-Dimensional Array

For a one-dimensional array in Fortran 77, you can code:

```
DIMENSION  A(E1:E2)
```

where A is the name of the array, E1 is the lower bound, and E2 is the upper bound of the single dimension in the array. If the lower bound is not specified, such as in A(E2), the value is assumed to be 1. The upper bound is required.

A one-dimensional array is stored in ascending storage locations (relative to some base storage address) in the following order:

Relative Location	**Array Element**
1	A(E1)
2	A(E1+1)
3	A(E1+2)
.	.
.	.
.	.
E2–E1+1	A(E2)

For example, the array A of length 4 specified in the DIMENSION statement as A(0:3) and containing the following elements:

A = (1, 2, 3, 4)

has its elements arranged in storage as follows:

Relative Location	**Array Element Value**
1	1
2	2
3	3
4	4

Two-Dimensional Array

For a two-dimensional array in Fortran 77, you can code:

```
DIMENSION  A(E1:E2,F1:F2)
```

where A is the name of the array. E1 and F1 are the lower bounds of the first and second dimensions, respectively, and E2 and F2 are the upper bounds of the first and second dimensions, respectively. If either of the lower bounds is not specified, such as in A(E2,F1:F2), the value is assumed to be 1. The upper bounds are always required for each dimension. For examples of Fortran 77 usage, see “SGEMV, DGEMV, CGEMV, ZGEMV, SGEMX, DGEMX, SGEMTX, and DGEMTX—Matrix-Vector Product for a General Matrix, Its Transpose, or Its Conjugate Transpose” on page 297.

The elements of a two-dimensional array are stored in column-major order; that is, they are stored in the following ascending storage locations (relative to some base storage address) with the value of the first (row) subscript expression increasing most rapidly and the value of the second (column) subscript expression increasing least rapidly. Following are the locations of the elements in the array:

Relative Location	**Array Element**
1	`A(E1,F1)` (starting column 1)
2	`A(E1+1,F1)`
.	.
.	.
.	.
E2–E1+1	`A(E2,F1)`
(E2–E1+1)+1	`A(E1,F1+1)` (starting column 2)
(E2–E1+1)+2	`A(E1+1,F1+1)`
.	.
.	.
.	.
(E2–E1+1)(2)	`A(E2,F1+1)`
(E2–E1+1)(2)+1	`A(E1,F1+2)` (starting column 3)
(E2–E1+1)(2)+2	`A(E1+1,F1+2)`
.	.
.	.
.	.
(E2–E1+1)(F2–F1)	`A(E2,F2-1)`
(E2–E1+1)(F2–F1)+1	`A(E1,F2)` (starting column F2–F1+1)
(E2–E1+1)(F2–F1)+2	`A(E1+1,F2)`
.	.
.	.
.	.
(E2–E1+1)(F2–F1+1)	`A(E2,F2)`

For example, the 3 by 4 array `A` specified in the DIMENSION statement as `A(2:4,1:4)` and containing the following elements:

$$A = \begin{bmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \end{bmatrix}$$

has its elements arranged in storage as follows:

Relative Location	**Array Element Value**
1	11 (starting column 1)
2	21
3	31
4	12 (starting column 2)
5	22
6	32
7	13 (starting column 3)
8	23
9	33
10	14 (starting column 4)
11	24

12 34

Each element A(I,J) of the array A, declared A(1:*n*, 1:*m*), containing real or complex data, occupies the storage location whose address is given by the following formula:

address {A(I,J)} = address {A} + (I−1 + *n*(J−1))*f*

for:

I = 1, *n* and
J = 1, *m*

where:

f = 4 for short-precision real numbers
f = 8 for long-precision real numbers
f = 8 for short-precision complex numbers
f = 16 for long-precision complex numbers

Three-Dimensional Array

For a three-dimensional array in Fortran 77, you can code:

```
DIMENSION  A(E1:E2,F1:F2,G1:G2)
```

where A is the name of the array. E1, F1, and G1 are the lower bounds of the first, second, and third dimensions, respectively, and E2, F2, and G2 are the upper bounds of the first, second, and third dimensions, respectively. If any of the lower bounds are not specified, such as in A(E1:E2,F1:F2,G2), the value is assumed to be 1. The upper bounds are always required for each dimension. For examples of Fortran 77 usage, see "SCFT3 and DCFT3—Complex Fourier Transform in Three Dimensions" on page 812.

The elements of a three-dimensional array can be thought of as a set of two-dimensional arrays, stored sequentially in ascending storage locations in the array. The elements in each two-dimensional array are stored as defined in the previous section. In the three-dimensional array, the value of the first (row) subscript expression increases most rapidly, the second (column) subscript expression increases less rapidly, and the third subscript expression (set of rows and columns) increases least rapidly. Following are the locations of the elements in the array:

Relative Location	**Array Element**
1	A(E1,F1,G1) (starting the first set)
2	A(E1+1,F1,G1)
.	.
.	.
.	.
(E2–E1+1)(F2–F1+1)	A(E2,F2,G1)
(E2–E1+1)(F2–F1+1)+1	A(E1,F1,G1+1) (starting the second set)
(E2–E1+1)(F2–F1+1)+2	A(E1+1,F1,G1+1)
.	.
.	.
.	.
(E2–E1+1)(F2–F1+1)(2)	A(E2,F2,G1+1)
(E2–E1+1)(F2–F1+1)(2)+1	A(E1,F1,G1+2) (starting the third set)
(E2–E1+1)(F2–F1+1)(2)+2	A(E1+1,F1+2)
.	.
.	.

.	.
(E2–E1+1)(F2–F1+1)(G2–G1)	A(E2,F2,G2–1)
(E2–E1+1)(F2–F1+1)(G2–G1)+1	A(E1,F1,G2) (starting the last set*)
(E2–E1+1)(F2–F1+1)(G2–G1)+2	A(E1+1,F1,G2)
.	.
.	.
.	.
(E2–E1+1)(F2–F1+1)(G2–G1+1)	A(E2,F2,G2)

* The last set is the G2–G1+1 set.

For example, the 3 by 2 by 4 array A specified in the DIMENSION statement as A(1:3,0:1,2:5) and containing the following sets of rows and columns of elements:

```
        ┌         ┐   ┌         ┐  ┌         ┐  ┌         ┐
        │ 111 121 │   │ 112 122 │  │ 113 123 │  │ 114 124 │
A =     │ 211 221 │   │ 212 222 │  │ 213 223 │  │ 214 224 │
        │ 311 321 │   │ 312 322 │  │ 313 323 │  │ 314 324 │
        └         ┘   └         ┘  └         ┘  └         ┘
```

has its elements arranged in storage as follows:

Relative Location	Array Element Value
1	111 (starting the first set)
2	211
3	311
4	121
5	221
6	321
7	112 (starting the second set)
8	212
9	312
10	122
11	222
12	322
13	113 (starting the third set)
14	213
15	313
16	123
17	223
18	323
19	114 (starting the fourth set)
20	214
21	314
22	124
23	224
24	324

Each element A(I,J,K) of the array A, declared A(1:*n*, 1:*m*, 1:*p*), containing real or complex data, occupies the storage location whose address is given by the following formula:

address {A(I,J,K)} = address {A} + (I–1 + *n*(J–1) + *mn*(K–1))*f*

for:

I = 1, n
J = 1, m
K = 1, p

where:

f = 4 for short-precision real numbers
f = 8 for long-precision real numbers
f = 8 for short-precision complex numbers
f = 16 for long-precision complex numbers

Creating Multiple Threads and Calling ESSL from Your Fortran Program

The example shown below shows how to create up to a maximum of eight threads, where each thread calls the DURAND and DGEICD subroutines.

Be sure to compile this program with the `xlf_r` command and the `-qnosave` option.

```
       program matinv_example
       implicit none
!
!  program to invert m nxn random matrices
!
       real(8), allocatable :: A(:,:,:), det(:,:), rcond(:)
       real(8)              :: dummy_aux, seed=1998, sd
       integer              :: rc, i, m=8, n=500, iopt=3, naux=0
!
! allocate storage
!
       allocate(A(n,n,m),stat=rc)
       call error_exit(rc,"Allocation of matrix A")
       allocate(det(2,m),stat=rc)
       call error_exit(rc,"Allocation of det")
       allocate(rcond(m),stat=rc)
       call error_exit(rc,"Allocation of rcond")

!
! Calculate inverses in parallel
!
!SMP$  parallel do private(i,sd), schedule(static),
!SMP$&    share(n,a,iopt,rcond,det,dummy_aux,naux)
       do i=1,m
```

```
        sd = seed + 100*i
        call durand(sd,n*n,A(1,1,i))
        call dgeicd(A(1,1,i),n,n,iopt,rcond(i),det(1,i),
     &              dummy_aux,naux)
      enddo


      write(*,*)'Reciprocal condition numbers of the matrices are:'
      write(*,'(4E12.4)') rcond
!
!
!
      deallocate(A,stat=rc)

      call error_exit(rc,"Deallocation of matrix A")
      deallocate(det,stat=rc)
      call error_exit(rc,"Deallocation of det")
      deallocate(rcond,stat=rc)
      call error_exit(rc,"Deallocation of rcond")
      stop

      contains
        subroutine error_exit(error_code,string)
        character(*)    :: string
        integer         :: error_code
        if(error_code .eq. 0 ) return
        write(0,*)string,": failing return code was ",error_code
        stop 1
        end subroutine error_exit
      end
```

Handling Errors in Your Fortran Program

ESSL provides you with flexibilities in handling both input-argument errors and computational errors:

- For input-argument errors 2015 and 2030, which are optionally-recoverable errors, ESSL allows you to obtain corrected input-argument values and react at run time.

  Note: In the case where error 2015 is unrecoverable, you have the option of dynamic allocation for most of the *aux* arguments. For details see the subroutine descriptions in Part 2 of this book.

- For computational errors, ESSL provides a return code and additional information to help you analyze the problem in your program and react at run time.

"Input-Argument Errors in Fortran" on page 119 and "Computational Errors in Fortran" on page 122 explain how to use these facilities by describing the additional statements you must code in your program.

For multithreaded application programs, if you want to initialize the error option table and change the default settings for input-argument and computational errors, you need to implement the steps shown in "Input-Argument Errors in Fortran" on page 119 and "Computational Errors in Fortran" on page 122 on each thread that calls ESSL. An example is shown in "Example of Handling Errors in a Multithreaded Application Program" on page 127.

Input-Argument Errors in Fortran

To obtain corrected input-argument values in a Fortran program and to avert program termination for the optionally-recoverable input-argument errors 2015 and 2030, add the statements in the following steps your program. Steps 3 and 7 for ERRSAV and ERRSTR, respectively, are optional. Adding these steps makes the effect of the call to ERRSET temporary.

Step 1. Declare ENOTRM as External

```
EXTERNAL ENOTRM
```

This declares the ESSL error exit routine ENOTRM as an external reference in your program. This should be coded in the beginning of your program before any of the following statements.

Step 2. Call EINFO for Initialization

```
CALL EINFO (0)
```

This calls the EINFO subroutine with one argument of value 0 to initialize the ESSL error option table. It is required only if you call ERRSET in your program. It is coded only once in the beginning of your program before any calls to ERRSET. For a description of EINFO, see "EINFO—ESSL Error Information-Handler Subroutine" on page 968.

Step 3. Call ERRSAV

```
CALL  ERRSAV (ierno,tabent)
```

(This is an optional step.) This calls the ERRSAV subroutine, which stores the error option table entry for error number *ierno* in an 8-byte storage area, *tabent*, which is accessible to your program. ERRSAV must be called for each entry you want to save. This step is used, along with step 7, for ERRSTR. For information on whether you should use ERRSAV and ERRSTR, see " How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?" on page 53. For an example, see "Example 3" on page 126, as the use is the same as for computational errors.

Step 4. Call ERRSET

```
CALL  ERRSET (ierno,inoal,inomes,itrace,iusadr,irange)
```

This calls the ERRSET subroutine, which allows you to dynamically modify the action taken when an error occurs. For optionally-recoverable ESSL input-argument errors, you need to call ERRSET only if you want to avoid terminating your program and you want the input arguments associated with this error to be assigned correct values in your

program when the error occurs. For one error (*ierno*) or a range of errors (*irange*), you can specify:

- How many times each error can occur before execution terminates (*inoal*)
- How many times each error message can be printed (*inomes*)
- The ESSL exit routine ENOTRM, to be invoked for the error indicated (*iusadr*)

ERRSET must be called for each error code you want to indicate as being recoverable. For ESSL, *ierno* should have a value of 2015 or 2030. If you want to eliminate error messages, you should indicate a negative number for *inomes*; otherwise, you should specify 0 for this argument. All the other ERRSET arguments should be specified as 0.

For a list of the default values set in the ESSL error option table, see Table 26 on page 51. For a description of the input-argument errors, see "Input-Argument Error Messages" on page 179. For a description of ERRSET, see Chapter 17 on page 965.

Step 5. Call ESSL

```
CALL  name  (arg-1,...,arg-n,*yyy,*zzz,...)
```

This calls the ESSL subroutine and specifies a branch on one or more return code values, where:

- *name* specifies the ESSL subroutine.
- *arg-1,..., arg-n* are the input and output arguments.
- *yyy*, *zzz*, and any other statement numbers preceded by an "*" are the Fortran statement numbers indicating where you want to branch when you get a nonzero return code. Each corresponds to a different ESSL value. Control goes to the corresponding statement number when a nonzero return code value is returned for the CALL statement. Return code values are described under "Error Conditions" in each ESSL subroutine description in Part 2 of this book.

Step 6. Perform the Desired Action: These are the statements at statement number *yyy* or *zzz*, shown in the CALL statement in Step 5, and preceded by an "*." The statement to which control is passed corresponds to the return code value for the error.

These statements perform whatever action is desired when the recoverable error occurs. These statements may check the new values set in the input arguments to determine whether adequate program storage is available, and then decide whether to continue or terminate the program. Otherwise, these statements may check that the size of the working storage arrays or the length of the transform agrees with other data in the program. The program may also store this corrected input argument value for future reference.

Step 7. Call ERRSTR

```
CALL  ERRSTR (ierno,tabent)
```

(This is an optional step.) This calls the ERRSTR subroutine, which stores an entry in the error option table for error number *ierno* from an 8-byte storage area, *tabent*, which is accessible to your program. ERRSTR must be called for each entry you want to store. This step is used, along with step 3, for ERRSAV. For information on whether you should use ERRSAV and ERRSTR, see " How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?" on page 53. For an example, see "Example 3" on page 126, as the use is the same as for computational errors.

Example

This example shows an error code 2015, which resets the size of the work area *aux*, specified in *naux*, if the value specified is too small. It also indicates that no error messages should be issued.

```
       .
       .
       .
C                   DECLARE ENOTRM AS EXTERNAL
       EXTERNAL ENOTRM
       .
       .
       .
C                   INITIALIZE THE ESSL ERROR
C                   OPTION TABLE
       CALL EINFO(0)
       .
       .
       .
C                   MAKE ERROR CODE 2015 A RECOVERABLE
C                   ERROR AND SUPPRESS PRINTING ALL
C                   ERROR MESSAGES FOR IT
       CALL ERRSET(2015,0,-1,0,ENOTRM,2015)
       .
       .
       .
C                   CALL ESSL ROUTINE SWLEV.
C                   IF THE NAUX INPUT
C                   ARGUMENT IS TOO SMALL, ERROR
C                   2015 OCCURS.  THE MINIMUM VALUE
C                   REQUIRED IS STORED IN THE NAUX
C                   INPUT ARGUMENT AND CONTROL GOES
C                   TO LABEL 400.
       CALL SWLEV(X,INCX,U,INCU,Y,INCY,N,AUX,NAUX,*400)
       .
       .
       .
```

```
C                    CHECK THE RESULTING INPUT ARGUMENT
C                    VALUE  IN NAUX AND TAKE THE
C                    DESIRED ACTION
400    .
       .
       .
```

Computational Errors in Fortran

To obtain information about an ESSL computational error in a Fortran program, add the statements in the following steps to your program. Steps 2 and 7 for ERRSAV and ERRSTR, respectively, are optional. Adding these steps makes the effect of the call to ERRSET temporary. For a list of those computational errors that return information and to which these steps apply, see "EINFO—ESSL Error Information-Handler Subroutine" on page 968.

Step 1. Call EINFO for Initialization

```
CALL EINFO (0)
```

This calls the EINFO subroutine with one argument of value 0 to initialize the ESSL error option table. It is required only if you call ERRSET in your program. It is coded only once in the beginning of your program before any calls to ERRSET. For a description of EINFO, see "EINFO—ESSL Error Information-Handler Subroutine" on page 968.

Step 2. Call ERRSAV

```
CALL  ERRSAV (ierno,tabent)
```

(This is an optional step.) This calls the ERRSAV subroutine, which stores the error option table entry for error number *ierno* in an 8-byte storage area, *tabent*, which is accessible to your program. ERRSAV must be called for each entry you want to save. This step is used, along with step 7, for ERRSTR. For information on whether you should use ERRSAV and ERRSTR, see " How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?" on page 53.

Step 3. Call ERRSET

```
CALL  ERRSET (ierno,inoal,inomes,itrace,iusadr,irange)
```

This calls the ERRSET subroutine, which allows you to dynamically modify the action taken when an error occurs. For ESSL computational errors, you need to call ERRSET only if you want to change the default values in the ESSL error option table. For one error (*ierno*) or a range of errors (*irange*), you can specify:

- How many times each error can occur before execution terminates (*inoal*)
- How many times each error message can be printed (*inomes*)

ERRSET must be called for each error code you want to change the default values for. For ESSL, *ierno* should be set to one of the eligible values listed in Table 167 on page 968. To allow your program to continue after an error in the specified range occurs, *inoal* must be set to a value greater than 1. For ESSL, *iusadr* should be specified as either 0 or 1, so a user exit is not taken.

For a list of the default values set in the ESSL error option table, see Table 26 on page 51. For a description of the computational errors, see "Computational Error Messages" on page 187. For a description of ERRSET, see Chapter 17 on page 965.

Step 4. Call ESSL

```
CALL  name  (arg-1,...,arg-n,*yyy,*zzz,...)
```

This calls the ESSL subroutine and specifies a branch on one or more return code values, where:

- *name* specifies the ESSL subroutine.
- *arg-1*,..., *arg-n* are the input and output arguments.
- *yyy*, *zzz*, and any other statement numbers preceded by an "*" are the Fortran statement numbers indicating where you want to branch when you get a nonzero return code. Each corresponds to a different ESSL value. Control goes to the corresponding statement number when a nonzero return code value is returned for the CALL statement. Return code values are described under "Error Conditions" in each ESSL subroutine description in Part 2 of this book.

Step 5. Call EINFO for Information

```
nmbr  CALL EINFO (icode,inf1)
-or-
nmbr  CALL EINFO (icode,inf1,inf2)
```

This calls the EINFO subroutine, which returns information about certain computational errors, where:

- *nmbr* is the statement number *yyy*, *zzz*, or any of the other statement numbers preceded by an "*" in the CALL statement in Step 4, corresponding to the return code value for this error code.
- *icode* is the error code of interest.
- *inf1* and *inf2* are the integer variables used to receive the information, where *inf1* is assigned a value for all errors, and *inf2* is assigned a value for some errors. For a description of EINFO, see "EINFO—ESSL Error Information-Handler Subroutine" on page 968.

Step 6. Check the Values in the Information Receivers: These statements check the values returned in the output argument information receivers, *inf1* and *inf2*, which contain the information about the computational error.

Step 7. Call ERRSTR

```
CALL  ERRSTR (ierno,tabent)
```

(This is an optional step.) This calls the ERRSTR subroutine, which stores an entry in the error option table for error number *ierno* from an 8-byte storage area, *tabent*, which is accessible to your program. ERRSTR must be called for each entry you want to store. This step is used, along with step 2, for ERRSAV. For information on whether you should use ERRSAV and ERRSTR, see " How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?" on page 53.

Example 1

This example shows an error code 2104, which returns one piece of information: the index of the last diagonal with nonpositive value (I1).

```
       .
       .
       .
C                   INITIALIZE THE ESSL ERROR
C                   OPTION TABLE
       CALL EINFO(0)
       .
       .
       .
C                   ALLOW 100 ERRORS FOR CODE 2104
       CALL ERRSET(2104,100,0,0,0,2104)
       .
       .
       .
C                   CALL ESSL ROUTINE DPPF.
C                   IF THE INPUT MATRIX IS NOT
C                   POSITIVE DEFINITE, CONTROL GOES TO
C                   LABEL 400
       IOPT=0
       CALL DPPF(APP,N,IOPT,*400)
       .
       .
       .
```

```
C                   CALL THE INFORMATION-HANDLER
C                   ROUTINE FOR ERROR CODE 2104 TO
C                   RETURN ONE PIECE OF INFORMATION
C                   IN VARIABLE I1, THE INDEX OF THE
C                   LAST NONPOSITIVE DIAGONAL FOUND
C                   BY ROUTINE DPPF
400    CALL EINFO (2104,I1)
       .
       .
       .
```

Example 2

This example shows an error code 2103, which returns one piece of information: the index of the zero diagonal (I1) found by DGEF.

```
       .
       .
       .
C                   INITIALIZE THE ESSL ERROR
C                   OPTION TABLE
       CALL EINFO(0)
       .
       .
       .
C                   ALLOW 100 ERRORS FOR CODE 2103
       CALL ERRSET(2103,100,0,0,0,2103)
       .
       .
       .
C                   CALL ESSL SUBROUTINE DGEF.
C                   IF THE INPUT MATRIX IS
C                   SINGULAR, CONTROL GOES TO
C                   LABEL 400
       CALL DGEF(A,LDA,N,IPVT,*400)
       .
       .
       .
```

```
C                    CALL THE INFORMATION-HANDLER
C                    ROUTINE FOR ERROR CODE 2103 TO
C                    RETURN ONE PIECE OF INFORMATION
C                    IN VARIABLE I1, THE INDEX OF THE
C                    LAST ZERO DIAGONAL FOUND BY
C                    SUBROUTINE DGEF
400    CALL EINFO (2103,I1)
       .
       .
       .
```

Example 3

This example shows an error code 2101, which returns two pieces of information: the eigenvalue (I1) that failed to converge after the indicated (I2) number of iterations. It uses ERRSAV and ERRSTR to insulate the effects of the error handling for error 2101 by this program.

```
       .
       .
C                    DECLARE AN AREA TO SAVE THE
C                    ERROR OPTION TABLE INFORMATION
C                    FOR ERROR CODE 2101
       CHARACTER*8 SAV2101
       .
       .
C                    INITIALIZE THE ESSL ERROR
C                    OPTION TABLE
       CALL EINFO(0)
C                    SAVE THE EXISTING ERROR OPTION
C                    TABLE ENTRY FOR ERROR CODE 2101
       CALL ERRSAV(2101,SAV2101)
       .
       .
C                    ALLOW 255 ERRORS FOR CODE 2101
       CALL ERRSET(2101,255,0,0,0,2101)
       .
       .
```

```
C                   CALL ESSL SUBROUTINE DGEEV.
C                   IF THE EIGENVALUE FAILED TO
C                   CONVERGE, CONTROL GOES TO LABEL 400
      CALL DGEEV(IOPT,A,LDA,W,Z,LDZ,SELECT,N,AUX,NAUX,*400)
      .
      .

C                   CALL THE INFORMATION-HANDLER
C                   ROUTINE FOR ERROR CODE 2101 TO
C                   RETURN TWO PIECES OF INFORMATION.
C                   VARIABLE I1 CONTAINS THE EIGENVALUE
C                   THAT FAILED TO CONVERGE.  VARIABLE
C                   I2 CONTAINS THE NUMBER OF ITERATIONS.
400   CALL EINFO (2101,I1,I2)
      .
      .

C                   RESTORE THE PREVIOUS ERROR OPTION
C                   TABLE ENTRY FOR ERROR CODE 2101.
C                   ERROR PROCESSING RETURNS TO HOW IT
C                   WAS BEFORE IT WAS ALTERED BY THE ABOVE
C                   ERRSET STATEMENT.
      CALL ERRSTR(2101,SAV2101)
      .
      .
```

Example of Handling Errors in a Multithreaded Application Program

This example shows how to modify the MATINV_EXAMPLE program in " Creating Multiple Threads and Calling ESSL from Your Fortran Program" on page 117 with calls to the ESSL error handling subroutines. The ESSL error handling subroutines are called from each thread to: initialize the error option table, save the current error option table values for input-argument error 2015 and computational error 2105, change the default values for errors 2015 and 2105, and then restore the original default values for errors 2015 and 2105.

```
       program matinv_example
       implicit none
!
!  program to invert m nxn random matrices
!
       real(8), allocatable :: A(:,:,:), det(:,:), rcond(:)
       real(8)              :: dummy_aux, seed=1998, sd
       integer              :: rc, i, m=8, n=500, iopt=3, naux=0
       integer              :: inf1(8)
       character(8)         :: sav2015(8)
       character(8)         :: sav2105(8)
       integer              :: ENOTRM

!
       external ENOTRM
!
! allocate storage
!
       allocate(A(n,n,m),stat=rc)
       call error_exit(rc,"Allocation of matrix A")
       allocate(det(2,m),stat=rc)
       call error_exit(rc,"Allocation of det")
       allocate(rcond(m),stat=rc)

       call error_exit(rc,"Allocation of rcond")
!
! Calculate inverses in parallel
!
!SMP$  parallel do private(i,sd), schedule(static),
!SMP$&    share(n,m,a,iopt,rcond,det,dummy_aux,naux,sav2015,sav2105,inf1)
       do i=1,m
!
!       initialize error handling
         call einfo(0)

!
!       Save existing option table values for error 2015
         call errsav(2015,sav2015(i))
!
!       Set Error 2015 to be non-recoverable so dgeicd will dynamically
!       allocate the work area.
         call errset(2015,100,100,0,1,2015)
!
!       Save existing option table values for error 2105
         call errsav(2105,sav2105(i))
```

```
!
!       Set Error 2105 to be recoverable
         call errset(2105,100,100,0,ENOTRM,2105)
!
         sd = seed + 100*i
         call durand(sd,n*n,A(1,1,i))
         call dgeicd(A(1,1,i),n,n,iopt,rcond(i),det(1,i),
     &               dummy_aux,naux,*10,*20)
10       goto 30
!
!       Catch singular matrix returned by dgeicd.
20       CALL EINFO(2105,inf1(i))
         WRITE(*,*) 'ERROR: Zero pivot found at location ',inf1(i)
!
!       Restore the error option table entries
30       continue
         call errstr(2015,SAV2015(i))
         call errstr(2105,SAV2105(i))

       enddo

       write(*,*)'Reciprocal condition numbers of the matrices are:'
       write(*,'(4E12.4)') rcond
!
!
!
       deallocate(A,stat=rc)
       call error_exit(rc,"Deallocation of matrix A")
       deallocate(det,stat=rc)
       call error_exit(rc,"Deallocation of det")

       deallocate(rcond,stat=rc)
       call error_exit(rc,"Deallocation of rcond")
       stop
       contains
         subroutine error_exit(error_code,string)
         character(*)    :: string
         integer         :: error_code
         if(error_code .eq. 0 ) return
         write(0,*)string,": failing return code was ",error_code
         stop 1
         end subroutine error_exit
       end
```

C Programs

This section describes how to code your C program.

Calling ESSL Subroutines and Functions in C

This section shows how to call ESSL subroutines and functions from your C program.

Before You Call ESSL

Before you can call the ESSL subroutines from your C program, you must have the appropriate ESSL header file installed on your system. The ESSL header file allows you to code your function calls as described in this section. It contains entries for all the ESSL subroutines. The ESSL header file is distributed with the ESSL package. The ESSL header file to be used with the C compiler is named `essl.h`. You should check with your system support group to verify that the appropriate ESSL header file is installed.

In the beginning of your program, before you call any of the ESSL subroutines, **you must code the following statement for the ESSL header file:**

```
#include <essl.h>
```

If you are planning to create your own threads for the ESSL Thread-Safety or SMP Library, you must include the `pthread.h` header file as the first include file in your C program. For an example, see " Creating Multiple Threads and Calling ESSL from Your C Program" on page 134.

Coding the Calling Sequences

In C programs, the ESSL subroutines, not returning a function value, are invoked with the following type of statement:

```
subroutine-name (argument-1, . . . , argument-n);
```

An example of a calling sequence for SAXPY might be:

```
saxpy (5,a,x,incx,y,1);
```

The ESSL subroutines returning a function value are invoked with the following type of statement:

```
function-value-name=subroutine-name (argument-1, . . . , argument-n);
```

An example of invoking DASUM might be:

```
sum = dasum (n,x,incx);
```

See the following C publications for details about how to code the function calls:

- *IBM C Set ++ for AIX/6000 C Language Reference*
- *IBM C Set ++ for AIX/6000 User's Guide*

Passing Arguments in C

This section describes how to pass arguments in your C program.

About the Syntax Shown in This Book

The argument syntax shown in this book assumes that you have installed and are using the ESSL header file. For further details, see "Calling ESSL Subroutines and Functions in C" on page 130.

No Optional Arguments

In the ESSL calling sequences for C, there are no optional arguments, as for some programming languages. You must code all the arguments listed in the syntax.

Arguments That Must Be Passed by Value

All scalar arguments that are not modified must be passed by value in the ESSL calling sequence. (This refers to input-only scalar arguments, such as *incx*, *m*, and *lda*.)

Arguments That Must Be Passed by Reference

Following are the instances in which you pass your arguments by reference (as a pointer) in the ESSL calling sequence:

Arrays: Arguments that are arrays are passed by reference, as usual.

Subroutine Names: Some ESSL subroutines call a user-supplied subroutine. The name is part of the ESSL calling sequence. It must be passed by reference.

Output Scalar Arguments: When an output argument is a scalar data item, it must be passed by reference. This is true for all scalar data types: real, complex, and so forth. **When this occurs, it is listed in the notes of each subroutine description in Part 2 of this book.**

Character Arguments: Character arguments must be passed as strings, by reference. You specify the character, in upper- or lowercase, in the ESSL calling sequence with double quotation marks around it, as in "t". Following is an example of how you can call SGEADD, specifying the *transa* and *transb* arguments as strings *n* and *t*, respectively:

```
sgeadd (a,5,"n",b,3,"t",c,4,4,3);
```

Altered Arguments When Using Error Handling: If you use ESSL error handling in your C program, as described in "Handling Errors in Your C Program" on page 136, you must pass by reference all the arguments that can potentially be altered by ESSL error handling. This applies to all your ESSL call statements after the point where you code the #define statement, shown in step 1 in "Input-Argument Errors in C" on page 136 and in step 1 in "Computational Errors in C" on page 141. The two types of ESSL arguments are:

- *naux* arguments for auxiliary storage
- *n* arguments for transform lengths

Setting Up a User-Supplied Subroutine for ESSL in C

Some ESSL numerical quadrature subroutines call a user-supplied subroutine, *subf*, identified in the ESSL calling sequence. If your program that calls the numerical quadrature subroutines is coded in C, there are some coding rules you must follow for the *subf* subroutine:

- You can code the *subf* subroutine using only C or Fortran.
- You must declare *subf* as an external subroutine in your application program.
- You should code the *subf* subroutine to the specifications given in " Programming Considerations for the SUBF Subroutine" on page 926. For an example of coding a *subf* subroutine in C, see "Example 1" on page 937.

Setting Up Scalar Data in C

Table 28 lists the scalar data types in C that are used for ESSL. Only those types and lengths used by ESSL are listed.

Table 28. Scalar Data Types in C Programs

Terminology Used by ESSL	**C Equivalent**
Character item[1] 'N', 'T', 'C' or 'n', 't', 'c'	char * "n", "t", "c"
Logical item .TRUE., .FALSE.	Specify it as described in "Setting Up Complex and Logical Data Types in C" on page 133.[2]
Fullword integer 12345, –12345	signed int
Short-precision real number[3] 12.345	float
Long-precision real number[3] 12.345	double
Short-precision complex number[3] (123.45, –54321.0)	Specify it as described in "Setting Up Complex and Logical Data Types in C" on page 133.[2]
Long-precision complex number[3] (123.45, –54321.0)	Specify it as described in "Setting Up Complex and Logical Data Types in C" on page 133.[2]

[1] ESSL accepts character data in either upper- or lowercase in its calling sequences.

[2] There are no equivalent data types for logical and complex data in C. These require special procedures. For details, see the referenced section.

[3] Short- and long-precision numbers look the same in this book.

Setting Up Complex and Logical Data Types in C

Complex and logical data types are not part of the C language; however, some ESSL subroutines require arguments of these data types.

Complex Data

ESSL provides identifiers, `cmplx` and `dcmplx`, for complex data types, defined in the ESSL header file, as well as two macro definitions, RE and IM, for handling the real and imaginary parts of complex numbers:

```
#ifndef _CMPLX
#ifndef _REIM
#define _REIM  1
#endif
typedef union  { struct { float __re, __im;}
                 __data;  double __align;} cmplx;
#endif
#ifndef _DCMPLX
#ifndef _REIM
#define _REIM  1
#endif
typedef union  { struct { double __re, __im;}
                 __data;  double __align;} dcmplx;
#endif
#ifdef _REIM
#define RE(x)  ((x).__data.__re)
#define IM(x)  ((x).__data.__im)
#endif
```

You must, therefore, code an include statement for the ESSL header file in the beginning of your program to use these definitions. For details, see "Calling ESSL Subroutines and Functions in C" on page 130.

Assuming you are using the ESSL header file, if you declare data items to be of type `cmplx` or `dcmplx`, you can pass them as short- and long-precision complex data to ESSL, respectively. You may want to write a CSET macro to initialize complex variables, using the RE and IM macros provided in the ESSL header file. Following is an example of how to use the CSET macro to initialize the complex variable `alpha`:

```
#include <essl.h>
#define CSET(x,a,b)  (RE(x)=a, IM(x)=b)
main()
{
cmplx alpha,t[3],s[5];
.
.
.
CSET (alpha,2.0,3.0);
caxpy (3,alpha,s,1,t,2);
.
.
.
}
```

If you choose to use your own definitions for complex data, instead of those provided in the ESSL header file, you can define _CMPLX and _DCMPLX in your program for short- and long-precision complex data, respectively, using the following #define

statements. These statements are coded with your global declares in the front of your program and must be coded before the `#include` statement for the ESSL header file.

```
#define _CMPLX
#define _DCMPLX
```

If you prefer to define your complex data at compile time, you can use the job processing procedures described in "Compiling" on page 163.

Logical Data

By coding the following simple macro definitions in your program, you can then use `TRUE` or `FALSE` in assigning values to or specifying any logical arguments passed to ESSL:

```
#define FALSE 0
#define TRUE 1
```

Setting Up Arrays in C

C arrays are arranged in storage in row-major order. This means that the last subscript expression increases most rapidly, the next-to-the-last subscript expression increases less rapidly, and so forth, with the first subscript expression increasing least rapidly. ESSL subroutines require that arrays passed as arguments be in column-major order. This is the array storage convention used by Fortran, described in "Setting Up Arrays in Fortran" on page 112. To pass an array from your C program to ESSL, to have ESSL process the data correctly, and to get a result that is in the proper form for your C program, you can do any of the following:

- Build and process the matrix, logically transposed from the outset, and transpose the results as necessary.
- Before the ESSL call, transpose the input arrays. Then, following the ESSL call, transpose any arrays updated as output.
- If there are arguments in the ESSL calling sequence indicating whether the arrays are to be processed in normal or transposed form, such as the *transa* and *transb* arguments in the _GEMM subroutines, use these arguments in combination with the matrix equivalence rules to avoid having to transpose your data in separate operations. For further detail, see "SGEMMS, DGEMMS, CGEMMS, and ZGEMMS—Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes Using Winograd's Variation of Strassen's Algorithm" on page 408.

Creating Multiple Threads and Calling ESSL from Your C Program

The example shown below shows how to create two threads, where each thread calls the ISAMAX subroutine. To use the AIX pthreads library, you must specify the `pthread.h` header file as the first include file in your program.

Be sure to compile this program with the `cc_r` command.

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <essl.h>

/* Create structure for argument list  */
typedef struct {
  int        n;
  float     *x;
  int     incx;
} arg_list;

/* Define prototype for thread routine */
void *Thread(void *v);

int main()
{
 float  sx1[9] = { 1., 2., 7., -8., -5., -10., -9., 10., 6. };
 float  sx2[8] = { 1.,12., 7., -8., -5., -10., -9., 19.};
 pthread_t first_th;
 pthread_t second_th;
 int rc;
 arg_list a_l,b_l;

 /* Creating argument list for the first thread  */
 a_l.n = 9;
 a_l.incx = 1;
 a_l.x = sx1;

 /* Creating argument list for the second thread  */
 b_l.n = 8;
 b_l.incx = 1;
 b_l.x = sx2;

 /*  Creating first thread which calls the ESSL subroutine ISAMAX  */
 rc = pthread_create(&first_th, NULL, Thread, (void *) &a_l);
 if (rc) exit(-1);

 /*  Creating second thread which calls the ESSL subroutine ISAMAX */
 rc = pthread_create(&second_th, NULL, Thread, (void *) &b_l);
 if (rc) exit(-1);

 sleep(1);
 exit(0);
}
```

```
/* Thread routine which call ESSL routine ISAMAX */
void *Thread(void *v)
{
 arg_list *al;
 float *x;
 int n,incx;
 int i;

 al = (arg_list *)(v);
 x = al->x;
 n = al->n;
 incx = al->incx;

 /* Calling the ESSL subroutine ISAMAX  */

 i = isamax(n,x,incx);
 if ( i == 8)
   printf("max for sx2 should be 8 = %d\n",i);
 else
   printf("max for sx1 should be 6 = %d\n",i);
}
```

Handling Errors in Your C Program

ESSL provides you with flexibilities in handling both input-argument errors and computational errors:

- For input-argument errors 2015 and 2030, which are optionally-recoverable errors, ESSL allows you to obtain corrected input-argument values and react at run time.

  Note: In the case where error 2015 is unrecoverable, you have the option of dynamic allocation for most of the *aux* arguments. For details see the subroutine descriptions in Part 2 of this book.

- For computational errors, ESSL provides a return code and additional information to help you analyze the problem in your program and react at run time.

"Input-Argument Errors in C" and "Computational Errors in C" on page 141 explain how to use these facilities by describing the additional statements you must code in your program.

For multithreaded application programs, if you want to initialize the error option table and change the default settings for input-argument and computational errors, you need to implement the steps shown in "Input-Argument Errors in C" and "Computational Errors in C" on page 141 on each thread that calls ESSL.

Input-Argument Errors in C

To obtain corrected input-argument values in a C program and to avert program termination for the optionally-recoverable input-argument errors 2015 and 2030, add the statements in the following steps to your program. Steps 4 and 8 for ERRSAV and ERRSTR, respectively, are optional. Adding these steps makes the effect of the call to ERRSET temporary.

Step 1. Code the Global Statements for ESSL Error Handling

```
/* Code two underscores */
/* before the letters ESVERR */
#define __ESVERR
#include <essl.h>
      extern int enotrm();
```

These statements are coded with your global declares in the front of your program. The `#define` must be coded before the `#include` statement for the ESSL header file. The `extern` statement declares the ESSL error exit routine ENOTRM as an external reference in your program. **After the point where you code these statements in your program, you must pass by reference all ESSL calling sequence arguments that can potentially be altered by ESSL error handling.** This applies to all your ESSL call statements. The two types of arguments are:

- *naux* arguments for auxiliary storage
- *n* arguments for transform lengths

Step 2. Declare the Variables

```
int (*iusadr) ();
int ierno,inoal,inomes,itrace,irange,irc,dummy;
char storarea[8];
```

This declares a pointer, *iusadr*, to be used for the ESSL error exit routine ENOTRM. Also included are declares for the variables used by the ESSL and Fortran error-handling subroutines. Note that *storarea* must be 8 characters long. These should be coded in the beginning of your program before any of the following statements.

Step 3. Do Initialization for ESSL

```
iusadr = enotrm;
einfo (0,&dummy,&dummy);
```

The first statement sets the function pointer, *iusadr*, to ENOTRM, the ESSL error exit routine. The last statement calls the EINFO subroutine to initialize the ESSL error option table, where *dummy* is a declared integer and is a placeholder. For a description of EINFO, see "EINFO—ESSL Error Information-Handler Subroutine" on page 968. These statements should be coded only once in the beginning of your program before calls to ERRSET.

Step 4. Call ERRSAV

```
errsav (&ierno,storarea);
```

(This is an optional step.) This calls the ERRSAV subroutine, which stores the error option table entry for error number *ierno* in an 8-byte storage area, *storarea*, which is accessible to your program. ERRSAV must be called for each entry you want to save. This step is used, along with step 8, for ERRSTR. For information on whether you should use ERRSAV and ERRSTR, see " How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?" on page 53. For an example, see "Example 1" on page 144, as the use is the same as for computational errors.

Step 5. Call ERRSET

```
errset (&ierno,&inoal,&inomes,&itrace,&iusadr,&irange);
```

This calls the ERRSET subroutine, which allows you to dynamically modify the action taken when an error occurs. For optionally-recoverable ESSL input-argument errors, you need to call ERRSET only if you want to avoid terminating your program and you want the input arguments associated with this error to be assigned correct values in your program when the error occurs. For one error (*ierno*) or a range of errors (*irange*), you can specify:

- How many times each error can occur before execution terminates (*inoal*)
- How many times each error message can be printed (*inomes*)
- The ESSL exit routine ENOTRM, to be invoked for the error indicated (*iusadr*)

ERRSET must be called for each error code you want to indicate as being recoverable. For ESSL, *ierno* should have a value of 2015 or 2030. If you want to eliminate error messages, you should indicate a negative number for *inomes*; otherwise, you should specify 0 for this argument. All the other ERRSET arguments should be specified as 0.

For a list of the default values set in the ESSL error option table, see Table 26 on page 51. For a description of the input-argument errors, see "Input-Argument Error Messages" on page 179. For a description of ERRSET, see Chapter 17 on page 965.

Step 6. Call ESSL

```
irc = name (arg1,...,argn);
if irc == rc1
        {
           .
           .
           .
              }
```

This calls the ESSL subroutine and specifies a branch on one or more return code values, where:

- *name* specifies the ESSL subroutine.

- *arg1*,...,*argn* are the input and output arguments. As explained in step 1, all arguments that can potentially be altered by error handling must be coded by reference.
- *irc* is the integer variable containing the return code resulting from the computation performed by the ESSL subroutine.
- *rc1*, *rc2*, and so forth are the possible return code values that can be passed back from the ESSL subroutine to C. The values can be 0, 1, 2, and so forth. Return code values are described under "Error Conditions" in each ESSL subroutine description in Part 2 of this book.

Step 7. Perform the Desired Action: These are the statements following the test for each value of the return code, returned in *irc* in step 6. These statements perform whatever action is desired when the recoverable error occurs. These statements may check the new values set in the input arguments to determine whether adequate program storage is available, and then decide whether to continue or terminate the program. Otherwise, these statements may check that the size of the working storage arrays or the length of the transform agrees with other data in the program. The program may also store this corrected input argument value for future reference.

Step 8. Call ERRSTR

```
errstr (&ierno,storarea);
```

(This is an optional step.) This calls the ERRSTR subroutine, which stores an entry in the error option table for error number *ierno* from an 8-byte storage area, *storarea*, which is accessible to your program. ERRSTR must be called for each entry you want to store. This step is used, along with step 4, for ERRSAV. For information on whether you should use ERRSAV and ERRSTR, see " How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?" on page 53. For an example, see "Example 1" on page 144, as the use is the same as for computational errors.

Example 1

This example shows an error code 2015, which resets the size of the work area *aux*, specified in *naux*, if the value specified is too small. It also indicates that no error messages should be issued.

```
.
.
.
    /*GLOBAL STATEMENTS FOR ESSL ERROR HANDLING*/
#define __ESVERR
#include <essl.h>
extern int enotrm();

.
.
.
    /*DECLARE THE VARIABLES*/
main ()
{
int (*iusadr) ();
int ierno,inoal,inomes,itrace,irc,dummy;
int naux;

.
.
.
    /*INITIALIZE THE POINTER TO THE ENOTRM ROUTINE*/
iusadr = enotrm;

.
.
.
    /*INITIALIZE THE ESSL ERROR OPTION TABLE*/
einfo (0,&dummy,&dummy);

.
.
.
    /*MAKE ERROR CODE 2015 A RECOVERABLE ERROR AND
      SUPPRESS PRINTING ALL ERROR MESSAGES FOR IT*/
ierno = 2015;
inoal = 0;
inomes = -1;
itrace = 0;
irange = 2015;
errset (&ierno,&inoal,&inomes,&itrace,&iusadr,&irange);

.
.
.
```

```
     /*CALL ESSL SUBROUTINE SWLEV.  NAUX IS PASSED BY
       REFERENCE.  IF THE NAUX INPUT IS TOO SMALL,
       ERROR 2015 OCCURS.  THE MINIMUM VALUE REQUIRED
       IS STORED IN THE NAUX INPUT ARGUMENT, AND THE
       RETURN CODE OF 1 IS SET IN IRC.*/
irc = swlev (x,incx,u,incu,y,incy,n,aux,&naux);
if irc == 1
{
.    /*CHECK THE RESULTING INPUT ARGUMENT VALUE
.      IN NAUX AND TAKE THE DESIRED ACTION*/
.
}

.
.
.
}
```

Computational Errors in C

To obtain information about an ESSL computational error in a C program, add the statements in the following steps to your program. Steps 4 and 9 for ERRSAV and ERRSTR, respectively, are optional. Adding these steps makes the effect of the call to ERRSET temporary. For a list of those computational errors that return information and to which these steps apply, see "EINFO—ESSL Error Information-Handler Subroutine" on page 968.

Step 1. Code the Global Statements for ESSL Error Handling

```
/* Code two underscores */
/* before the letters ESVERR */
#define __ESVERR
#include <essl.h>
```

These statements are coded with your global declares in the front of your program. The `#define` must be coded before the `#include` statement for the ESSL header file. **After the point where you code these statements in your program, you must pass by reference all ESSL calling sequence arguments that can potentially be altered by ESSL error handling.** This applies to all your ESSL call statements. The two types of arguments are:

- *naux* arguments for auxiliary storage
- *n* arguments for transform lengths

Step 2. Declare the Variables

```
int ierno,inoal,inomes,itrace,iusadr,irange,irc;
int inf1,inf2,dummy;
char storarea[8];
```

These statements include declares for the variables used by the ESSL and Fortran error-handling subroutines. Note that *storarea* must be 8 characters long. These should be coded in the beginning of your program before any of the following statements.

Step 3. Do Initialization for ESSL

```
einfo (0,&dummy,&dummy);
```

This statement calls the EINFO subroutine to initialize the ESSL error option table, where *dummy* is a declared integer and is a placeholder. For a description of EINFO, see "EINFO—ESSL Error Information-Handler Subroutine" on page 968. These statements should be coded only once in the beginning of your program before calls to ERRSET.

Step 4. Call ERRSAV

```
errsav (&ierno,storarea);
```

(This is an optional step.) This calls the ERRSAV subroutine, which stores the error option table entry for error number *ierno* in an 8-byte storage area, *storarea*, which is accessible to your program. ERRSAV must be called for each entry you want to save. This step is used, along with step 8, for ERRSTR. For information on whether you should use ERRSAV and ERRSTR, see " How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?" on page 53. For an example, see "Example 1" on page 144.

Step 5. Call ERRSET

```
errset (&ierno,&inoal,&inomes,&itrace,&iusadr,&irange);
```

This calls the ERRSET subroutine, which allows you to dynamically modify the action taken when an error occurs. For ESSL computational errors, you need to call ERRSET only if you want to change the default values in the ESSL error option table. For one error (*ierno*) or a range of errors (*irange*), you can specify:

- How many times each error can occur before execution terminates (*inoal*)
- How many times each error message can be printed (*inomes*)

ERRSET must be called for each error code you want to change the default values for. For ESSL, *ierno* should be set to one of the eligible values listed in Table 167 on page 968. To allow your program to continue after an error in the specified range occurs, *inoal* must be set to a value greater than 1. For ESSL, *iusadr* should be specified as either 0 or 1, so a user exit is not taken.

For a list of the default values set in the ESSL error option table, see Table 26 on page 51. For a description of the computational errors, see "Computational Error Messages" on page 187. For a description of ERRSET, see Chapter 17 on page 965.

Step 6. Call ESSL

```
irc = name (arg1,...,argn);
if irc == rc1
        {
          .
          .
          .
             }
if irc == rc2
        {
          .
          .
          .
            }
```

This calls the ESSL subroutine and specifies a branch on one or more return code values, where:

- *name* specifies the ESSL subroutine.
- *arg1*,...,*argn* are the input and output arguments. As explained in step 1, all arguments that can potentially be altered by error handling must be coded by reference.
- *irc* is the integer variable containing the return code resulting from the computation performed by the ESSL subroutine.
- *rc1*, *rc2*, and so forth are the possible return code values that can be passed back from the ESSL subroutine to C. The values can be 0, 1, 2, and so forth. Return code values are described under "Error Conditions" in each ESSL subroutine description in Part 2 of this book.

The statements following each test of the return code can perform any desired action. This includes calling EINFO for more information about the error, as described in step 7.

Step 7. Call EINFO for Information

```
einfo (ierno,&inf1,&inf2);
```

This calls the EINFO subroutine, which returns information about certain computational errors, where:

- *ierno* is the error code of interest.
- *inf1* and *inf2* are the integer variables used to receive the information, where *inf1* is assigned a value for all errors, and *inf2* is assigned a value for some errors. You must specify both arguments, as there are no optional arguments for C. Both arguments must be passed by reference, because they are output scalar arguments. For a description of EINFO, see "EINFO—ESSL Error Information-Handler Subroutine" on page 968.

Step 8. Check the Values in the Information Receivers: These statements check the values returned in the output argument information receivers, *inf1* and *inf2*, which contain the information about the computational error.

Step 9. Call ERRSTR

```
errstr (&ierno,storarea);
```

(This is an optional step.) This calls the ERRSTR subroutine, which stores an entry in the error option table for error number *ierno* from an 8-byte storage area, *storarea*, which is accessible to your program. ERRSTR must be called for each entry you want to store. This step is used, along with step 4, for ERRSAV. For information on whether you should use ERRSAV and ERRSTR, see " How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?" on page 53. For an example, see "Example 1."

Example 1

This example shows an error code 2105, which returns one piece of information: the index of the pivot element (*i*) near zero, causing factorization to fail. It uses ERRSAV and ERRSTR to insulate the effects of the error handling for error 2105 by this program.

```
      .
      .
                /*GLOBAL STATEMENTS FOR ESSL ERROR HANDLING*/
      #define __ESVERR
      #include <essl.h>
      .
      .
                /*DECLARE THE VARIABLES*/
      main ()
      {
      int ierno,inoal,inomes,itrace,iusadr,irange,irc;
      int inf1,inf2,dummy;
      char sav2105[8];
      .
      .
                /*INITIALIZE THE ESSL ERROR OPTION TABLE*/
      einfo (0,&dummy,&dummy);
                /*SAVE THE EXISTING ERROR OPTION TABLE ENTRY
                  FOR ERROR CODE 2105*/
      ierno = 2105;
      errsav (&ierno,sav2105);
      .
      .
```

```
            /*MAKE ERROR CODES 2101 THROUGH 2105 RECOVERABLE
              ERRORS AND SUPPRESS PRINTING ALL ERROR MESSAGES
              FOR THEM.  THIS SHOWS HOW YOU CODE THE
              ERRSET ARGUMENTS FOR A RANGE OF ERRORS. */
ierno = 2101;
inoal = 0;
inomes = 0; /*A DUMMY ARGUMENT*/
itrace = 0; /*A DUMMY ARGUMENT*/
iusadr = 0; /*A DUMMY ARGUMENT*/
irange = 2105
errset (&ierno,&inoal,&inomes,&itrace, &iusadr,&irange);

.
.
            /*CALL ESSL SUBROUTINE DGEICD. IF THE INPUT MATRIX
              IS SINGULAR OR NEARLY SINGULAR, ERROR 2105
              OCCURS.  A RETURN CODE OF 2 IS SET IN IRC.*/
irc = dgeicd (a,lda,n,iopt,&rcond,det,aux,&naux);
if irc == 2
   {
            /*CALL THE INFORMATION-HANDLER ROUTINE FOR ERROR
              CODE 2105 TO RETURN ONE PIECE OF INFORMATION
              IN VARIABLE INF1, THE INDEX OF THE PIVOT ELEMENT
              NEAR ZERO, CAUSING FACTORIZATION TO FAIL.
              INF2 IS NOT USED, BUT MUST BE SPECIFIED.
              BOTH INF1 AND INF2 ARE PASSED BY REFERENCE,
              BECAUSE THEY ARE OUTPUT SCALAR ARGUMENTS.*/
   ierno = 2105;
   einfo (ierno,&inf1,&inf2);
            /*CHECK THE VALUE IN VARIABLE INF1 AND TAKE THE
              DESIRED ACTION*/
   .
   .
   }

.
.
            /*RESTORE THE PREVIOUS ERROR OPTION TABLE ENTRY
              FOR ERROR CODE 2105.  ERROR PROCESSING
              RETURNS TO HOW IT WAS BEFORE IT WAS ALTERED BY
              THE ABOVE ERRSAV STATEMENT*/
ierno = 2105;
errstr (&ierno,sav2105);
.
.
}
```

C++ Programs

This section describes how to code your C++ program.

Calling ESSL Subroutines and Functions in C++

This section shows how to call ESSL subroutines and functions from your C++ program.

Before You Call ESSL

Before you can call the ESSL subroutines from your C++ program, you must have the appropriate ESSL header file installed on your system. The ESSL header file allows you to code your function calls as described in this section. It contains entries for all the ESSL subroutines. The ESSL header file is distributed with the ESSL package. The ESSL header file to be used with the C++ compiler is named `essl.h`.

In the beginning of your program, before you call any of the ESSL subroutines, **you must code the following statement for the ESSL header file:**

```
#include <essl.h>
```

If you are creating your own threads for the ESSL Thread-Safe or SMP Library, you must include the `pthread.h` header file in your C++ program. For an example, see " Creating Multiple Threads and Calling ESSL from Your C++ Program" on page 150.

Coding the Calling Sequences

In C++ programs, the ESSL subroutines, not returning a function value, are invoked with the following type of statement:

```
subroutine-name (argument-1, . . . , argument-n);
```

An example of a calling sequence for SAXPY might be:

```
saxpy (5,a,x,incx,y,1);
```

The ESSL subroutines returning a function value are invoked with the following type of statement:

```
function-value-name=subroutine-name (argument-1, . . . , argument-n);
```

An example of invoking DASUM might be:

```
sum = dasum (n,x,incx);
```

See the following C++ publications for details about how to code the function calls:

- *IBM C++ for AIX/6000 User's Guide*
- *IBM C Set ++ for AIX/6000 C++ Language Reference*
- *IBM C Set ++ for AIX/6000 Standard Class Library Reference Guide*

Passing Arguments in C++

This section describes how to pass arguments in your C++ program.

About the Syntax Shown in This Book

The argument syntax shown in this book assumes that you have installed and are using the ESSL header file. For further details, see "Calling ESSL Subroutines and Functions in C++" on page 146.

No Optional Arguments

In the ESSL calling sequences for C++, there are no optional arguments, as for some programming languages. You must code all the arguments listed in the syntax.

Arguments That Must Be Passed by Value

All scalar arguments that are not modified must be passed by value in the ESSL calling sequence. (This refers to input-only scalar arguments, such as *incx*, *m*, and *lda*.)

Arguments That Must Be Passed by Reference

Following are the instances in which you pass your arguments by reference (as a pointer) in the ESSL calling sequence:

Arrays: Arguments that are arrays are passed by reference, as usual.

Subroutine Names: Some ESSL subroutines call a user-supplied subroutine. The name is part of the ESSL calling sequence. It must be passed by reference.

Character Arguments: Character arguments must be passed as strings, by reference. You specify the character, in upper- or lowercase, in the ESSL calling sequence with double quotation marks around it, as in "t". Following is an example of how you can call SGEADD, specifying the *transa* and *transb* arguments as strings *n* and *t*, respectively:

```
sgeadd (a,5,"n",b,3,"t",c,4,4,3);
```

Setting Up a User-Supplied Subroutine for ESSL in C++

Some ESSL numerical quadrature subroutines call a user-supplied subroutine, *subf*, identified in the ESSL calling sequence. If your program that calls the numerical quadrature subroutines is coded in C++, there are some coding rules you must follow for the *subf* subroutine:

- You can code the *subf* subroutine using only C, C++, or Fortran.
- You must declare *subf* as an external subroutine in your application program.
- You should code the *subf* subroutine to the specifications given in " Programming Considerations for the SUBF Subroutine" on page 926. For an example of coding a *subf* subroutine in C++, see "Example 1" on page 937.

Setting Up Scalar Data in C++

Table 29 on page 148 lists the scalar data types in C++ that are used for ESSL. Only those types and lengths used by ESSL are listed.

Table 29. Scalar Data Types in C++ Programs

Terminology Used by ESSL	C++ Equivalent
Character item[1] 'N', 'T', 'C' or 'n', 't', 'c'	char * “n”, “t”, “c”
Logical item .TRUE., .FALSE.	Specify it as described in “Setting Up Short-Precision Complex Data Types and Logical Data Types in C++” on page 148.[2]
Fullword integer 12345, –12345	signed int
Short-precision real number[3] 12.345	float
Long-precision real number[3] 12.345	double
Short-precision complex number[3] (123.45, –54321.0)	Specify it as described in “Setting Up Short-Precision Complex Data Types and Logical Data Types in C++” on page 148.[2]
Long-precision complex number[3] (123.45, –54321.0)	complex[4]

[1] ESSL accepts character data in either upper- or lowercase in its calling sequences.

[2] There are no equivalent data types for logical and short-precision complex data in C++. These require special procedures. For details, see the referenced section.

[3] Short- and long-precision numbers look the same in this book.

[4] This data type is defined in file <complex.h> for C++.

Setting Up Short-Precision Complex Data Types and Logical Data Types in C++

Short-precision complex data types and logical data types are not part of the C++ language; however, some ESSL subroutines require arguments of these data types.

Short-Precision Complex Data

ESSL provides an identifier, `cmplx`, for the short-precision complex data type, defined in the ESSL header file, as well as two member functions, `sreal` and `simag`, for handling the real and imaginary parts of short-precision complex numbers:

```
#ifndef _CMPLX
class cmplx
  {
  private:
    union { struct { float _re,_im; } _data; double _esvalign;};
  public:
    cmplx() { _data._re = 0.0; _data._im = 0.0; }
    cmplx(float r, float i = 0.0) { _data._re = r; _data._im = i; }
    cmplx(cmplx &c) { _data._re = c._data._re;
                              _data._im = c._data._im; }
    friend inline float sreal(const cmplx& a) { return a._data._re; }
    friend inline float simag(const cmplx& a) { return a._data._im; }
  };
#endif
```

You must, therefore, code an include statement for the ESSL header file in the beginning of your program to use these definitions. For details, see "Calling ESSL Subroutines and Functions in C++" on page 146.

Assuming you are using the ESSL header file, if you declare data items to be of type `cmplx` or `complex`, you can pass them as short- or long-precision complex data to ESSL, respectively. Following is an example of how you might code your program:

```
#include <complex.h>
#include <essl.h>
main()
{
cmplx alpha,t[3],s[5];
complex beta,td[3],sd[5];
.
.
.
alpha = cmplx(2.0,3.0);
caxpy (3,alpha,s,1,t,2);
.
.
.
beta = complex(2.0,3.0);
zaxpy (3,beta,sd,1,td,2);
.
.
.
}
```

If you choose to use your own definition for short-precision complex data, instead of that provided in the ESSL header file, you can define _CMPLX in your program, using the following `#define` statement. This statement is coded with your global declares in the front of your program and must be coded before the `#include` statement for the ESSL header file.

```
#define _CMPLX
```

If you prefer to define your short-precision complex data at compile time, you can use the job processing procedures described in "Compiling" on page 163.

Logical Data

By coding the following simple macro definitions in your program, you can then use TRUE or FALSE in assigning values to or specifying any logical arguments passed to ESSL:

```
#define FALSE 0
#define TRUE 1
```

Setting Up Arrays in C++

C++ arrays are arranged in storage in row-major order. This means that the last subscript expression increases most rapidly, the next-to-the-last subscript expression increases less rapidly, and so forth, with the first subscript expression increasing least rapidly. ESSL subroutines require that arrays passed as arguments be in column-major order. This is the array storage convention used by Fortran, described in "Setting Up Arrays in Fortran" on page 112. To pass an array from your C++ program to ESSL, to have ESSL process the data correctly, and to get a result that is in the proper form for your C++ program, you can do any of the following:

- Build and process the matrix, logically transposed from the outset, and transpose the results as necessary.
- Before the ESSL call, transpose the input arrays. Then, following the ESSL call, transpose any arrays updated as output.
- If there are arguments in the ESSL calling sequence indicating whether the arrays are to be processed in normal or transposed form, such as the *transa* and *transb* arguments in the _GEMM subroutines, use these arguments in combination with the matrix equivalence rules to avoid having to transpose your data in separate operations. For further detail, see "SGEMMS, DGEMMS, CGEMMS, and ZGEMMS—Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes Using Winograd's Variation of Strassen's Algorithm" on page 408.

Creating Multiple Threads and Calling ESSL from Your C++ Program

The example shown below shows how to create two threads, where each thread calls the ISAMAX subroutine. To use the AIX pthreads library, you must remember to code the pthread.h header file in your C++ program.

Be sure to compile this program with the xlC_r command.

```
#include "essl.h"
#include <iostream.h>

/* Define prototype for thread routine */
void *Thread(void *v);
```

```
/* Define prototype for thread library routine, which is in C */
extern "C" {
#include <pthread.h>
#include <stdlib.h>
int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
}


extern "Fortran" int isamax(const int &,  float *, const int &);

/* Create structure for argument list  */
struct arg_list {
  int      n;
  float   *x;
  int   incx;
};


void main()
{
 float  sx1[9] = { 1., 2., 7., -8., -5., -10., -9., 10., 6. };
 float  sx2[8] = { 1.,12., 7., -8., -5., -10., -9., 19.};
 pthread_t first_th;
 pthread_t second_th;
 int rc;
 struct arg_list a_l,b_l;


 a_l.n = 9;
 a_l.incx = 1;
 a_l.x = sx1;

 b_l.n = 8;
 b_l.incx = 1;
 b_l.x = sx2;


 /* Creating argument list for first thread  */
 rc = pthread_create(&first_th, NULL, Thread, (void *) &a_l);
 if (rc) exit(-1);

 /* Creating argument list for second thread  */
 rc = pthread_create(&second_th, NULL, Thread, (void *) &b_l);
 if (rc) exit(-1);


 sleep(20);
 exit(0);
}
```

```
/* Thread routine which calls the ESSL subroutine ISAMAX */
void* Thread(void *v)
{
 struct arg_list *al;
 float *t;
 int n,incx;
 int i;


 al = (struct arg_list *)(v);
 t = al->x;
 n = al->n;
 incx = al->incx;


 /* Calling the ESSL subroutine ISAMAX  */

 i = isamax(n,t,incx);
 if ( i == 8)
   cout << "max for sx2 should be 8 = " << i << "\n";
 else
   cout << "max for sx1 should be 6 = " << i << "\n";
 return NULL;
}
```

Handling Errors in Your C++ Program

ESSL provides you with flexibilities in handling both input-argument errors and computational errors:

- For input-argument errors 2015 and 2030, which are optionally-recoverable errors, ESSL allows you to obtain corrected input-argument values and react at run time.

  Note: In the case where error 2015 is unrecoverable, you have the option of dynamic allocation for most of the *aux* arguments. For details see the subroutine descriptions in Part 2 of this book.

- For computational errors, ESSL provides a return code and additional information to help you analyze the problem in your program and react at run time.

"Input-Argument Errors in C++" on page 153 and "Computational Errors in C++" on page 157 explain how to use these facilities by describing the additional statements you must code in your program.

For multithreaded application programs, if you want to initialize the error option table and change the default settings for input-argument and computational errors, you need to implement the steps shown in "Input-Argument Errors in C++" on page 153 and "Computational Errors in C++" on page 157 on each thread that calls ESSL.

Input-Argument Errors in C++

To obtain corrected input-argument values in a C++ program and to avert program termination for the optionally-recoverable input-argument errors 2015 and 2030, add the statements in the following steps to your program. Steps 4 and 8 for ERRSAV and ERRSTR, respectively, are optional. Adding these steps makes the effect of the call to ERRSET temporary.

Step 1. Code the Global Statements for ESSL Error Handling

```
/* Code one underscore */
/* before the letters ESVERR */
#define _ESVERR
#include <iostream.h>
#include <stdio.h>
#include <essl.h>
extern "Fortran" int enotrm(int &,int &);
```

These statements are coded with your global declares in the front of your program. The `#define` must be coded before the `#include` statements for the ESSL header file. The `extern` statement declares the ESSL error exit routine ENOTRM as an external reference in your program.

Step 2. Declare the Variables

```
int (*iusadr) (int &,int &);
int ierno,inoal,inomes,itrace,irange,irc,dummy;
char storarea[8];
```

This declares a pointer, *iusadr*, to be used for the ESSL error exit routine ENOTRM. Also included are declares for the variables used by the ESSL and Fortran error-handling subroutines. Note that *storarea* must be 8 characters long. These should be coded in the beginning of your program before any of the following statements.

Step 3. Do Initialization for ESSL

```
iusadr = enotrm;
dummy = 0;
einfo (0,dummy,dummy);
```

The first statement sets the function pointer, *iusadr*, to ENOTRM, the ESSL error exit routine. The last statement calls the EINFO subroutine to initialize the ESSL error option table, where *dummy* is a declared integer and is a placeholder. For a description of EINFO, see "EINFO—ESSL Error Information-Handler Subroutine" on page 968. These statements should be coded only once in the beginning of your program before calls to ERRSET.

Step 4. Call ERRSAV

```
errsav (ierno,storarea);
```

(This is an optional step.) This calls the ERRSAV subroutine, which stores the error option table entry for error number *ierno* in an 8-byte storage area, *storarea*, which is accessible to your program. ERRSAV must be called for each entry you want to save. This step is used, along with step 8, for ERRSTR. For information on whether you should use ERRSAV and ERRSTR, see " How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?" on page 53. For an example, see "Example" on page 160, as the use is the same as for computational errors.

Step 5. Call ERRSET

```
errset (ierno,inoal,inomes,itrace,iusadr,irange);
```

This calls the ERRSET subroutine, which allows you to dynamically modify the action taken when an error occurs. For optionally-recoverable ESSL input-argument errors, you need to call ERRSET only if you want to avoid terminating your program and you want the input arguments associated with this error to be assigned correct values in your program when the error occurs. For one error (*ierno*) or a range of errors (*irange*), you can specify:

- How many times each error can occur before execution terminates (*inoal*)
- How many times each error message can be printed (*inomes*)
- The ESSL exit routine ENOTRM, to be invoked for the error indicated (*iusadr*)

ERRSET must be called for each error code you want to indicate as being recoverable. For ESSL, *ierno* should have a value of 2015 or 2030. If you want to eliminate error messages, you should indicate a negative number for *inomes*; otherwise, you should specify 0 for this argument. All the other ERRSET arguments should be specified as 0.

For a list of the default values set in the ESSL error option table, see Table 26 on page 51. For a description of the input-argument errors, see "Input-Argument Error Messages" on page 179. For a description of ERRSET, see Chapter 17 on page 965.

Step 6. Call ESSL

```
irc = name (arg1,...,argn);
if irc == rc1
    {
      .
      .
      .
        }
if irc == rc2
    {
      .
      .
      .
        }
```

This calls the ESSL subroutine and specifies a branch on one or more return code values, where:

- *name* specifies the ESSL subroutine.
- *arg1*,...,*argn* are the input and output arguments.
- *irc* is the integer variable containing the return code resulting from the computation performed by the ESSL subroutine.
- *rc1*, *rc2*, and so forth are the possible return code values that can be passed back from the ESSL subroutine to C++. The values can be 0, 1, 2, and so forth. Return code values are described under "Error Conditions" in each ESSL subroutine description in Part 2 of this book.

Step 7. Perform the Desired Action: These are the statements following the test for each value of the return code, returned in *irc* in step 6. These statements perform whatever action is desired when the recoverable error occurs. These statements may check the new values set in the input arguments to determine whether adequate program storage is available, and then decide whether to continue or terminate the program. Otherwise, these statements may check that the size of the working storage arrays or the length of the transform agrees with other data in the program. The program may also store this corrected input argument value for future reference.

Step 8. Call ERRSTR

```
errstr (ierno,storarea);
```

(This is an optional step.) This calls the ERRSTR subroutine, which stores an entry in the error option table for error number *ierno* from an 8-byte storage area, *storarea*, which is accessible to your program. ERRSTR must be called for each entry you want to store. This step is used, along with step 4, for ERRSAV. For information on whether you should use ERRSAV and ERRSTR, see " How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?" on page 53. For an example, see "Example" on page 160, as the use is the same as for computational errors.

Example

This example shows an error code 2015, which resets the size of the work area *aux*, specified in *naux*, if the value specified is too small. It also indicates that no error messages should be issued.

```
.
.
.
    /*GLOBAL STATEMENTS FOR ESSL ERROR HANDLING*/
#define _ESVERR
#include <essl.h>
#include <iostream.h>
#include <stdio.h>
extern "Fortran" int enotrm(int &,int &);

.
.
.
    /*DECLARE THE VARIABLES*/
main ()
{
int (*iusadr) (int &,int &);
int ierno,inoal,inomes,itrace,irc,dummy;
int naux;

.
.
.
    /*INITIALIZE THE POINTER TO THE ENOTRM ROUTINE*/
iusadr = enotrm;

.
.
.
    /*INITIALIZE THE ESSL ERROR OPTION TABLE*/
dummy = 0;
einfo (0,dummy,dummy);

.
.
.
    /*MAKE ERROR CODE 2015 A RECOVERABLE ERROR AND
      SUPPRESS PRINTING ALL ERROR MESSAGES FOR IT*/
ierno = 2015;
inoal = 0;
inomes = -1;
itrace = 0;
irange = 2015;
errset (ierno,inoal,inomes,itrace,iusadr,irange);
```

```
.
.
.
    /*CALL ESSL SUBROUTINE SWLEV.  NAUX IS PASSED BY
      REFERENCE.  IF THE NAUX INPUT IS TOO SMALL,
      ERROR 2015 OCCURS.  THE MINIMUM VALUE REQUIRED
      IS STORED IN THE NAUX INPUT ARGUMENT, AND THE
      RETURN CODE OF 1 IS SET IN IRC.*/
irc = swlev (x,incx,u,incu,y,incy,n,aux,naux);
if irc == 1
{
.    /*CHECK THE RESULTING INPUT ARGUMENT VALUE
.      IN NAUX AND TAKE THE DESIRED ACTION*/
.
}

.
.
.
}
```

Computational Errors in C++

To obtain information about an ESSL computational error in a C++ program, add the statements in the following steps to your program. Steps 4 and 9 for ERRSAV and ERRSTR, respectively, are optional. Adding these steps makes the effect of the call to ERRSET temporary. For a list of those computational errors that return information and to which these steps apply, see "EINFO—ESSL Error Information-Handler Subroutine" on page 968.

Step 1. Code the Global Statements for ESSL Error Handling

```
/* Code one underscore */
/* before the letters ESVERR */
#define _ESVERR
#include <iostream.h>
#include <stdio.h>
#include <essl.h>
```

These statements are coded with your global declares in the front of your program. The `#define` must be coded before the `#include` statement for the ESSL header file.

Step 2. Declare the Variables

```
int ierno,inoal,inomes,itrace,iusadr,irange,irc;
int inf1,inf2,dummy;
char storarea[8];
```

These statements include declares for the variables used by the ESSL and Fortran error-handling subroutines. Note that *storarea* must be 8 characters long. These should be coded in the beginning of your program before any of the following statements.

Step 3. Do Initialization for ESSL

```
dummy = 0;
einfo (0,dummy,dummy);
```

The last statement calls the EINFO subroutine to initialize the ESSL error option table, where *dummy* is a declared integer and is a placeholder. For a description of EINFO, see "EINFO—ESSL Error Information-Handler Subroutine" on page 968. These statements should be coded only once in the beginning of your program before calls to ERRSET.

Step 4. Call ERRSAV

```
errsav (ierno,storarea);
```

(This is an optional step.) This calls the ERRSAV subroutine, which stores the error option table entry for error number *ierno* in an 8-byte storage area, *storarea*, which is accessible to your program. ERRSAV must be called for each entry you want to save. This step is used, along with step 8, for ERRSTR. For information on whether you should use ERRSAV and ERRSTR, see " How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?" on page 53. For an example, see "Example" on page 160.

Step 5. Call ERRSET

```
errset (ierno,inoal,inomes,itrace,iusadr,irange);
```

This calls the ERRSET subroutine, which allows you to dynamically modify the action taken when an error occurs. For ESSL computational errors, you need to call ERRSET only if you want to change the default values in the ESSL error option table. For one error (*ierno*) or a range of errors (*irange*), you can specify:

- How many times each error can occur before execution terminates (*inoal*)
- How many times each error message can be printed (*inomes*)

ERRSET must be called for each error code you want to change the default values for. For ESSL, *ierno* should be set to one of the eligible values listed in Table 167 on page 968. To allow your program to continue after an error in the specified range occurs, *inoal* must be set to a value greater than 1. For ESSL, *iusadr* should be specified as either 0 or 1, so a user exit is not taken.

For a list of the default values set in the ESSL error option table, see Table 26 on page 51. For a description of the computational errors, see "Computational Error Messages" on page 187. For a description of ERRSET, see Chapter 17 on page 965.

Step 6. Call ESSL

```
irc = name (arg1,...,argn);
if irc == rc1
    {
      .
      .
      .
        }
if irc == rc2
    {
      .
      .
      .
        }
```

This calls the ESSL subroutine and specifies a branch on one or more return code values, where:

- *name* specifies the ESSL subroutine.
- *arg1*,...,*argn* are the input and output arguments.
- *irc* is the integer variable containing the return code resulting from the computation performed by the ESSL subroutine.
- *rc1*, *rc2*, and so forth are the possible return code values that can be passed back from the ESSL subroutine to C++. The values can be 0, 1, 2, and so forth. Return code values are described under "Error Conditions" in each ESSL subroutine description in Part 2 of this book.

The statements following each test of the return code can perform any desired action. This includes calling EINFO for more information about the error, as described in step 7.

Step 7. Call EINFO for Information

```
einfo (ierno,inf1,inf2);
```

This calls the EINFO subroutine, which returns information about certain computational errors, where:

- *ierno* is the error code of interest.
- *inf1* and *inf2* are the integer variables used to receive the information, where *inf1* is assigned a value for all errors, and *inf2* is assigned a value for some errors. You must specify both arguments, as there are no optional arguments for C. For a description of EINFO, see "EINFO—ESSL Error Information-Handler Subroutine" on page 968.

Step 8. Check the Values in the Information Receivers: These statements check the values returned in the output argument information receivers, *inf1* and *inf2*, which contain the information about the computational error.

Step 9. Call ERRSTR

```
errstr (ierno,storarea);
```

(This is an optional step.) This calls the ERRSTR subroutine, which stores an entry in the error option table for error number *ierno* from an 8-byte storage area, *storarea*, which is accessible to your program. ERRSTR must be called for each entry you want to store. This step is used, along with step 4, for ERRSAV. For information on whether you should use ERRSAV and ERRSTR, see " How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?" on page 53. For an example, see "Example."

Example

This example shows an error code 2105, which returns one piece of information: the index of the pivot element (*i*) near zero, causing factorization to fail. It uses ERRSAV and ERRSTR to insulate the effects of the error handling for error 2105 by this program.

```
.
.
            /*GLOBAL STATEMENTS FOR ESSL ERROR HANDLING*/
#define _ESVERR
#include <essl.h>
#include <iostream.h>
#include <stdio.h>
.
.
            /*DECLARE THE VARIABLES*/
main ()
{
int ierno,inoal,inomes,itrace,iusadr,irange,irc;
int inf1,inf2,dummy;
char sav2105[8];
.
.
            /*INITIALIZE THE ESSL ERROR OPTION TABLE*/
dummy = 0;
einfo (0,dummy,dummy);
            /*SAVE THE EXISTING ERROR OPTION TABLE ENTRY
              FOR ERROR CODE 2105*/
ierno = 2105;
errsav (ierno,sav2105);
.
.
```

```
            /*MAKE ERROR CODES 2101 THROUGH 2105 RECOVERABLE
              ERRORS AND SUPPRESS PRINTING ALL ERROR MESSAGES
              FOR THEM.  THIS SHOWS HOW YOU CODE THE
              ERRSET ARGUMENTS FOR A RANGE OF ERRORS. */
ierno = 2101;
inoal = 0;
inomes = 0; /*A DUMMY ARGUMENT*/
itrace = 0; /*A DUMMY ARGUMENT*/
iusadr = 0; /*A DUMMY ARGUMENT*/
irange = 2105
errset (ierno,inoal,inomes,itrace, iusadr,irange);

.
.
            /*CALL ESSL SUBROUTINE DGEICD. IF THE INPUT MATRIX
              IS SINGULAR OR NEARLY SINGULAR, ERROR 2105
              OCCURS.  A RETURN CODE OF 2 IS SET IN IRC.*/
irc = dgeicd (a,lda,n,iopt,rcond,det,aux,naux);
if irc == 2
   {
            /*CALL THE INFORMATION-HANDLER ROUTINE FOR ERROR
              CODE 2105 TO RETURN ONE PIECE OF INFORMATION
              IN VARIABLE INF1, THE INDEX OF THE PIVOT ELEMENT
              NEAR ZERO, CAUSING FACTORIZATION TO FAIL.
              INF2 IS NOT USED, BUT MUST BE SPECIFIED.
              BOTH INF1 AND INF2 ARE PASSED BY REFERENCE,
              BECAUSE THEY ARE OUTPUT SCALAR ARGUMENTS.*/
   ierno = 2105;
   einfo (ierno,inf1,inf2);
            /*CHECK THE VALUE IN VARIABLE INF1 AND TAKE THE
              DESIRED ACTION*/
   .
   .
   }

.
.
            /*RESTORE THE PREVIOUS ERROR OPTION TABLE ENTRY
              FOR ERROR CODE 2105.  ERROR PROCESSING
              RETURNS TO HOW IT WAS BEFORE IT WAS ALTERED BY
              THE ABOVE ERRSAV STATEMENT*/
ierno = 2105;
errstr (ierno,sav2105);
.
.
}
```

PL/I Programs

If you are using ESSL with PL/I Set for AIX, Version 1, see the following PL/I publications for details on calling subroutines and functions:

- *PL/I Set for AIX Programming Guide*
- *PL/I Set for AIX Language Reference*

Chapter 5. Processing Your Program

This chapter provides information on how to process your program. It contains processing procedures that apply to Fortran, C, and C++ programs using ESSL. It describes **only the ESSL-specific changes** you need to make to your job setup procedures. For complete examples of job setup procedures, see your programming language's programming guide.

Notes:

1. If your existing application program calls ESSL, you only need to re-link your program to use the new ESSL SMP or Thread-Safe Library.
2. For the ESSL SMP Library, you can use the XL Fortran XLSMPOPTS environment variable to specify options which affect SMP execution. For details, see *IBM XL Fortran for AIX User's Guide Version 5*.
3. If you are using the ESSL POWER2 Library, you must be running on a RS/6000 POWER2 processor.

Compiling

This section describes how to compile your program.

General Procedures

You can use any procedures you are currently using for compiling your program. ESSL requires no changes to the compile-time setup procedures for C or C++ programs, because the ESSL header file, `essl.h`, which is used for C and C++ programs, is installed in the `/usr/include` directory.

ESSL supports the XL Fortran compile-time option **-qextname**. For details, see the Fortran manuals.

You must use only the allowable compilers or assemblers listed in Table 2 on page 8.

Using Your Own Complex Data Definitions in C Programs

If you want to specify your own definitions for short- and long-precision complex data, add `-D_CMPLX` and `-D_DCMPLX`, respectively, to your compile command, as shown here:

ESSL Library Name	Command
SMP or Thread-Safe	`cc_r -cO -D_CMPLX -D_DCMPLX xyz.c`
POWER2 or POWER	`cc -cO -D_CMPLX -D_DCMPLX xyz.c`

where *xyz*.c is the name of your C program. Otherwise, you automatically use the definitions of short- and long-precision complex data provided in the ESSL header file.

© Copyright IBM Corp. 1997

Using Your Own Short Complex Data Definitions in C++ Programs

If you want to specify your own definition for short-precision complex data, add `-D_CMPLX` to your command, as shown here:

ESSL Library Name	Command
SMP or Thread-Safe	`xlC_r -c0 -D_CMPLX xyz.C`
POWER2 or POWER	`xlC -c0 -D_CMPLX xyz.C`

where *xyz*.C is the name of your C++ program. Otherwise, you automatically use the definition of short-precision complex data provided in the ESSL header file.

Compiling and Linking

You can use any procedures you are currently using to link or run your program, as long as you make the necessary modifications for ESSL. This section describes these modifications. For details on the complete procedures, see your operating system and programming language manuals.

Fortran Programs

If you are accessing ESSL from a Fortran program and want to compile and link in one step, you can use the following command:

ESSL Library Name	Command
SMP	`xlf_r -O -qnosave xyz.f -lesslsmp`
Thread-Safe	`xlf_r -O -qnosave xyz.f -lessl_r`
POWER2	`xlf -O xyz.f -lesslp2`
POWER	`xlf -O xyz.f -lessl`

where *xyz*.f is the name of your Fortran program.

If you want to compile and link your Fortran program in separate steps, you can use the following commands:

ESSL Library Name	Command
SMP	`xlf_r -O -c -qnosave xyz.f` `xlf_r xyz.o -lesslsmp`
Thread-Safe	`xlf_r -O -c -qnosave xyz.f` `xlf_r xyz.o -lessl_r`

ESSL Library Name	Command
POWER2	`xlf -O -c xyz.f` `xlf xyz.o -lesslp2`
POWER	`xlf -O -c xyz.f` `xlf xyz.o -lessl`

where *xyz*.f is the name of your Fortran program, and *xyz*.0 is the name of your object file.

C Programs

If you are accessing ESSL from a C program and want to compile and link in one step, you can use the following command:

ESSL Library Name	Command
SMP	`cc_r -O xyz.c -lesslsmp`
Thread-Safe	`cc_r -O xyz.c -lessl_r`
POWER2	`cc -O xyz.c -lesslp2`
POWER	`cc -O xyz.c -lessl`

where *xyz*.c is the name of your C program.

If you want to compile and link your C program in separate steps, you can use the following commands:

ESSL Library Name	Command
SMP	`cc_r -c0 xyz.c` `cc_r xyz.o -lesslsmp`
Thread-Safe	`cc_r -c0 xyz.c` `cc_r xyz.o -lessl_r`
POWER2	`cc -c0 xyz.c` `cc xyz.o -lesslp2`
POWER	`cc -c0 xyz.c` `cc xyz.o -lessl`

where *xyz*.c is the name of your C program and *xyz*.o is the name of your object file.

In the above cases, you automatically use the definitions of short- and long-precision complex data provided in the ESSL header file. If you prefer to specify your own definitions for short- and long-precision complex data, add `-D_CMPLX` and `-D_DCMPLX`, respectively, to your commands, as shown here:

ESSL Library Name	Command
SMP	`cc_r -O -D_CMPLX -D_DCMPLX xyz.c -lesslsmp`
Thread-Safe	`cc_r -O -D_CMPLX -D_DCMPLX xyz.c -lessl_r`
POWER2	`cc -O -D_CMPLX -D_DCMPLX xyz.c -lesslp2`
POWER	`cc -O -D_CMPLX -D_DCMPLX xyz.c -lessl`

C++ Programs

If you are accessing ESSL from a C++ program and want to compile and link in one step, you can use the following command:

ESSL Library Name	Command
SMP	`xlC_r -O xyz.C -lesslsmp`
Thread-Safe	`xlC_r -O xyz.C -lessl_r`
POWER2	`xlC -O xyz.C -lesslp2`
POWER	`xlC -O xyz.C -lessl`

where *xyz*.C is the name of your C++ program.

If you want to compile and link your C++ program in separate steps, you can use the following commands:

ESSL Library Name	Command
SMP	`xlC_r -c0 xyz.C` `xlC_r xyz.o -lesslsmp`
Thread-Safe	`xlC_r -c0 xyz.C` `xlC_r xyz.o -lessl_r`
POWER2	`xlC -c0 xyz.C` `xlC xyz.o -lesslp2`
POWER	`xlC -c0 xyz.C` `xlC xyz.o -lessl`

where *xyz*.C is the name of your C++ program, and *xyz*.o is the name of your object file.

In the above cases, you automatically use the definition of short-precision complex data provided in the ESSL header file. If you prefer to specify your own definition for short-precision complex data, add -D_CMPLX to your commands, as shown here:

ESSL Library Name	Command
SMP	`xlC_r -O -D_CMPLX xyz.C -lesslsmp`
Thread-Safe	`xlC_r -O -D_CMPLX xyz.C -lessl_r`
POWER2	`xlC -O -D_CMPLX xyz.C -lesslp2`
POWER	`xlC -O -D_CMPLX xyz.C -lessl`

Chapter 6. Migrating Your Programs

This chapter explains many aspects of migrating your application programs to use the ESSL subroutines. It covers:

- Migrating ESSL Version 2 programs to Version 3
- Planning for future migration
- Migrating between RS/6000 processors
- Migrating from other libraries to ESSL

Migrating ESSL Version 2 Programs to Version 3

This section describes all the aspects of migrating your ESSL Version 2 application programs to Version 3.

Note: For a list of the new features and subroutines added in ESSL Version 3, see "What's New for ESSL for AIX" on page xxxi.

ESSL Subroutines

The calling sequences for the subroutines in the ESSL Version 2 and ESSL Version 3 product are identical. This includes the new ESSL SMP and Thread-Safe Libraries that are included in the ESSL Version 3 product. You do not have to change your existing application programs that call ESSL subroutines when migrating to the ESSL Version 3 product. You must, however, re-link your application program. Therefore, you can simply re-link your existing programs to take advantage of the the increased performance of using the ESSL SMP Library on the SMP processors.

For the _GEF and _GEFCD subroutines, the first column of the matrix $\boldsymbol{L}$ with the corresponding $\boldsymbol{U}_{ii} = 0$ diagonal element is identified in a computational error message. Previously, the last column was identified. You do not have to make any modifications to your existing application programs that call these subroutines.

ESSL Messages

The text of message format has changed. **ESV** has been removed from the message text. For details on the new format, see "Message Format" on page 178.

Some input-argument and computational error message numbers have been changed. The old message numbers can still be used when calling ERRSET, however, you should migrate to the new message numbers. The following describes which error messages have been modified:

Old Error Message Number	New Error Message Number
2074	2608
2123	2609
2128	2700

© Copyright IBM Corp. 1997

Planning for Future Migration

With respect to planning for the future, if working storage does not need to persist after the subroutine call, you should use dynamic allocation. Otherwise, you should use the processor-independent formulas or simple formulas for calculating the values for the *naux* arguments in the ESSL calling sequences. Two things may occur that could cause the minimum values of *naux*, returned by ESSL error handling, to increase in the future:

- If changes are made to the ESSL subroutines to improve performance
- If changes are necessary to support future processors

The formulas allow you to specify your auxiliary storage large enough to accommodate any future improvements to ESSL and any future processors. If you do not provide, at least, these amounts of storage, your program may not run in the future.

You should use the following rule of thumb: To protect your application from having to be recoded in the future because of possible increased requirements for auxiliary storage, use dynamic allocation if possible. If the working storage must persists after the subroutine call, then you should provide as much storage as possible in your current application. In determining the right amount to specify, you should weigh your storage constraints against the inconvenience of making future changes, then specify what you think is best. If possible, you should provide this larger amount of storage to prevent future migration problems.

Migrating between RS/6000 Processors

This section describes all the aspects of migrating your ESSL application programs (back and forth) between the RS/6000 PowerPC, POWER, and POWER2 processors.

Auxiliary Storage

The minimum amount of auxiliary storage returned by ESSL error handling may vary among the RS/6000 processors for the following subroutines: all the Fourier transform subroutines, SCONF, SCORF, and SACORF. Therefore, to guarantee that your application programs always migrate from any platform to any other platform, you should use the processor independent formulas to determine the amount of auxiliary storage to use.

Bitwise-Identical Results

Because of hardware and ESSL design differences, the results you obtain when migrating from one ESSL Library to another may not be bitwise identical. The results, however, are mathematically equivalent.

Migrating from Other Libraries to ESSL

This section describes some general aspects of moving from an IBM or non-IBM engineering and scientific library to ESSL.

Migrating from ESSL/370

There is a high degree of compatibility between ESSL/370 and ESSL for AIX. However you may need to make some coding changes for certain subroutines. See the *Engineering and Scientific Subroutine Library Version 2.2 Guide and Reference* for details.

Migrating from Another IBM Subroutine Library

If you are migrating from other IBM library products—such as Subroutine Library—Mathematics (SL MATH) or Scientific Subroutine Package (SSP), which have some functions similar to ESSL—the ESSL calling sequences differ from the calling sequences you are currently using. Your program must be modified to add the ESSL calling sequences and make the other ESSL-related coding changes.

If you are migrating from the Basic Linear Algebra Subroutine Library provided with the RS/6000 basic operating system, your calling sequences do not need to be changed.

Migrating from LAPACK

ESSL contains a few subroutines that conform to the LAPACK interface (see Appendix B on page APB-1). If you are using these subroutines, no coding changes are needed to migrate to ESSL.

Additionally, you may be interested in using the Call Conversion Interface (CCI) that is available with LAPACK. The CCI substitutes a call to an ESSL subroutine in place of an LAPACK subroutine whenever an ESSL subroutine provides either functional or near-functional equivalence. Using the CCI allows LAPACK users to obtain the optimized performance of ESSL for an additional subset of LAPACK subroutines. For details, see reference [40].

Migrating from a Non-IBM Subroutine Library

If you are using a non-IBM library, ESSL may provide subroutines corresponding to those you are currently using. You may choose to migrate your program to benefit from the increased performance offered by the ESSL subroutines. In this case, you may have to recode your program to use the ESSL calling sequences, because the names and arguments used by ESSL may be different from those used by the non-IBM library. On the other hand, if you are using any of the standard Level 1, 2, and 3 BLAS or LAPACK routines that correspond to ESSL subroutines, you do not need to recode the calling sequences. The ESSL calling sequences are the same as the public domain code.

Chapter 7. Handling Problems

This chapter provides the following information for your use when dealing with errors:

- How to obtain IBM support.
- What to do about NLS (National Language Support) problems.
- A description of the different types of errors that can occur in ESSL. It explains what happens when an error occurs and, in some instances, how you can use error handling to obtain further information.
- All of the ESSL error messages are categorized into the different error types. There is also a description of the error message format.

Where to Find More Information About Errors

Specific errors associated with each ESSL subroutine are listed under "Error Conditions" in each subroutine description in Part 3 of this book.

Getting Help from IBM Support

Should you require help from IBM in resolving an ESSL problem, report it and provide the following information, if available and appropriate.

1. Your customer number
2. The ESSL program number:
   - 5765-C42

   This is important information that speeds up the correct routing of your call.
3. The version and release of the operating system that you are running on. To get this information on AIX, enter the following command:

   ```
   oslevel
   ```
4. The names and versions of key products being run. To get this information on AIX, enter the following command:

   ```
   lslpp  -h product
   ```

   where the appropriate values of *product* for AIX Version 4 are listed in Table 30.

Table 30. Product File Set Names when Using AIX Version 4

Product File Sets	**Descriptive Name**
essl.common.rte	ESSL
xlfrte	XL Fortran Run-Time Environment
xlfcmp	XL Fortran Compiler
xlC.C	C for AIX Compiler
xlC.C++.cmp	C Set ++ for AIX Compiler

5. The message that is returned when an error is detected.

© Copyright IBM Corp. 1997

6. Any error message relating to core dumps.
7. The compiler listings, including compiler options in effect, and any run-time listings produced
8. Program changes made in comparison with a previous successful run
9. A small test case demonstrating the problem using the minimum number of statements and variables, including input data

Consult your IBM Service representative for more assistance.

National Language Support

For National Language Support (NLS), all ESSL subroutines display messages located in externalized message catalogs. English versions of the message catalogs are shipped with the IBM Engineering and Scientific Subroutine Library for AIX product, but your site maybe using its own translated message catalogs. The AIX environment variable **NLSPATH** is used by the various ESSL subroutines to find the appropriate message catalog. **NLSPATH** specifies a list of directories to search for message catalogs. The directories are searched, in the order listed, to locate the message catalog. In resolving the path to the message catalog, **NLSPATH** is affected by the value of the environment variables **LC_MESSAGES** and **LANG**. If you get an error saying that a message catalog is not found and want the default message catalog, enter the following:

```
export NLSPATH = /usr/lib/nls/msg/%L/%N

export LANG = C
```

The ESSL message catalogs are in English, and are located in the following directories:

```
/usr/lib/nls/msg/C
/usr/lib/nls/msg/En_US
/usr/lib/nls/msg/en_US
```

If your site is using its own translations of the message catalogs, consult your system administrator for the appropriate value of **NLSPATH** or **LANG**. For additional information on NLS and message catalogs, see *IBM AIX Version 4 for RISC System/6000 General Programming Concepts*.

If ESSL cannot successfully find a message, ESSL returns message 2799, indicating which message could not be located. Message 2799 is described in "Miscellaneous Error Messages" on page 178.

Dealing with Errors

At run time, you can encounter a number of different types of errors that are specifically related to the use of the ESSL subroutines:

- Program exceptions
- Input-argument errors (2001-2099) and (2801-2899)
- Computational errors (2100-2199)
- Resource errors (2401-2499)
- Informational and Attention messages (2600-2699)

- Miscellaneous errors (2700-2799)

Program Exceptions

The program exceptions you can encounter in ESSL are described in the RS/6000 architecture manuals. For details, see:

- *ANSI/IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*.
- *RS/6000 POWERstation and POWERserver Hardware Technical Reference—General Information*.

ESSL Input-Argument Error Messages

If you receive an error message in the form 2538-20*nn*, you have an input-argument error in the calling sequence for an ESSL subroutine. Your program terminated at this point unless you did one of the following:

- Specified the ESSL user exit routine, ENOTRM, with ERRSET to determine the correct input argument values in your program for the optionally-recoverable ESSL errors 2015 or 2030. For details on how to do this, see Chapter 4 on page 111.
- Reset the number of allowable errors (2099) during ESSL installation or using ERRSET in your program. **This is not recommended for input-argument errors.**

Note: For many of the ESSL subroutines requiring auxiliary storage, you can avoid program termination due to error 2015 by allowing ESSL to dynamically allocate auxiliary storage for you. You do this by setting *naux* = 0 and making error 2015 unrecoverable. For details on which *aux* arguments allow dynamic allocation and how to specify them, see the subroutine descriptions in Part 2 of this book.

The name of the ESSL subroutine detecting the error is listed as part of the message. The argument number(s) involved in the error appears in the message text. See "Input-Argument Error Messages" on page 179 for a complete description of the information contained in each message and for an indication of which messages correspond to optionally-recoverable errors. Regardless of whether the name in the message is a user-callable ESSL subroutine or an internal ESSL routine, the message-text and its unique parts apply to the user-callable ESSL subroutine. Return code values are described under "Error Conditions" for each ESSL subroutine in Part 2 of this book.

You may get more than one error message, because most of the arguments are checked by ESSL for possible errors during each call to the subroutine. The ESSL subroutine returns as many messages as there are errors detected. As a result, fewer runs are necessary to diagnose your program.

Fix the error(s), recompile, relink, and rerun your program.

ESSL Computational Error Messages

If you receive an error message in the form 2538-21*nn*, you have a computational error in the ESSL subroutine. A computational error is any error occurring in the ESSL subroutine while using the computational data (that is, scalar and array data). The name of the ESSL subroutine detecting the error is listed as part of the message. Regardless of whether the name in the message is a user-callable ESSL subroutine or an internal ESSL routine, the message-text and its unique parts apply to the user-callable ESSL subroutine. A nonzero return code is returned when the ESSL subroutine encounters a computational error. See "Computational Error Messages" on page 187 for a complete description of the

information in each message. Return code values are described under "Error Conditions" for each ESSL subroutine in Part 2 of this book.

Your program terminates for some computational errors unless you have called ERRSET to reset the number of allowable errors for that particular error, and the number has not been exceeded. A message is issued for each computational error. You should use the message to determine where the error occurred in your program.

If you called ERRSET and you have not reached the limit of errors you had set, you can check the return code. If it is not 0, you should call the EINFO subroutine to obtain information about the data involved in the error. EINFO provides the same information provided in the messages; however, it is provided to your program so your program can check the information during run time. Depending on what you want to do, you may choose to continue processing or terminate your program after the error occurs. For information on how to make these changes in your program to reset the number of allowable errors, how to diagnose the error, and how to decide whether to continue or terminate your program, see Chapter 4 on page 111.

If you are unable to solve the problem, report it and provide the following information, if available and appropriate:

- The message number and the module that detected an error
- The system dump, system error code, and system log of this job
- The compiler listings, including compiler options in effect, and any run-time listings produced
- Program changes made in comparison with a previous successful run
- A small test case demonstrating the problem using the minimum number of statements and variables, including input data
- A brief description of the problem

ESSL Resource Error Messages

If you receive a message in the form 2538-24*nn*, it means that ESSL issued a resource error message.

A resource error occurs when a buffer storage allocation request fails in a ESSL subroutine. In general, the ESSL subroutines allocate internal auxiliary storage dynamically as needed. Without sufficient storage, the subroutine cannot complete the computation.

When a buffer storage allocation request fails, a resource error message is issued, and the application program is terminated. You need to reduce the memory constraint on the system or increase the amount of memory available before rerunning the application program.

The following ways may reduce memory constraints:

- Investigate the load of your process and run in a more dedicated environment.
- Increase your processor's paging space.
- Select a machine with more memory.

- Consider specifying the `-bmaxdata` binder option when linking your program (see the *IBM AIX Version 4 Commands Reference* and *IBM XL Fortran for AIX User's Guide Version 5*).
- Check the setting of your user ID's user limit (`ulimit`). (see the *IBM AIX Version 4 Commands Reference*).

ESSL Informational and Attention Messages

If you receive a message in the form 2538-26*nn*, it means that ESSL issued an informational or attention message.

Informational Messages

When you receive an informational message, check your application to determine why the condition was detected.

ESSL Attention Messages

An attention message is issued to describe a condition that occurred. ESSL is able to continue processing, but performance may be degraded.

One condition that may produce an attention message is when enough work area was available to continue processing, but was not the amount initially requested. ESSL does not terminate your application program, but performance may be degraded. If you want to reduce the memory constraint on the system or increase the amount of memory available to eliminate the attention message, see the suggestions in "ESSL Resource Error Messages" on page 176. For a list of subroutines that may generate this type of attention message, see Table 31.

Table 31 (Page 1 of 2). ESSL Subroutines

Subroutine Names
Matrix-Vector Linear Algebra Subprograms: _GEMV, _GER, _SPMV, _SYMV, _SPR, _SYR, _SPR2, _SYR2 _GERC, _GERU, _HPMV, _HEMV, _HPR, _HER, _HPR2, _HER2 _SBMV, _TBMV, SGBMV, DGBMV, CGBMV _TPMV _TRMV
Matrix Operations: _GEMM, _GEMUL _SYMM, _SYR2K, _TRMM _HEMM, _HER2K
Dense Linear Algebraic Equations: _POF, _POICD, _PPICD _GEICD, _TPI, _TRI _TRSM, _TPSV _TRSV
Banded Linear Algebraic Equations: STBSV, DTBSV
Linear Least Squares: _GESVS

Table 31 (Page 2 of 2). ESSL Subroutines
Subroutine Names
Fourier Transforms: _CFT, _CFT3 _RCFT, _RCFT3 _CRFT, _CRFT3

Miscellaneous Error Messages

If you receive a message in the form 2538-27*nn*, it means that ESSL issued a miscellaneous error message.

A miscellaneous error is an error that does not fall under any other categories.

When ESSL detects a miscellaneous error, you receive an error message with information on how to proceed and your application program is terminated.

Messages

This section explains the conventions used for the ESSL messages and lists all the ESSL messages. For a description of each of the four types of ESSL messages, see "Dealing with Errors" on page 174.

Message Conventions

This section describes the message conventions for the ESSL product.

About Upper- and Lowercase

The literals, such as, 'N', 'T', 'U', and so forth, appear in the messages in this book in uppercase; however, they may be specified in your ESSL calling sequence in either upper- or lowercase, for example, 'n', 't', and 'u'.

Message Format

The ESSL messages are issued in your output in the following format:

```
rtn-name : 2538-mmnn
message-text
```

Figure 10. Message Format

The parts of the ESSL message are as follows:

rtn-name gives the name of the ESSL subroutine that encountered the error.

2538 is the ESSL component identification number.

mm indicates the type of ESSL error message:

- 20—Input-argument error message
- 21—Computational error message
- 24—Resource error message
- 26—Information and attention message
- 27—Miscellaneous error message

nn is the message identification number.

message-text describes the nature of the error. Where one of several possible message-texts can be issued for a particular ESSL error, they are listed in this book with an "or" between them. The possible unique parts are:

- The argument number of each argument involved in the error is included in the message description as (ARG NO. _)
- Additional information about the error is included in the message. The placement of this information is shown in the messages as (_)

Input-Argument Error Messages

```
RTN_NAME : 2538-2001
The number of elements (ARG NO. _) in a vector must be greater than
or equal to zero.

RTN_NAME : 2538-2002
The stride (ARG NO. _) for a vector must be nonzero.

RTN_NAME : 2538-2003
The number of rows (ARG NO. _) in a matrix must be greater than or equal
to zero.

RTN_NAME : 2538-2004
The number of columns (ARG NO. _) in a matrix must be greater than or
equal to zero.

RTN_NAME : 2538-2005
The size of the leading dimension (ARG NO. _) of an array must be greater
than zero.

RTN_NAME : 2538-2006
The number of rows (ARG NO. _) of a matrix must be less than or equal
to the size of the leading dimension (ARG NO. _) of its array.

RTN_NAME : 2538-2007
The degree of a polynomial (ARG NO. _) must be greater than or equal to zero.

RTN_NAME : 2538-2008
The number of elements (ARG NO. _) to be scanned must be greater than or
equal to 2.

RTN_NAME : 2538-2009
The number of elements (ARG NO. _) in a vector to be processed must be
greater than or equal to 3.

RTN_NAME : 2538-2010
The transform length (ARG NO. _) must be a power of 2.

RTN_NAME : 2538-2011
The number of points used in the interpolation (ARG NO. _) must be
greater than or equal to zero and less than or equal to the number of data
points (ARG NO. _).

RTN_NAME : 2538-2012
The transform length (ARG NO. _) must be less than or equal to (_).
```

```
RTN_NAME : 2538-2013
The transform length (ARG NO. _) must be greater than or equal to (_).

RTN_NAME : 2538-2014
The routine must be initialized with the present value of (ARG NO. _).

RTN_NAME : 2538-2015
The number of elements (ARG NO. _) in a work array must be greater than
or equal to (_).

RTN_NAME : 2538-2016
The form (ARG NO. _) of a matrix must be 'N' or 'T'.
   or
The form (ARG NO. _) of a matrix must be 'N', 'T', or 'C'.
   or
The form (ARG NO. _) of a matrix must be 'N' or 'C'.

RTN_NAME : 2538-2017
The dimension (ARG NO. _) of the matrices must be greater than or equal to zero.

RTN_NAME : 2538-2018
The matrix form is specified by (ARG NO. _); therefore, the leading
dimension (ARG NO. _) of its array must be greater than or equal to the
number of its rows (ARG NO. _).

RTN_NAME : 2538-2019
The number of sequences (ARG NO. _) must be greater than zero.

RTN_NAME : 2538-2020
(ARG NO. _) must be nonzero.

RTN_NAME : 2538-2021
The storage control switch (ARG NO. _) must be 1, 2, 3, or 4.

RTN_NAME : 2538-2022
(ARG NO. _) must be less than (_).

RTN_NAME : 2538-2023
The outer loop increment (ARG NO. _) must be greater than or equal to zero.

RTN_NAME : 2538-2024
The stride (ARG NO. _) for a vector must be greater than or equal to zero.

RTN_NAME : 2538-2025
The stride (ARG NO. _) for a vector must be greater than zero.

RTN_NAME : 2538-2026
The stride (ARG NO. _) for a vector must be greater than or equal to (_).

RTN_NAME : 2538-2027
The order (ARG NO. _) of a matrix must be greater than or equal to zero.

RTN_NAME : 2538-2028
The job option argument (ARG NO. _) must be 0, 1, or 2.
   or
The job option argument (ARG NO. _) must be 0, 1, 2, or 3.
```

```
   or
The job option argument (ARG NO. _) must be 0, 1, 2, 10, 11, or 12.
   or
The job option argument (ARG NO. _) must be 0, 1, 10, or 11.
   or
The job option argument (ARG NO. _) must be 0, 1, 20, or 21.
   or
The job option argument (ARG NO. _) must be 0, 1, 10, 11, 20, 21, 30,
or 31.
   or
The job option argument (ARG NO. _) must be 0, 1, 2, 3, or 4.

RTN_NAME : 2538-2029
The job option argument (ARG NO. _) must be 0 or 1.

RTN_NAME : 2538-2030
The transform length (ARG NO. _) is not an allowed value. The next higher
allowed value is (_).

RTN_NAME : 2538-2031
The resulting convolution length obtained from ARG NO. 10 = (_),
ARG NO.11 = (_), ARG NO.13 = (_), and ARG NO.14 = (_)
must be less than (_).

RTN_NAME : 2538-2032
The size of the leading dimension (ARG NO. _) of the matrix must be greater
than or equal to (_), the bandwidth constraint.

RTN_NAME : 2538-2033
The lower bandwidth (ARG NO. _) must be greater than or equal to zero.

RTN_NAME : 2538-2034
The upper bandwidth (ARG NO. _) must be greater than or equal to zero.

RTN_NAME : 2538-2035
The half-band bandwidth (ARG NO. _) must be greater than or equal to zero.

RTN_NAME : 2538-2036
The lower bandwidth (ARG NO. _) must be less than the order (ARG NO. _)
of the matrix.

RTN_NAME : 2538-2037
The upper bandwidth (ARG NO. _) must be less than the order (ARG NO. _)
of the matrix.

RTN_NAME : 2538-2038
The half-band bandwidth (ARG NO. _) must be less than the order (ARG NO. _)
of the matrix.

RTN_NAME : 2538-2039
(ARG NO. _) must be greater than zero.

RTN_NAME : 2538-2040
Insufficient storage allocated for positive definite solve.
(_) additional bytes required.
```

RTN_NAME : 2538-2041
The resulting correlation length obtained from ARG NO. 8 = (_) and
ARG NO. 10 = (_) must be less than (_).

RTN_NAME : 2538-2042
(ARG NO. _) must be greater than or equal to zero.

RTN_NAME : 2538-2043
(ARG NO. _) must be greater than (_).

RTN_NAME : 2538-2044
The number of initialized coefficients (ARG NO. _) cannot exceed the size
of the coefficient vector (ARG NO. _).

RTN_NAME : 2538-2045
The order specified (ARG NO. _) is not supported for this quadrature method.
The nearest supported order is (_).

RTN_NAME : 2538-2046
The scaling parameter (ARG NO. _) must be greater than zero for this
quadrature method.

RTN_NAME : 2538-2047
The scaling parameter (ARG NO. _) must be nonzero for this quadrature
method.

RTN_NAME : 2538-2048
The sum of (ARG NO. _) and (ARG NO. _) must be nonzero for this quadrature
method.

RTN_NAME : 2538-2049
The number of data points (ARG NO. _) must be greater than one in order
to perform numerical quadrature.

RTN_NAME : 2538-2050
The number of columns specified for the arrays to store the matrix in
compressed matrix mode (ARG NO. _) must be greater than or equal to (_).

RTN_NAME : 2538-2051
The number of columns (ARG NO. _) specified for the matrix used to store
the sparse matrix in compressed mode must be greater than zero.

RTN_NAME : 2538-2052
The total number of non-zero elements of the input sparse matrix stored by
rows, obtained from element (_) of the row pointers array (ARG NO. _),
must be greater than or equal to zero.

RTN_NAME : 2538-2053
The number of non-zero elements in row (_) obtained from the row pointer
array (ARG NO. _) is less than zero.

RTN_NAME : 2538-2054
The number of diagonals (ARG NO. _) specified for the matrix used to store the
sparse matrix in compressed diagonal mode must be greater than zero.

RTN_NAME : 2538-2055
Element (_) of the vector used to store the diagonal numbers (ARG NO. _)
is incompatible with the order of the sparse matrix (ARG NO. _).

RTN_NAME : 2538-2056
The matrix is singular because the number of non-zero entries (ARG NO. _)
is zero.

RTN_NAME : 2538-2057
Element (_) in the integer parameter vector (ARG NO. _) must be greater than
or equal to zero.

RTN_NAME : 2538-2058
Element (_) in the integer parameter vector (ARG NO. _) must be (_),(_), or (_).

RTN_NAME : 2538-2059
Element (_) in the real parameter vector (ARG NO. _) must be greater than zero.

RTN_NAME : 2538-2060
The size of the leading dimension (ARG NO. _) of an array must be greater
than or equal to the maximum of (ARG NO. _) and (ARG NO. _).

RTN_NAME : 2538-2061
Parameter (ARG NO. _), which specifies the number of columns of the input
sparse matrix (ARG NO. _ and ARG NO. _) must be greater than or equal to (_).

RTN_NAME : 2538-2062
The number of random numbers generated (ARG NO. _) must be even and
greater than or equal to zero.

RTN_NAME : 2538-2063
SIDE (ARG NO. _), which specifies whether the triangular input matrix
(ARG NO. _) appears on the left or right of the other input matrix, must
be 'L' or 'R'.

RTN_NAME : 2538-2064
UPLO (ARG NO. _), which specifies whether an input matrix (ARG NO. _)
is upper or lower triangular, must be 'U' or 'L'.

RTN_NAME : 2538-2065
DIAG (ARG NO. _), which specifies whether an input matrix (ARG NO. _)
is unit triangular, must be 'U' or 'N'.

RTN_NAME : 2538-2066
Given the value which has been assigned to SIDE (ARG NO. _), the leading
dimension (ARG NO. _) for the triangular input matrix must be greater
than or equal to (ARG NO. _).

RTN_NAME : 2538-2067
TRANSA (ARG NO. _) specifies whether an input matrix (ARG NO. _), its transpose,
or its conjugate transpose should be used. TRANSA must be 'N', 'T', or 'C'.

RTN_NAME : 2538-2068
The size of the leading dimension (ARG NO. _) of an array must be greater
than or equal to zero.

RTN_NAME : 2538-2069
The vector section size of the scalar library (ARG NO. _) must be 128 or 256.

```
RTN_NAME : 2538-2070
Element (_) in (ARG NO. _) must be 0 or 1.
   or
Element (_) in (ARG NO. _) must be greater than zero.
   or
Element (_) in (ARG NO. _) must be greater than or equal to zero.
   or
Element (_) in (ARG NO. _) must be greater than or equal to zero and
less than or equal to 1.
   or
Element (_) in (ARG NO. _) must be greater than the preceding element.
   or
Element (_) in (ARG NO. _) must be greater than or equal to 1 and
less than or equal to n.
   or
Element (_) in (ARG NO. _) must be -1 or 1.
   or
Element (_) in (ARG NO. _) must be nonzero.
   or
Element (_) in (ARG NO. _) must be 0, 1, 2, 10, or 11.
   or
Element (_) in (ARG NO. _) must be 0, 1, 2, 10, 11, 100, 102, or 110.
   or
Element (_) in (ARG NO. _) must be 0.
   or
Element (_) in (ARG NO. _) must be 1.
   or
Element (_) in (ARG NO. _) must be 0, 1, 2, 10, 11, 100, 101, 102,
110, or 111.
   or
Element (_) in (ARG NO. _) must be 1, 2, 3, or 4.
   or
Element (_) in (ARG NO. _) must be 1, 2, 3, 4, or 5.

RTN_NAME : 2538-2071
The number of eigenvalues (ARG NO. _) must be less than or equal to the
order of the matrix (ARG NO. _).

RTN_NAME : 2538-2072
The work area (ARG NO. _) does not contain a valid vector seed.  The
routine must be called with a nonzero value of ISEED (ARG NO. _).

RTN_NAME : 2538-2073
(ARG NO. _) must be a double precision whole number greater than or equal
to 1.0 and less than 2147483647.0.

RTN_NAME : 2538-2074
Performance can be improved by using a larger work array. For best performance,
specify the number of elements (ARG NO. _) in the work array to be greater
than or equal to (_).

RTN_NAME : 2538-2075
The data type parameter (ARG NO. _) must be 'S', 'D', 'C', or 'Z'.

RTN_NAME : 2538-2076
(ARG NO. _) must be greater than or equal to (_) and smaller than (_).
```

RTN_NAME : 2538-2077
The matrix is singular. Column (_) is empty in the matrix specified by
(ARG NO. _), (ARG NO. _), and (ARG NO. _).

RTN_NAME : 2538-2078
The matrix is singular. Row (_) is empty in the matrix specified by
(ARG NO. _), (ARG NO. _), and (ARG NO. _).

RTN_NAME : 2538-2079
The matrix, specified by (ARG NO. _), (ARG NO. _), and (ARG NO. _),
contains at least one duplicate column index in row (_).

RTN_NAME : 2538-2080
Element (_) in (ARG NO. _) must be greater than or equal to (_)
and less than or equal to (_).
   or
Element (_) in (ARG NO. _) must be greater than or equal to (_) and
less than or equal to (ARG NO. _).
   or
Element (_) in (ARG NO. _) must be greater than or equal to element (_)
and less than or equal to (_).
   or
Element (_) in (ARG NO. _) must be zero or must be greater than or
equal to (_).

RTN_NAME : 2538-2081
Element (_) in (ARG NO. _) must be less than or equal to (_).

RTN_NAME : 2538-2082
Element (_) in (ARG NO. _) may cause incorrect or misleading results.
A nonzero number with absolute value less than or equal to 1 is recommended.
   or
Element (_) in (ARG NO. _) may cause incorrect or misleading results.
A positive number less than or equal to 1 is recommended.

RTN_NAME : 2538-2083
The pivot tolerance (element (_) in (ARG NO. _)) may cause incorrect
or misleading results. A number greater than or equal to 0 and less than or
equal to 1 is recommended.

RTN_NAME : 2538-2084
The dimension (ARG NO. _) of the array (ARG NO. _) must be greater than or
equal to (_).

RTN_NAME : 2538-2085
The number of steps after which the generalized minimum residual method is
restarted, element (_) in (ARG NO. _), must be greater than 0.

RTN_NAME : 2538-2086
The acceleration parameter, element (_) in (ARG NO. _), must be greater than 0
when using the SSOR preconditioner.

RTN_NAME : 2538-2087
STOR (ARG NO. _), which specifies the storage variation used to represent
the input sparse matrix, must be 'G', 'L', or 'U'.

```
RTN_NAME : 2538-2088
INIT (ARG NO. _), which specifies the type of computation to be performed,
must be 'I', or 'S'.

RTN_NAME : 2538-2089
Element (_) in (ARG NO. _) must be greater than or equal to (_).
   or
Element (_) in (ARG NO. _) must be greater than or equal to element (_).

RTN_NAME : 2538-2090
For level (_), the number of grid points for dimension (_) must be
an odd number greater than 1.

RTN_NAME : 2538-2091
Since the mesh spacing (ARG NO. _) here is not constant, the second order
prolongation method must be used.  That is, element (_) of (ARG NO. _)
must be (_).

RTN_NAME : 2538-2092
The index into (ARG NO. _) is out of range.
This index is element (_,_) of (ARG NO. _).

RTN_NAME : 2538-2093
The index into (ARG NO. _) is out of range.
This index is element (_,_,_) of (ARG NO. _).

RTN_NAME : 2538-2094
For dimension (_) on level (_), the mesh spacing must be changed to
a positive value.

RTN_NAME : 2538-2095
Excess space in (ARG NO. _) has been decreased and may be inadequate.
To avoid this, specify the coarse level matrix as the final item in this
argument.

RTN_NAME : 2538-2096
For level (_), the matrix type, solver, and preconditioner are incompatible.

RTN_NAME : 2538-2097
The solver requested for level (_) requires a square matrix.
Elements (_,_,_) and (_,_,_) in (ARG NO. _) must be equal.

RTN_NAME : 2538-2098
Element (_,_) of (ARG NO. _) must be greater than or equal to (_).

RTN_NAME : 2538-2099
End of input argument error reporting. For more information, refer to
Engineering and Scientific Subroutine Library Guide and Reference (SA22-7272).
```

Computational Error Messages

```
RTN_NAME : 2538-2100
The computed index of a vector is out of the range (_) to (_).

RTN_NAME : 2538-2101
Eigenvalue (_) failed to converge after (_) iterations.

RTN_NAME : 2538-2102
Eigenvector (_) failed to converge after (_) iterations.

RTN_NAME : 2538-2103
The matrix (ARG NO. _) is singular.
Zero diagonal element (_) has been detected.

RTN_NAME : 2538-2104
The matrix (ARG NO. _) is not positive definite. The last diagonal element
with nonpositive value is (_).

RTN_NAME : 2538-2105
Factorization failed due to near zero pivot number (_).

RTN_NAME : 2538-2106
Vector boundary misalignment detected in ESSL scalar library.

RTN_NAME : 2538-2107
Singular value (_) failed to converge after (_) iterations.

RTN_NAME : 2538-2108
The matrix specified by (ARG NO. _) and (ARG NO. _) is not definite
because the diagonal is not of constant sign.

RTN_NAME : 2538-2109
The matrix specified by (ARG NO. _) and (ARG NO. _) is not definite
and the iterative process is stopped at iteration number (_).

RTN_NAME : 2538-2110
The maximum allowed number of iterations, element number (_) of (ARG NO. _),
were performed but the iterative process did not converge to a solution
according to the stopping procedure.

RTN_NAME : 2538-2111
The factorization matrix (ARG NO. _) is not consistent with the sparse
matrix specified by (ARG NO. _) and (ARG NO. _).

RTN_NAME : 2538-2112
The incomplete factorization of the sparse matrix specified by
(ARG NO. _) and (ARG NO. _) is not stable.

RTN_NAME : 2538-2113
Unexpected nonzero vector mask detected in ESSL scalar routine.
Contact your IBM Service Representative.

RTN_NAME : 2538-2114
Eigenvalue (_) failed to converge after (_) iterations.
```

```
RTN_NAME : 2538-2115
The matrix (ARG NO. _) is not positive definite.
The leading minor of order (_) has a nonpositive determinant.

RTN_NAME : 2538-2116
The matrix specified by (ARG NO. _) and (ARG NO. _) is singular.

RTN_NAME : 2538-2117
The pivot element in column (_) is smaller than the first element in
(ARG NO. _).

RTN_NAME : 2538-2118
The pivot element in row (_) is smaller than the first element in
(ARG NO. _).

RTN_NAME : 2538-2119
The storage space, specified by (ARG NO. _), is insufficient.

RTN_NAME : 2538-2120
The matrix is singular. The last row processed in the matrix was row (_).

RTN_NAME : 2538-2121
The matrix is singular. the last column processed was column (_).

RTN_NAME : 2538-2122
The factorization failed. No pivot element was found in the active submatrix.

RTN_NAME : 2538-2123
Performance can be improved by specifying a larger value for (ARG NO. _).
(_) compressions were performed.

RTN_NAME : 2538-2124
The data contained in AUX1, (ARG NO. _), was computed for a different
algorithm.

RTN_NAME : 2538-2125
This subroutine initializes part of the ESSL run-time environment.  It may
be called only once, at the beginning of the run.  It must be called before
any ESSL computational routines are called.

RTN_NAME : 2538-2126
The pivot value at row (_) is not acceptable based on pivot criteria
((ARG NO. _) and (ARG NO. _)). No fixup was applicable to this pivot.
The matrix (ARG NO. _) may be singular or not definite.

RTN_NAME : 2538-2127
The pivot value at row (_) was replaced with element (_) in
(ARG NO. _).  The matrix (ARG NO. _) may be singular or not definite.

RTN_NAME : 2538-2128
Internal ESSL error. contact your IBM service representative.

RTN_NAME : 2538-2129
The matrix specified by (ARG NO. _), (ARG NO. _), and (ARG NO. _)
is not definite because the diagonal is not of constant sign or some diagonal
element is zero.
```

RTN_NAME : 2538-2130
The incomplete factorization of the sparse matrix specified by (ARG NO. _), (ARG NO. _), and (ARG NO. _) is not stable.

RTN_NAME : 2538-2131
The matrix specified by (ARG NO. _), (ARG NO. _), and (ARG NO. _) is singular.

RTN_NAME : 2538-2132
Element (_) in (ARG NO. _) indicates that factorization was done on a previous call. The data passed is not the result of a prior valid factorization.

RTN_NAME : 2538-2133
An error occurred on level (_), in the user-supplied subroutine specified by (ARG NO. _).

RTN_NAME : 2538-2134
The data contained in (ARG NO. _) is not consistent with the sparse matrix specified by (ARG NO. _), (ARG NO. _), and (ARG NO. _).

RTN_NAME : 2538-2135
For level (_), loss of orthogonality occurred in a minimum residual solver because the input matrix (element (_,_) of (ARG NO. _)) is inappropriate. Choose one of the other non-symmetric solvers.

RTN_NAME : 2538-2136
For level (_), the main diagonal element for row (_) of a matrix is 0.

RTN_NAME : 2538-2137
This subroutine may be called only once in the beginning of the program.

RTN_NAME : 2538-2138
An error was detected while attempting to open the ESSLPARM file. A default ESSL processor family number of (_) will be assumed. This may not be the best choice for optimal performance. Be sure that you have defined the interface to ESSLPARM correctly for the run. If you have, contact ESSL system installation personnel.

RTN_NAME : 2538-2139
An error was detected while attempting to close the ESSLPARM file. An ESSL processor family number of (_) was assigned.

RTN_NAME : 2538-2140
An input/output error was detected while attempting to read line (_) of the ESSLPARM file. A default ESSL process or family number of (_) will be assumed. This may not be the best choice for optimal performance. Be sure that you have defined the interface to ESSLPARM correctly for the run. If you have, contact ESSL system installation personnel.

RTN_NAME : 2538-2141
The end of file marker on line (_) of the ESSLPARM file was reached before a valid specification for the ESSL processor family number was read. A default ESSL processor family number of (_) will be assumed. This may not be the best choice for optimal performance. Be sure that you have defined the interface to ESSLPARM correctly for the run. If you have, contact ESSL system installation personnel.

RTN_NAME : 2538-2142
A syntax error was detected on line (_), column (_), of the ESSLPARM

file. No valid specification for the ESSL processor family number was read.
A default ESSL processor family number of (_) will be assumed.
This may not be the best choice for optimal performance. Contact ESSL system
installation personnel.

RTN_NAME : 2538-2143
The processor family number (_) on line (_) of the ESSLPARM file is
not in the allowable range. Please specify a processor family number greater
than or equal to (_) and less than or equal to (_). A default ESSL
processor family number of (_) will be assumed. This may not be the best
choice for optimal performance. Contact ESSL system installation personnel.

RTN_NAME : 2538-2144
This subroutine must be called before any computational ESSL routines are
called.

RTN_NAME : 2538-2145
The input matrix (ARG NO. _) is singular. The first diagonal element found to
be exactly 0, was in column (_).

RTN_NAME : 2538-2146
The input matrix (ARG NO. _) is singular. The first diagonal element found to
be exactly 0, was in column (_).

RTN_NAME : 2538-2199
End of computational error reporting. For more information, refer to
Engineering and Scientific Subroutine Library Guide and Reference (SA22-7272).

Resource Error Messages

RTN_NAME : 2538-2400
An internal buffer allocation has failed due to insufficient memory.

Informational and Attention Error Messages

RTN_NAME : 2538-2600
Performance may be degraded due to limited buffer space availability.

RTN_NAME : 2538-2601
Execution terminating due to error count for error number (_)
Message summary: Message number - Count

RTN_NAME : 2538-2602
User error corrective routine entered.
User corrective action taken. Execution continuing.

RTN_NAME : 2538-2603
Standard corrective action taken. Execution continuing.

RTN_NAME : 2538-2604
Execution terminating due to error count for error number 0.

RTN_NAME : 2538-2605
Message summary: 0   - _

RTN_NAME : 2538-2606
Serial execution is taking place since the input array is equal
to the output array and either:

```
INC2X (ARG NO. _) is not equal to 2 times INC2Y (ARG NO. _) or
INC3X (ARG NO. _) is not equal to 2 times INC3Y (ARG NO. _).

RTN_NAME : 2538-2607
Serial execution is taking place since the input array is equal
to the output array and either:
INC2X (ARG NO. _) is not equal to INC2Y (ARG NO. _) or
INC3X (ARG NO. _) is not equal to INC3Y (ARG NO. _).

RTN_NAME : 2538-2608
Performance may be improved by using a larger work array. For best
performance, specify the number of elements (ARG NO. _) in the work array
to be greater than or equal to (_).

RTN_NAME : 2538-2609
Performance may be improved by specifying a larger value for (ARG NO. _).
(_) compressions were performed.
```

Miscellaneous Error Messages

```
RTN_NAME : 2538-2700
Internal ESSL error number (_).
Contact your IBM service representative.

RTN_NAME : 2538-2703
Internal ESSL error: message number requested (_) is outside of the
valid range.  Contact your IBM service representative.
```

Part 2. Reference Information

This part of the book is organized into ten areas, providing reference information for coding the ESSL calling sequences. It is organized as follows:

- Linear Algebra Subprograms
- Matrix Operations
- Linear Algebraic Equations
- Eigensystem Analysis
- Fourier Transforms, Convolutions and Correlations, and Related Computations
- Sorting and Searching
- Interpolation
- Numerical Quadrature
- Random Number Generation
- Utilities

© Copyright IBM Corp. 1997

Chapter 8. Linear Algebra Subprograms

The linear algebra subprograms, provided in four areas, are described in this chapter.

Overview of the Linear Algebra Subprograms

This section describes the subprograms in each of the four linear algebra subprogram areas:

- Vector-scalar linear algebra subprograms (Table 32)
- Sparse vector-scalar linear algebra subprograms (Table 33)
- Matrix-vector linear algebra subprograms (Table 34)
- Sparse matrix-vector linear algebra subprograms (Table 35)

Notes:

1. The term **subprograms** is used to be consistent with the Basic Linear Algebra Subprograms (BLAS), because many of these subprograms correspond to the BLAS.
2. Some of the linear algebra subprograms were designed in accordance with the Level 1 and Level 2 BLAS de facto standard. If these subprograms do not comply with the standard as approved, IBM will consider updating them to do so. If IBM updates these subprograms, the updates could require modifications of the calling application program.

Vector-Scalar Linear Algebra Subprograms

The vector-scalar linear algebra subprograms include a subset of the standard set of Level 1 BLAS. For details on the BLAS, see reference [73]. The remainder of the vector-scalar linear algebra subprograms are commonly used computations provided for your applications. Both real and complex versions of the subprograms are provided.

Table 32 (Page 1 of 2). List of Vector-Scalar Linear Algebra Subprograms

Descriptive Name	Short-Precision Subprogram	Long-Precision Subprogram	Page
Position of the First or Last Occurrence of the Vector Element Having the Largest Magnitude	ISAMAX†▪ ICAMAX†▪	IDAMAX†▪ IZAMAX†▪	202
Position of the First or Last Occurrence of the Vector Element Having Minimum Absolute Value	ISAMIN†	IDAMIN†	205
Position of the First or Last Occurrence of the Vector Element Having Maximum Value	ISMAX†	IDMAX†	208
Position of the First or Last Occurrence of the Vector Element Having Minimum Value	ISMIN†	IDMIN†	211
Sum of the Magnitudes of the Elements in a Vector	SASUM†▪ SCASUM†▪	DASUM†▪ DZASUM†▪	214
Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in the Vector Y	SAXPY▪ CAXPY▪	DAXPY▪ ZAXPY▪	217

© Copyright IBM Corp. 1997

Table 32 (Page 2 of 2). List of Vector-Scalar Linear Algebra Subprograms

Descriptive Name	Short-Precision Subprogram	Long-Precision Subprogram	Page
Copy a Vector	SCOPY▪ CCOPY▪	DCOPY▪ ZCOPY▪	220
Dot Product of Two Vectors	SDOT†▪ CDOTU†▪ CDOTC†▪	DDOT†▪ ZDOTU†▪ ZDOTC†▪	223
Compute SAXPY or DAXPY N Times	SNAXPY	DNAXPY	227
Compute Special Dot Products N Times	SNDOT	DNDOT	232
Euclidean Length of a Vector with Scaling of Input to Avoid Destructive Underflow and Overflow	SNRM2†▪ SCNRM2†▪	DNRM2†▪ DZNRM2†▪	237
Euclidean Length of a Vector with No Scaling of Input	SNORM2† CNORM2†	DNORM2† ZNORM2†	240
Construct a Givens Plane Rotation	SROTG▪ CROTG▪	DROTG▪ ZROTG▪	243
Apply a Plane Rotation	SROT▪ CROT▪ CSROT▪	DROT▪ ZROT▪ ZDROT▪	250
Multiply a Vector X by a Scalar and Store in the Vector X	SSCAL▪ CSCAL▪ CSSCAL▪	DSCAL▪ ZSCAL▪ ZDSCAL▪	254
Interchange the Elements of Two Vectors	SSWAP▪ CSWAP▪	DSWAP▪ ZSWAP▪	257
Add a Vector X to a Vector Y and Store in a Vector Z	SVEA CVEA	DVEA ZVEA	260
Subtract a Vector Y from a Vector X and Store in a Vector Z	SVES CVES	DVES ZVES	264
Multiply a Vector X by a Vector Y and Store in a Vector Z	SVEM CVEM	DVEM ZVEM	268
Multiply a Vector X by a Scalar and Store in a Vector Y	SYAX CYAX CSYAX	DYAX ZYAX ZDYAX	272
Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in a Vector Z	SZAXPY CZAXPY	DZAXPY ZZAXPY	275

† This subprogram is invoked as a function in a Fortran program.

▪ Level 1 BLAS

Sparse Vector-Scalar Linear Algebra Subprograms

The sparse vector-scalar linear algebra subprograms operate on sparse vectors using optimized storage techniques; that is, only the nonzero elements of the vector are stored. These subprograms provide similar functions to the vector-scalar subprograms. These subprograms represent a subset of the sparse extensions to the Level 1 BLAS described in reference [29]. Both real and complex versions of the subprograms are provided.

Table 33. List of Sparse Vector-Scalar Linear Algebra Subprograms

Descriptive Name	Short-Precision Subprogram	Long-Precision Subprogram	Page
Scatter the Elements of a Sparse Vector X in Compressed-Vector Storage Mode into Specified Elements of a Sparse Vector Y in Full-Vector Storage Mode	SSCTR CSCTR	DSCTR ZSCTR	280
Gather Specified Elements of a Sparse Vector Y in Full-Vector Storage Mode into a Sparse Vector X in Compressed-Vector Storage Mode	SGTHR CGTHR	DGTHR ZGTHR	283
Gather Specified Elements of a Sparse Vector Y in Full-Vector Mode into a Sparse Vector X in Compressed-Vector Mode, and Zero the Same Specified Elements of Y	SGTHRZ CGTHRZ	DGTHRZ ZGTHRZ	286
Multiply a Sparse Vector X in Compressed-Vector Storage Mode by a Scalar, Add to a Sparse Vector Y in Full-Vector Storage Mode, and Store in the Vector Y	SAXPYI CAXPYI	DAXPYI ZAXPYI	289
Dot Product of a Sparse Vector X in Compressed-Vector Storage Mode and a Sparse Vector Y in Full-Vector Storage Mode	SDOTI† CDOTCI† CDOTUI†	DDOTI† ZDOTCI† ZDOTUI†	292

† This subprogram is invoked as a function in a Fortran program.

Matrix-Vector Linear Algebra Subprograms

The matrix-vector linear algebra subprograms operate on a higher-level data structure—matrix-vector rather than vector-scalar—using optimized algorithms to improve performance. These subprograms represent a subset of the Level 2 BLAS described in references [34] and [35]. Both real and complex versions of the subprograms are provided.

Table 34 (Page 1 of 2). List of Matrix-Vector Linear Algebra Subprograms

Descriptive Name	Short-Precision Subprogram	Long-Precision Subprogram	Page
Matrix-Vector Product for a General Matrix, Its Transpose, or Its Conjugate Transpose	SGEMV◂ CGEMV◂ SGEMX§ SGEMTX§	DGEMV◂ ZGEMV◂ DGEMX§ DGEMTX§	297

Table 34 (Page 2 of 2). List of Matrix-Vector Linear Algebra Subprograms

Descriptive Name	Short-Precision Subprogram	Long-Precision Subprogram	Page
Rank-One Update of a General Matrix	SGER◂ CGERU◂ CGERC◂	DGER◂ ZGERU◂ ZGERC◂	308
Matrix-Vector Product for a Real Symmetric or Complex Hermitian Matrix	SSPMV◂ CHPMV◂ SSYMV◂ CHEMV◂ SSLMX§	DSPMV◂ ZHPMV◂ DSYMV◂ ZHEMV◂ DSLMX§	316
Rank-One Update of a Real Symmetric or Complex Hermitian Matrix	SSPR◂ CHPR◂ SSYR◂ CHER◂ SSLR1§	DSPR◂ ZHPR◂ DSYR◂ ZHER◂ DSLR1§	324
Rank-Two Update of a Real Symmetric or Complex Hermitian Matrix	SSPR2◂ CHPR2◂ SSYR2◂ CHER2◂ SSLR2§	DSPR2◂ ZHPR2◂ DSYR2◂ ZHER2◂ DSLR2§	332
Matrix-Vector Product for a General Band Matrix, Its Transpose, or Its Conjugate Transpose	SGBMV◂ CGBMV◂	DGBMV◂ ZGBMV◂	341
Matrix-Vector Product for a Real Symmetric or Complex Hermitian Band Matrix	SSBMV◂ CHBMV◂	DSBMV◂ ZHBMV◂	348
Matrix-Vector Product for a Triangular Matrix, Its Transpose, or Its Conjugate Transpose	STRMV◂ CTRMV◂ STPMV◂ CTPMV◂	DTRMV◂ ZTRMV◂ DTPMV◂ ZTPMV◂	353
Matrix-Vector Product for a Triangular Band Matrix, Its Transpose, or Its Conjugate Transpose	STBMV◂ CTBMV◂	DTBMV◂ ZTBMV◂	359

◂ Level 2 BLAS

§ These subroutines are provided only for migration from earlier releases of ESSL and are not intended for use in new programs.

Sparse Matrix-Vector Linear Algebra Subprograms

The sparse matrix-vector linear algebra subprograms operate on sparse matrices using optimized storage techniques; that is, only the nonzero elements of the vector are stored. These subprograms provide similar functions to the matrix-vector subprograms.

Table 35. List of Sparse Matrix-Vector Linear Algebra Subprograms		
Descriptive Name	**Long- Precision Subprogram**	**Page**
Matrix-Vector Product for a Sparse Matrix in Compressed-Matrix Storage Mode	DSMMX	366
Transpose a Sparse Matrix in Compressed-Matrix Storage Mode	DSMTM	369
Matrix-Vector Product for a Sparse Matrix or Its Transpose in Compressed-Diagonal Storage Mode	DSDMX	373

Use Considerations

If your program uses a sparse matrix stored by rows, as defined in "Storage-by-Rows" on page 99, you should first convert your sparse matrix to compressed-matrix storage mode by using the subroutine DSRSM on page 986. DSRSM converts a matrix to compressed-matrix storage mode. To convert your sparse matrix to compressed-diagonal storage mode, you need to perform this conversion in your application program before calling the ESSL subroutine.

Performance and Accuracy Considerations

1. In ESSL, the SSCAL and DSCAL subroutines provide the fastest way to zero out contiguous (stride 1) arrays, by specifying *incx* = 1 and $\alpha = 0$.
2. Where possible, use the matrix-vector linear algebra subprograms, rather than the vector-scalar, to optimize performance. Because data is presented in matrices rather than vectors, multiple operations can be performed by a single ESSL subprogram.
3. Where possible, use subprograms that do multiple computations, such as SNDOT and SNAXPY, rather than individual computations, such as SDOT and SAXPY. You get better performance.
4. Many of the short-precision subprograms provide increased accuracy by accumulating results in long precision. This is noted in the functional description of each subprogram.
5. In some of the subprograms, because implementation techniques vary to optimize performance, accuracy of the results may vary for different array sizes. In the subprograms in which this occurs, a general description of the implementation techniques is given in the functional description for each subprogram.
6. To select the sparse matrix subroutine that gives you the best performance, you must consider the layout of the data in your matrix. From this, you can determine the most efficient storage mode for your sparse matrix. ESSL provides two versions of each of its sparse matrix-vector subroutines that you can use. One operates on sparse matrices stored in compressed-matrix storage mode, and the other operates on sparse matrices stored in compressed-diagonal storage mode. These two storage modes are described in "Sparse Matrix" on page 92.

   Compressed-matrix storage mode is generally applicable. It should be used when each row of the matrix contains approximately the same number of nonzero elements. However, if the matrix has a special form—that is, where the nonzero elements are

concentrated along a few diagonals—compressed-diagonal storage mode gives improved performance.

7. There are some ESSL-specific rules that apply to the results of computations on the workstation processors using the ANSI/IEEE standards. For details, see "What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?" on page 44.

Vector-Scalar Subprograms

This section contains the vector-scalar subprogram descriptions.

ISAMAX, IDAMAX, ICAMAX, and IZAMAX—Position of the First or Last Occurrence of the Vector Element Having the Largest Magnitude

ISAMAX and IDAMAX find the position *i* of the first or last occurrence of a vector element having the maximum absolute value. ICAMAX and IZAMAX find the position *i* of the first or last occurrence of a vector element having the largest sum of the absolute values of the real and imaginary parts of the vector elements.

You get the position of the first or last occurrence of an element by specifying positive or negative stride, respectively, for vector ***x***. Regardless of the stride, the position *i* is always relative to the location specified in the calling sequence for vector ***x*** (in argument *x*).

Table 36. Data Types

x	**Subprogram**
Short-precision real	ISAMAX
Long-precision real	IDAMAX
Short-precision complex	ICAMAX
Long-precision complex	IZAMAX

Syntax

Fortran	ISAMAX | IDAMAX | ICAMAX | IZAMAX (*n*, *x*, *incx*)
C and C++	isamax | idamax | icamax | izamax (*n*, *x*, *incx*);
PL/I	ISAMAX | IDAMAX | ICAMAX | IZAMAX (*n*, *x*, *incx*);

On Entry

n
is the number of elements in vector ***x***. Specified as: a fullword integer; $n \geq 0$.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 36.

incx
is the stride for vector ***x***. Specified as: a fullword integer. It can have any value.

On Return

Function value

is the position *i* of the element in the array, where:

If $incx \geq 0$, *i* is the position of the first occurrence.

If $incx < 0$, *i* is the position of the last occurrence.

Returned as: a fullword integer; $0 \leq i \leq n$.

Note: Declare the ISAMAX, IDAMAX, ICAMAX, and IZAMAX functions in your program as returning a fullword integer value.

Function: ISAMAX and IDAMAX find the first element x_k, where k is defined as the smallest index k, such that:

$$|x_k| = \max\{|x_j| \text{ for } j = 1, n\}$$

ICAMAX and IZAMAX find the first element x_k, where k is defined as the smallest index k, such that:

$$|a_k|+|b_k| = \max\{|a_j|+|b_j| \text{ for } j = 1, n\}$$
where $x_k = (a_k, b_k)$

By specifying a positive or negative stride for vector ***x***, the first or last occurrence, respectively, is found in the array. The position *i*, returned as the value of the function, is always figured relative to the location specified in the calling sequence for vector ***x*** (in argument *x*). Therefore, depending on the stride specified for *incx*, *i* has the following values:

For $incx \geq 0$, $i = k$
For $incx < 0$, $i = n-k+1$

See reference [73]. The result is returned as a function value. If *n* is 0, then 0 is returned as the value of the function.

Error Conditions

Computational Errors: None

Input-Argument Errors: $n < 0$

Example 1: This example shows a vector, ***x***, with a stride of 1.

Function Reference and Input

```
               N   X   INCX
               |   |    |
IMAX = ISAMAX( 9 , X ,  1   )

X        =  (1.0, 2.0, 7.0, -8.0, -5.0, -10.0, -9.0, 10.0, 6.0)
```

Output

```
IMAX     =  6
```

Example 2: This example shows a vector, ***x***, with a stride greater than 1.

Function Reference and Input

```
               N   X   INCX
               |   |    |
IMAX = ISAMAX( 5 , X ,  2   )

X        =  (1.0, . , 7.0, . , -5.0, . , -9.0, . , 6.0)
```

Output

```
IMAX     =  4
```

Example 3: This example shows a vector, ***x***, with a stride of 0.

Function Reference and Input

```
               N   X   INCX
               |   |    |
IMAX = ISAMAX( 9 , X ,  0   )

X        =  (1.0, . , . , . , . , . , . , . , .)
```

Output

```
IMAX     =  1
```

Example 4: This example shows a vector, ***x***, with a negative stride. Processing begins at element `X(15)`, which is 2.0.

Function Reference and Input

```
               N   X   INCX
               |   |    |
IMAX = ISAMAX( 8 , X , -2   )

X        =  (3.0, . , 5.0, . , -8.0, . , 6.0, . , 8.0, . ,
             4.0, . , 8.0, . , 2.0)
```

Output

```
IMAX     =  7
```

Example 5: This example shows a vector, ***x***, containing complex numbers and having a stride of 1.

Function Reference and Input

```
               N   X   INCX
               |   |    |
IMAX = ICAMAX( 5 , X ,  1   )

X        =  ((9.0 , 2.0) , (7.0 , -8.0) , (-5.0 , -10.0) , (-4.0 , 10.0),
             (6.0 , 3.0))
```

Output

```
IMAX     =  2
```

ISAMIN and IDAMIN—Position of the First or Last Occurrence of the Vector Element Having Minimum Absolute Value

These subprograms find the position *i* of the first or last occurrence of a vector element having the minimum absolute value.

You get the position of the first or last occurrence of an element by specifying positive or negative stride, respectively, for vector ***x***. Regardless of the stride, the position *i* is always relative to the location specified in the calling sequence for vector ***x*** (in argument *x*).

Table 37. Data Types

x	**Subprogram**
Short-precision real	ISAMIN
Long-precision real	IDAMIN

Syntax

Fortran	ISAMIN | IDAMIN (*n*, *x*, *incx*)
C and C++	isamin | idamin (*n*, *x*, *incx*);
PL/I	ISAMIN | IDAMIN (*n*, *x*, *incx*);

On Entry

n
is the number of elements in vector ***x***. Specified as: a fullword integer; $n \geq 0$.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 37.

incx
is the stride for vector ***x***. Specified as: a fullword integer. It can have any value.

On Return

Function value

is the position *i* of the element in the array, where:

If $incx \geq 0$, *i* is the position of the first occurrence.

If $incx < 0$, *i* is the position of the last occurrence.

Returned as: a fullword integer; $0 \leq i \leq n$.

Note: Declare the ISAMIN and IDAMIN functions in your program as returning a fullword integer value.

Function: These subprograms find the first element x_k, where *k* is defined as the smallest index *k*, such that:

$$|x_k| = \min\{|x_j| \text{ for } j = 1, n\}$$

By specifying a positive or negative stride for vector ***x***, the first or last occurrence, respectively, is found in the array. The position *i*, returned as the value of the function, is always figured relative to the location specified in the calling sequence for vector ***x*** (in argument *x*). Therefore, depending on the stride specified for *incx*, *i* has the following values:

For $incx \geq 0$, $i = k$
For $incx < 0$, $i = n-k+1$

See reference [73]. The result is returned as a function value. If *n* is 0, then 0 is returned as the value of the function.

Error Conditions

Computational Errors: None

Input-Argument Errors: $n < 0$

Example 1: This example shows a vector, ***x***, with a stride of 1.

Function Reference and Input

```
               N   X   INCX
               |   |    |
IMIN = ISAMIN( 6 , X ,  1  )

X        =  (3.0, 4.0, 1.0, 8.0, 1.0, 3.0)
```

Output

```
IMIN     =  3
```

Example 2: This example shows a vector, ***x***, with a stride greater than 1.

Function Reference and Input

```
               N   X   INCX
               |   |    |
IMIN = ISAMIN( 4 , X ,  2  )

X        =  (-3.0, . , -9.0, . , -8.0, . , 3.0)
```

Output

```
IMIN     =  1
```

Example 3: This example shows a vector, ***x***, with a positive stride and two elements with the minimum absolute value. The position of the first occurrence is returned.

Function Reference and Input

```
               N   X   INCX
               |   |    |
IMIN = ISAMIN( 4 , X ,  2  )

X        =  (2.0, . , -1.0, . , 4.0, . , 1.0)
```

Output

```
IMIN     =  2
```

Example 4: This example shows a vector, ***x***, with a negative stride and two elements with the minimum absolute value. The position of the last occurrence is returned. Processing begins at element `X(7)`, which is 1.0.

Function Reference and Input

```
                N   X   INCX
                |   |    |
IMIN = ISAMIN( 4 , X , -2  )

X        =  (2.0, . , -1.0, . , 4.0, . , 1.0)
```

Output

```
IMIN     =  4
```

ISMAX and IDMAX—Position of the First or Last Occurrence of the Vector Element Having the Maximum Value

These subprograms find the position *i* of the first or last occurrence of a vector element having the maximum value.

You get the position of the first or last occurrence of an element by specifying positive or negative stride, respectively, for vector ***x***. Regardless of the stride, the position *i* is always relative to the location specified in the calling sequence for vector ***x*** (in argument *x*).

Table 38. Data Types

x	**Subprogram**
Short-precision real	ISMAX
Long-precision real	IDMAX

Syntax

Fortran	ISMAX | IDMAX (*n*, *x*, *incx*)
C and C++	ismax | idmax (*n*, *x*, *incx*);
PL/I	ISMAX | IDMAX (*n*, *x*, *incx*);

On Entry

n
is the number of elements in vector ***x***. Specified as: a fullword integer; $n \geq 0$.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 38.

incx
is the stride for vector ***x***. Specified as: a fullword integer. It can have any value.

On Return

Function value

is the position *i* of the element in the array, where:

If $incx \geq 0$, *i* is the position of the first occurrence.

If $incx < 0$, *i* is the position of the last occurrence.

Returned as: a fullword integer; $0 \leq i \leq n$.

Note: Declare the ISMAX and IDMAX functions in your program as returning a fullword integer value.

Function: These subprograms find the first element x_k, where *k* is defined as the smallest index *k*, such that:

$x_k = \max\{x_j \text{ for } j = 1, n\}$

By specifying a positive or negative stride for vector ***x***, the first or last occurrence, respectively, is found in the array. The position *i*, returned as the value of the function, is always figured relative to the location specified in the calling sequence for vector ***x*** (in argument *x*). Therefore, depending on the stride specified for *incx*, *i* has the following values:

For $incx \geq 0$, $i = k$
For $incx < 0$, $i = n-k+1$

See reference [73]. The result is returned as a function value. If *n* is 0, then 0 is returned as the value of the function.

Error Conditions

Computational Errors: None

Input-Argument Errors: $n < 0$

Example 1: This example shows a vector, ***x***, with a stride of 1.

Function Reference and Input

```
              N   X  INCX
              |   |   |
IMAX = ISMAX( 6 , X , 1  )

X        =  (3.0, 4.0, 1.0, 8.0, 1.0, 8.0)
```

Output

```
IMAX     =  4
```

Example 2: This example shows a vector, ***x***, with a stride greater than 1.

Function Reference and Input

```
              N   X  INCX
              |   |   |
IMAX = ISMAX( 4 , X , 2  )

X        =  (-3.0, . , 9.0, . , -8.0, . , 3.0)
```

Output

```
IMAX     =  2
```

Example 3: This example shows a vector, ***x***, with a positive stride and two elements with the maximum value. The position of the first occurrence is returned.

Function Reference and Input

```
              N   X  INCX
              |   |   |
IMAX = ISMAX( 4 , X , 2  )

X        =  (2.0, . , 4.0, . , 4.0, . , 1.0)
```

Output

```
IMAX     =  2
```

Example 4: This example shows a vector, ***x***, with a negative stride and two elements with the maximum value. The position of the last occurrence is returned. Processing begins at element `X(7)`, which is 1.0.

Function Reference and Input

```
              N   X   INCX
              |   |    |
IMAX = ISMAX( 4 , X , -2  )

X        =  (2.0, . , 4.0, . , 4.0, . , 1.0)
```

Output

```
IMAX     =  3
```

ISMIN and IDMIN—Position of the First or Last Occurrence of the Vector Element Having Minimum Value

These subprograms find the position *i* of the first or last occurrence of a vector element having the minimum value.

You get the position of the first or last occurrence of an element by specifying positive or negative stride, respectively, for vector ***x***. Regardless of the stride, the position *i* is always relative to the location specified in the calling sequence for vector ***x*** (in argument *x*).

Table 39. Data Types

x	**Subprogram**
Short-precision real	ISMIN
Long-precision real	IDMIN

Syntax

Fortran	ISMIN | IDMIN (*n*, *x*, *incx*)
C and C++	ismin | idmin (*n*, *x*, *incx*);
PL/I	ISMIN | IDMIN (*n*, *x*, *incx*);

On Entry

n
is the number of elements in vector ***x***. Specified as: a fullword integer; $n \geq 0$.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 39.

incx
is the stride for vector ***x***. Specified as: a fullword integer. It can have any value.

On Return

Function value

is the position *i* of the element in the array, where:

If $incx \geq 0$, *i* is the position of the first occurrence.

If $incx < 0$, *i* is the position of the last occurrence.

Returned as: a fullword integer; $0 \leq i \leq n$.

Note: Declare the ISMIN and IDMIN functions in your program as returning a fullword integer value.

Function: These subprograms find the first element x_k, where *k* is defined as the smallest index *k*, such that:

$x_k = \min\{x_j \text{ for } j = 1, n\}$

By specifying a positive or negative stride for vector ***x***, the first or last occurrence, respectively, is found in the array. The position *i*, returned as the value of the function, is always figured relative to the location specified in the calling sequence for vector ***x*** (in argument *x*). Therefore, depending on the stride specified for *incx*, *i* has the following values:

For $incx \geq 0$, $i = k$
For $incx < 0$, $i = n-k+1$

See reference [73]. The result is returned as a function value. If *n* is 0, then 0 is returned as the value of the function.

Error Conditions

Computational Errors: None

Input-Argument Errors: $n < 0$

Example 1: This example shows a vector, ***x***, with a stride of 1.

Function Reference and Input

```
              N   X  INCX
              |   |   |
IMIN = ISMIN( 6 , X , 1  )

X        =  (3.0, 4.0, 1.0, 8.0, 1.0, 3.0)
```

Output

```
IMIN     =  3
```

Example 2: This example shows a vector, ***x***, with a stride greater than 1.

Function Reference and Input

```
              N   X  INCX
              |   |   |
IMIN = ISMIN( 4 , X , 2  )

X        =  (-3.0, . , -9.0, . , -8.0, . , 3.0)
```

Output

```
IMIN     =  2
```

Example 3: This example shows a vector, ***x***, with a positive stride and two elements with the minimum value. The position of the first occurrence is returned. Processing begins at element `X(7)`, which is 1.0.

Function Reference and Input

```
              N   X  INCX
              |   |   |
IMIN = ISMIN( 4 , X , 2  )

X        =  (2.0, . , 1.0, . , 4.0, . , 1.0)
```

Output

```
IMIN     =  2
```

Example 4: This example shows a vector, ***x***, with a negative stride and two elements with the minimum value. The position of the last occurrence is returned. Processing begins at element X(7), which is 1.0.

Function Reference and Input

```
              N   X   INCX
              |   |    |
IMIN = ISMIN( 4 , X , -2  )

X        =  (2.0, . , 1.0, . , 4.0, . , 1.0)
```

Output

```
IMIN     =  4
```

SASUM, DASUM, SCASUM, and DZASUM—Sum of the Magnitudes of the Elements in a Vector

SASUM and DASUM compute the sum of the absolute values of the elements in vector ***x***. SCASUM and DZASUM compute the sum of the absolute values of the real and imaginary parts of the elements in vector ***x***.

Table 40. Data Types

x	**Result**	**Subprogram**
Short-precision real	Short-precision real	SASUM
Long-precision real	Long-precision real	DASUM
Short-precision complex	Short-precision real	SCASUM
Long-precision complex	Long-precision real	DZASUM

Syntax

Fortran	SASUM | DASUM | SCASUM | DZASUM (*n*, *x*, *incx*)
C and C++	sasum | dasum | scasum | dzasum (*n*, *x*, *incx*);
PL/I	SASUM | DASUM | SCASUM | DZASUM (*n*, *x*, *incx*);

On Entry

n
is the number of elements in vector ***x***. Specified as: a fullword integer; $n \geq 0$.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 40.

incx
is the stride for vector ***x***. Specified as: a fullword integer. It can have any value.

On Return

Function value
is the result of the summation. Returned as: a number of the data type indicated in Table 40.

Note: Declare this function in your program as returning a value of the type indicated in Table 40.

Function: SASUM and DASUM compute the sum of the absolute values of the elements of ***x***, which is expressed as follows:

$$\sum_{i=1}^{n} |x_i| = |x_1| + |x_2| + \ldots + |x_n|$$

SCASUM and DZASUM compute the sum of the absolute values of the real and imaginary parts of the elements of ***x***, which is expressed as follows:

$$\sum_{i=1}^{n}\left(|a_i| + |b_i|\right) = \left(|a_1| + |b_1|\right) + \left(|a_2| + |b_2|\right) + \ldots + \left(|a_n| + |b_n|\right)$$
$$\text{where } x_i = (a_i, b_i)$$

See reference [73]. The result is returned as a function value. If *n* is 0, then 0.0 is returned as the value of the function. For SASUM and SCASUM, intermediate results are accumulated in long precision.

Error Conditions

Computational Errors: None

Input-Argument Errors: $n < 0$

Example 1: This example shows a vector, ***x***, with a stride of 1.

Function Reference and Input

```
              N   X  INCX
              |   |   |
SUMM = SASUM( 7 , X , 1  )

X        =  (1.0, -3.0, -6.0, 7.0, 5.0, 2.0, -4.0)
```

Output

```
SUMM     =  28.0
```

Example 2: This example shows a vector, ***x***, with a stride greater than 1.

Function Reference and Input

```
              N   X  INCX
              |   |   |
SUMM = SASUM( 4 , X , 2  )

X        =  (1.0, . , -6.0, . , 5.0, . , -4.0)
```

Output

```
SUMM     =  16.0
```

Example 3: This example shows a vector, ***x***, with negative stride. Processing begins at element X(7), which is –4.0.

Function Reference and Input

```
              N   X   INCX
              |   |    |
SUMM = SASUM( 4 , X , -2  )

X        =  (1.0, . , -6.0, . , 5.0, . , -4.0)
```

Output

```
SUMM     =  16.0
```

Example 4: This example shows a vector, ***x***, with a stride of 0. The result in SUMM is nx_1.

Function Reference and Input

```
              N   X  INCX
              |   |   |
SUMM = SASUM( 7 , X , 0  )

X        =  (-2.0, . , . , . , . , . , .)
```

Output

```
SUMM     =  14.0
```

Example 5: This example shows a vector, ***x***, containing complex numbers and having a stride of 1.

Function Reference and Input

```
               N   X  INCX
               |   |   |
SUMM = SCASUM( 5 , X , 1  )

X        =  ((1.0, 2.0), (-3.0, 4.0), (5.0, -6.0 ), (-7.0, -8.0),
             (9.0, 10.0))
```

Output

```
SUMM     =  55.0
```

SAXPY, DAXPY, CAXPY, and ZAXPY—Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in the Vector Y

These subprograms perform the following computation, using the scalar α and vectors $\boldsymbol{x}$ and $\boldsymbol{y}$:

$$\boldsymbol{y} \leftarrow \boldsymbol{y}+\alpha\boldsymbol{x}$$

Table 41. Data Types

alpha, ***x***, ***y***	**Subprogram**
Short-precision real	SAXPY
Long-precision real	DAXPY
Short-precision complex	CAXPY
Long-precision complex	ZAXPY

Syntax

Fortran	CALL SAXPY | DAXPY | CAXPY | ZAXPY (*n*, *alpha*, *x*, *incx*, *y*, *incy*)
C and C++	saxpy | daxpy | caxpy | zaxpy (*n*, *alpha*, *x*, *incx*, *y*, *incy*);
PL/I	CALL SAXPY | DAXPY | CAXPY | ZAXPY (*n*, *alpha*, *x*, *incx*, *y*, *incy*);

On Entry

n
is the number of elements in vectors ***x*** and ***y***. Specified as: a fullword integer; $n \geq 0$.

alpha
is the scalar *alpha*. Specified as: a number of the data type indicated in Table 41.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 41.

incx
is the stride for vector ***x***. Specified as: a fullword integer. It can have any value.

y
is the vector ***y*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incy|$, containing numbers of the data type indicated in Table 41.

incy
is the stride for vector ***y***. Specified as: a fullword integer. It can have any value.

On Return

y
is the vector ***y***, containing the results of the computation $\boldsymbol{y}+\alpha\boldsymbol{x}$. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 41.

Notes

1. If you specify the same vector for ***x*** and ***y***, *incx* and *incy* must be equal; otherwise, results are unpredictable.

2. If you specify different vectors for ***x*** and ***y***, they must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.

Function: The computation is expressed as follows:

$$\begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix} \leftarrow \begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix} + \alpha \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix}$$

See reference [73]. If *alpha* or *n* is zero, no computation is performed. For CAXPY, intermediate results are accumulated in long precision.

Error Conditions

Computational Errors: None

Input-Argument Errors: $n < 0$

Example 1: This example shows vectors ***x*** and ***y*** with positive strides.

Call Statement and Input

```
            N  ALPHA  X  INCX  Y  INCY
            |    |    |   |    |   |
CALL SAXPY( 5 , 2.0 , X , 1  , Y , 2  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (1.0, . , 1.0, . , 1.0, . , 1.0, . , 1.0)
```

Output

```
Y        =  (3.0, . , 5.0, . , 7.0, . , 9.0, . , 11.0)
```

Example 2: This example shows vectors ***x*** and ***y*** having strides of opposite signs. For ***y***, which has negative stride, processing begins at element `Y(5)`, which is 1.0.

Call Statement and Input

```
            N  ALPHA  X  INCX  Y   INCY
            |    |    |   |    |    |
CALL SAXPY( 5 , 2.0 , X , 1  , Y , -1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output

```
Y        =  (15.0, 12.0, 9.0, 6.0, 3.0)
```

Example 3: This example shows a vector, ***x***, with 0 stride. Vector ***x*** is treated like a vector of length *n*, all of whose elements are the same as the single element in ***x***.

Call Statement and Input

```
            N  ALPHA  X  INCX  Y  INCY
            |    |    |   |    |   |
CALL SAXPY( 5 , 2.0 , X , 0  , Y , 1  )

X        =  (1.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output

```
Y        =  (7.0, 6.0, 5.0, 4.0, 3.0)
```

Example 4: This example shows how SAXPY can be used to compute a scalar value. In this case, vectors ***x*** and ***y*** contain scalar values and the strides for both vectors are 0. The number of elements to be processed, *n*, is 1.

Call Statement and Input

```
            N  ALPHA  X  INCX  Y  INCY
            |    |    |   |    |   |
CALL SAXPY( 1 , 2.0 , X , 0  , Y , 0  )

X        =  (1.0)
Y        =  (5.0)
```

Output

```
Y        =  (7.0)
```

Example 5: This example shows how to use CAXPY, where vectors ***x*** and ***y*** contain complex numbers. In this case, vectors ***x*** and ***y*** have positive strides.

Call Statement and Input

```
            N  ALPHA  X  INCX  Y  INCY
            |    |    |   |    |   |
CALL CAXPY( 3 ,ALPHA, X , 1  , Y , 2  )

ALPHA    =  (2.0, 3.0)
X        =  ((1.0, 2.0), (2.0, 0.0), (3.0, 5.0))
Y        =  ((1.0, 1.0 ), . , (0.0, 2.0), . , (5.0, 4.0))
```

Output

```
Y        =  ((-3.0, 8.0), . , (4.0, 8.0), . , (-4.0, 23.0))
```

SCOPY, DCOPY, CCOPY, and ZCOPY—Copy a Vector

These subprograms copy vector ***x*** to another vector, ***y***:

$$y \leftarrow x$$

Table 42. Data Types

x, ***y***	**Subprogram**
Short-precision real	SCOPY
Long-precision real	DCOPY
Short-precision complex	CCOPY
Long-precision complex	ZCOPY

Syntax

Fortran	CALL SCOPY | DCOPY | CCOPY | ZCOPY (*n*, *x*, *incx*, *y*, *incy*)
C and C++	scopy | dcopy | ccopy | zcopy (*n*, *x*, *incx*, *y*, *incy*);
PL/I	CALL SCOPY | DCOPY | CCOPY | ZCOPY (*n*, *x*, *incx*, *y*, *incy*);

On Entry

n
is the number of elements in vectors ***x*** and ***y***. Specified as: a fullword integer; $n \geq 0$.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 42.

incx
is the stride for vector ***x***. Specified as: a fullword integer. It can have any value.

y
See On Return.

incy
is the stride for vector ***y.*** Specified as: a fullword integer. It can have any value.

On Return

y
is the vector ***y*** of length *n*. Returned as: a one-dimensional array of (at least) length $1+(n-1)|incy|$, containing numbers of the data type indicated in Table 42.

Notes

1. If you specify the same vector for ***x*** and ***y***, *incx* and *incy* must be equal; otherwise, results are unpredictable.
2. If you specify different vectors for ***x*** and ***y***, they must have no common elements; otherwise, results are unpredictable. See “Concepts” on page 55.

Function: The copy is expressed as follows:

$$\begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix} \leftarrow \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix}$$

See reference [73]. If n is 0, no copy is performed.

Error Conditions

Computational Errors: None

Input-Argument Errors: $n < 0$

Example 1: This example shows input vector ***x*** and output vector ***y*** with positive strides.

Call Statement and Input

```
            N   X  INCX  Y  INCY
            |   |   |    |   |
CALL SCOPY( 5 , X , 1  , Y , 2  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
```

Output

```
Y        =  (1.0, . , 2.0, . , 3.0, . , 4.0, . , 5.0)
```

Example 2: This example shows how to obtain a reverse copy of the input vector ***x*** by specifying strides with the same absolute value, but with opposite signs, for input vector ***x*** and output vector ***y***. For ***y***, which has a negative stride, results are stored beginning at element `Y(5)`.

Call Statement and Input

```
            N   X  INCX  Y   INCY
            |   |   |    |    |
CALL SCOPY( 5 , X , 1  , Y , -1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
```

Output

```
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Example 3: This example shows an input vector, ***x***, with 0 stride. Vector ***x*** is treated like a vector of length *n*, all of whose elements are the same as the single element in ***x***. This is a technique for replicating an element of a vector.

Call Statement and Input

```
            N   X  INCX  Y  INCY
            |   |   |    |   |
CALL SCOPY( 5 , X , 0  , Y , 1  )

X        =  (13.0)
```

Output

```
Y        =  (13.0, 13.0, 13.0, 13.0, 13.0)
```

Example 4: This example shows input vector ***x*** and output vector ***y***, containing complex numbers and having positive strides.

Call Statement and Input

```
            N   X  INCX  Y  INCY
            |   |   |    |   |
CALL CCOPY( 4 , X , 1  , Y , 2  )

X        =  ((1.0, 1.0), (2.0, 2.0), (3.0, 3.0), (4.0, 4.0))
```

Output

```
Y        =  ((1.0, 1.0), . , (2.0, 2.0), . , (3.0, 3.0), . ,
             (4.0, 4.0))
```

SDOT, DDOT, CDOTU, ZDOTU, CDOTC, and ZDOTC—Dot Product of Two Vectors

SDOT, DDOT, CDOTU, and ZDOTU compute the dot product of vectors ***x*** and ***y***:

$$x \bullet y$$

CDOTC and ZDOTC compute the dot product of the complex conjugate of vector ***x*** with vector ***y***:

$$\bar{x} \bullet y$$

Table 43. Data Types

x, ***y***, **Result**	**Subprogram**
Short-precision real	SDOT
Long-precision real	DDOT
Short-precision complex	CDOTU and CDOTC
Long-precision complex	ZDOTU and ZDOTC

Syntax

Fortran	SDOT | DDOT | CDOTU | ZDOTU | CDOTC | ZDOTC (*n*, *x*, *incx*, *y*, *incy*)
C and C++	sdot | ddot | cdotu | zdotu | cdotc | zdotc (*n*, *x*, *incx*, *y*, *incy*);
PL/I	SDOT | DDOT | CDOTU | ZDOTU | CDOTC | ZDOTC (*n*, *x*, *incx*, *y*, *incy*);

On Entry

n
is the number of elements in vectors ***x*** and ***y***. Specified as: a fullword integer; $n \geq 0$.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 43.

incx
is the stride for vector ***x***. Specified as: a fullword integer. It can have any value.

y
is the vector ***y*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incy|$, containing numbers of the data type indicated in Table 43.

incy
is the stride for vector ***y***. Specified as: a fullword integer. It can have any value.

On Return

Function value
is the result of the dot product computation. Returned as: a number of the data type indicated in Table 43.

Note: Declare this function in your program as returning a value of the data type indicated in Table 43.

Function: SDOT, DDOT, CDOTU, and ZDOTU compute the dot product of the vectors ***x*** and ***y***, which is expressed as follows:

$$x \bullet y = x_1y_1 + x_2y_2 + \ldots + x_ny_n$$

CDOTC and ZDOTC compute the dot product of the complex conjugate of vector ***x*** with vector ***y***, which is expressed as follows:

$$\bar{x} \bullet y = \bar{x}_1y_1 + \bar{x}_2y_2 + \ldots + \bar{x}_ny_n$$

See reference [73]. The result is returned as a function value. If *n* is 0, then zero is returned as the value of the function.

For SDOT, CDOTU, and CDOTC, intermediate results are accumulated in long precision.

Error Conditions

Computational Errors: None

Input-Argument Errors: $n < 0$

Example 1: This example shows how to compute the dot product of two vectors, ***x*** and ***y***, having strides of 1.

Function Reference and Input

```
             N   X  INCX  Y  INCY
             |   |   |    |   |
DOTT = SDOT( 5 , X , 1  , Y , 1  )

X        =  (1.0, 2.0, -3.0, 4.0, 5.0)
Y        =  (9.0, 8.0, 7.0, -6.0, 5.0)
```

Output

```
DOTT     =  (9.0 + 16.0 - 21.0 - 24.0 + 25.0) = 5.0
```

Example 2: This example shows how to compute the dot product of a vector, ***x***, with a stride of 1, and a vector, ***y***, with a stride greater than 1.

Function Reference and Input

```
             N   X  INCX  Y  INCY
             |   |   |    |   |
DOTT = SDOT( 5 , X , 1  , Y , 2  )

X        =  (1.0, 2.0, -3.0, 4.0, 5.0)
Y        =  (9.0, . , 7.0, . , 5.0, . , -3.0, . , 1.0)
```

Output

```
DOTT     =  (9.0 + 14.0 - 15.0 - 12.0 + 5.0) = 1.0
```

Example 3: This example shows how to compute the dot product of a vector, ***x***, with a negative stride, and a vector, ***y***, with a stride greater than 1. For ***x***, processing begins at element `X(5)`, which is 5.0.

Function Reference and Input

```
             N   X   INCX  Y  INCY
             |   |    |    |   |
DOTT = SDOT( 5 , X , -1  , Y , 2  )

X        =  (1.0, 2.0, -3.0, 4.0, 5.0)
Y        =  (9.0, . , 7.0, . , 5.0, . , -3.0, . , 1.0)
```

Output

```
DOTT     =  (45.0 + 28.0 - 15.0 - 6.0 + 1.0) = 53.0
```

Example 4: This example shows how to compute the dot product of a vector, ***x***, with a stride of 0, and a vector, ***y***, with a stride of 1. The result in `DOTT` is $x_1(y_1+...+y_n)$.

Function Reference and Input

```
             N   X  INCX  Y  INCY
             |   |   |    |   |
DOTT = SDOT( 5 , X , 0  , Y , 1  )

X        =  (1.0, . , . , . , .)
Y        =  (9.0, 8.0, 7.0, -6.0, 5.0)
```

Output

```
DOTT     =  (1.0) × (9.0 + 8.0 + 7.0 - 6.0 + 5.0) = 23.0
```

Example 5: This example shows how to compute the dot product of two vectors, ***x*** and ***y***, with strides of 0. The result in `DOTT` is nx_1y_1.

Function Reference and Input

```
             N   X  INCX  Y  INCY
             |   |   |    |   |
DOTT = SDOT( 5 , X , 0  , Y , 0  )

X        =  (1.0, . , . , . , .)
Y        =  (9.0, . , . , . , .)
```

Output

```
DOTT     =  (5) × (1.0) × (9.0) = 45.0
```

Example 6: This example shows how to compute the dot product of two vectors, ***x*** and ***y***, containing complex numbers, where ***x*** has a stride of 1, and ***y*** has a stride greater than 1.

Function Reference and Input

```
              N   X  INCX  Y  INCY
              |   |   |    |   |
DOTT = CDOTU( 3 , X , 1  , Y , 2  )

X        =  ((1.0, 2.0), (3.0, -4.0), (-5.0, 6.0))
Y        =  ((10.0, 9.0), . , (-6.0, 5.0), . , (2.0, 1.0))
```

Output

```
DOTT     =  ((10.0 - 18.0 - 10.0) - (18.0 - 20.0 + 6.0),
             (9.0 + 15.0 - 5.0) + (20.0 + 24.0 + 12.0))
         =  (-22.0, 75.0)
```

Example 7: This example shows how to compute the dot product of the conjugate of a vector, ***x***, with vector ***y***, both containing complex numbers, where ***x*** has a stride of 1, and ***y*** has a stride greater than 1.

Function Reference and Input

```
              N   X  INCX  Y  INCY
              |   |   |    |   |
DOTT = CDOTC( 3 , X , 1  , Y , 2  )

X        =  ((1.0, 2.0), (3.0, -4.0), (-5.0, 6.0))
Y        =  ((10.0, 9.0), . , (-6.0, 5.0), . , (2.0, 1.0))
```

Output

```
DOTT     =  ((10.0 - 18.0 - 10.0) + (18.0 - 20.0 + 6.0),
             (9.0  +  15.0  -  5.0)  -  (20.0  +  24.0  +  12.0))
         =  (-14.0, -37.0)
```

SNAXPY and DNAXPY—Compute SAXPY or DAXPY N Times

These subprograms compute SAXPY or DAXPY, respectively, *n* times:

$$\boldsymbol{y}_i \leftarrow \boldsymbol{y}_i + \alpha_i \boldsymbol{x}_i \qquad \text{for } i = 1, n$$

where each α_i is a scalar value, contained in the vector $\boldsymbol{a}$, and each $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$ are vectors, contained in vectors (or matrices) $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively. For an explanation of the SAXPY and DAXPY computations, see "SAXPY, DAXPY, CAXPY, and ZAXPY—Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in the Vector Y" on page 217.

Table 44. Data Types

a, ***x***, ***y***	**Subprogram**
Short-precision real	SNAXPY
Long-precision real	DNAXPY

Syntax

Fortran	CALL SNAXPY | DNAXPY (*n*, *m*, *a*, *inca*, *x*, *incxi*, *incxo*, *y*, *incyi*, *incyo*)
C and C++	snaxpy | dnaxpy (*n*, *m*, *a*, *inca*, *x*, *incxi*, *incxo*, *y*, *incyi*, *incyo*);
PL/I	CALL SNAXPY | DNAXPY (*n*, *m*, *a*, *inca*, *x*, *incxi*, *incxo*, *y*, *incyi*, *incyo*);

On Entry

n
is the number of SAXPY or DAXPY computations to be performed and the number of elements in vector $\boldsymbol{a}$. Specified as: a fullword integer; $n \geq 0$.

m
is the number of elements in vectors $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$ for each SAXPY or DAXPY computation. Specified as: a fullword integer; $m \geq 0$.

a
is the vector $\boldsymbol{a}$ of length *n*, containing the scalar values α_i, used in each computation of $\boldsymbol{y}_i + \alpha_i \boldsymbol{x}_i$. Specified as: a one-dimensional array of (at least) length $1+(n-1)|inca|$, containing numbers of the data type indicated in Table 44.

inca
is the stride for vector $\boldsymbol{a}$. Specified as: a fullword integer. It can have any value.

x
is the vector (or matrix) $\boldsymbol{x}$, containing the $\boldsymbol{x}_i$ vectors of length *m*, used in the *n* computations of $\boldsymbol{y}_i + \alpha_i \boldsymbol{x}_i$. Specified as: a one- or two-dimensional array of (at least) length $(1+(n-1)(incxo)) + (m-1)|incxi|$, containing numbers of the data type indicated in Table 44.

incxi
is the stride for $\boldsymbol{x}$ in the inner loop—that is, the stride identifying the elements in each vector $\boldsymbol{x}_i$. Specified as: a fullword integer. It can have any value.

incxo
is the stride for $\boldsymbol{x}$ in the outer loop—that is, the stride identifying each vector $\boldsymbol{x}_i$ in $\boldsymbol{x}$. Specified as: a fullword integer; $incxo \geq 0$.

y
is the vector (or matrix) $\boldsymbol{y}$, containing the $\boldsymbol{y}_i$ vectors of length *m*, used in the *n* computations of $\boldsymbol{y}_i + \alpha_i \boldsymbol{x}_i$. Specified as: a one- or two-dimensional array of (at least)

length (1+(*n*−1)(*incyo*)) + (*m*−1)|*incyi*|, containing numbers of the data type indicated in Table 44.

incyi
is the stride for ***y*** in the inner loop—that is, the stride identifying the elements in each vector $\boldsymbol{y}_i$ in ***y***. Specified as: a fullword integer; *incyi* > 0 or *incyi* < 0.

incyo
is the stride for ***y*** in the outer loop—that is, the stride identifying each vector $\boldsymbol{y}_i$ in ***y***. Specified as: a fullword integer; *incyo* ≥ 0.

On Return

y
is the vector (or matrix) ***y***, containing the $\boldsymbol{y}_i$ vectors of length *m*, which contain the results of the *n* SAXPY or DAXPY computations, $\boldsymbol{y}_i + \alpha_i\boldsymbol{x}_i$ for $i = 1, n$. Returned as: a one- or two-dimensional array, containing numbers of the data type indicated in Table 44 on page 227.

Note: Vector ***y*** must have no common elements with vector ***a*** or vector ***x***; otherwise, results are unpredictable. See "Concepts" on page 55.

Function: The SAXPY or DAXPY computations:

$$\boldsymbol{y} \leftarrow \boldsymbol{y} + \alpha\boldsymbol{x}$$

are performed *n* times. This is expressed as follows:

$$\boldsymbol{y}_i \leftarrow \boldsymbol{y}_i + \alpha_i\boldsymbol{x}_i \qquad \text{for } i = 1, n$$

where each α_i is a scalar value, contained in the vector ***a***, and each $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$ are vectors, contained in vectors (or matrices) ***x*** and ***y***, respectively.

Each computation of SAXPY or DAXPY on page 217 uses the length of the $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$ vectors, *m*, for its input argument, *n*. It also uses the strides for the inner loop, *incxi* and *incyi*, for its parameters *incx* and *incy*, respectively. See "Function" on page 218 for a description of how the computation is done.

The outer loop of the SNAXPY or DNAXPY computation uses the strides of *inca*, *incxo*, and *incyo* to locate the elements in ***a*** and vectors in ***x*** and ***y*** for each *i*-th computation. These are:

For $i = 1, n$:

$\alpha_{((i-1)inca+1)}$	for *inca* ≥ 0
$\alpha_{((i-n)inca+1)}$	for *inca* < 0
$x_{((i-1)incxo+1)}$	
$y_{((i-1)incyo+1)}$	

If *m* or *n* is 0, no computation is performed.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. *n* < 0
2. *m* < 0

3. *incxo* < 0
4. *incyi* = 0
5. *incyo* < 0

Example 1: This example shows vectors, contained in matrices, with the stride of the inner loops *incxi* and *incyi* equal to 1.

Call Statement and Input

```
             N   M   A  INCA  X  INCXI INCXO  Y  INCYI INCYO
             |   |   |   |    |    |     |    |    |     |
CALL SNAXPY( 3 , 4 , A , 1  , X ,  1  , 10  , Y ,  1  ,  5  )

A        =  (3.0, 2.0, 4.0)

         ⎡                ⎤
         | 1.0  4.0  3.0  |
         | 2.0  3.0  4.0  |
         | 3.0  2.0  2.0  |
         | 4.0  1.0  1.0  |
X    =   |  .    .    .   |
         |  .    .    .   |
         |  .    .    .   |
         |  .    .    .   |
         |  .    .    .   |
         |  .    .    .   |
         ⎣                ⎦

         ⎡                ⎤
         | 4.0  1.0  3.0  |
         | 3.0  2.0  4.0  |
Y    =   | 2.0  3.0  2.0  |
         | 1.0  4.0  1.0  |
         |  .    .    .   |
         ⎣                ⎦
```

Output

```
         ⎡                 ⎤
         |  7.0  9.0  15.0  |
         |  9.0  8.0  20.0  |
Y    =   | 11.0  7.0  10.0  |
         | 13.0  6.0   5.0  |
         |   .    .     .   |
         ⎣                 ⎦
```

Example 2: This example shows vectors, contained in matrices, with a stride of the inner loop *incxi* greater than 1.

Call Statement and Input

```
             N   M   A  INCA  X  INCXI INCXO  Y  INCYI INCYO
             |   |   |   |    |    |     |    |    |     |
CALL SNAXPY( 3 , 4 , A , 1  , X ,  2  , 10  , Y ,  1  ,  5  )

A        =  (3.0, 2.0, 4.0)
```

```
        ⎡                ⎤
        ⎢ 1.0  4.0  3.0  ⎥
        ⎢  .    .    .   ⎥
        ⎢ 2.0  3.0  4.0  ⎥
        ⎢  .    .    .   ⎥
X    =  ⎢ 3.0  2.0  2.0  ⎥
        ⎢  .    .    .   ⎥
        ⎢ 4.0  1.0  1.0  ⎥
        ⎢  .    .    .   ⎥
        ⎢  .    .    .   ⎥
        ⎣  .    .    .   ⎦

        ⎡                ⎤
        ⎢ 4.0  1.0  3.0  ⎥
        ⎢ 3.0  2.0  4.0  ⎥
Y    =  ⎢ 2.0  3.0  2.0  ⎥
        ⎢ 1.0  4.0  1.0  ⎥
        ⎣  .    .    .   ⎦
```

Output

```
Y          =(same as output Y in Example 1)
```

Example 3: This example shows vectors, contained in matrices, with a negative stride, *incyi*, for the inner loop.

Call Statement and Input

```
             N   M   A  INCA  X  INCXI INCXO  Y  INCYI INCYO
             |   |   |   |    |    |     |    |    |     |
CALL SNAXPY( 3 , 4 , A , 1  , X ,  1  , 10  , Y , -1  ,  5  )

A        =  (3.0, 2.0, 4.0)

        ⎡                ⎤
        ⎢ 1.0  4.0  3.0  ⎥
        ⎢ 2.0  3.0  4.0  ⎥
        ⎢ 3.0  2.0  2.0  ⎥
        ⎢ 4.0  1.0  1.0  ⎥
X    =  ⎢  .    .    .   ⎥
        ⎢  .    .    .   ⎥
        ⎢  .    .    .   ⎥
        ⎢  .    .    .   ⎥
        ⎢  .    .    .   ⎥
        ⎣  .    .    .   ⎦

        ⎡                ⎤
        ⎢ 1.0  4.0  1.0  ⎥
        ⎢ 2.0  3.0  2.0  ⎥
Y    =  ⎢ 3.0  2.0  4.0  ⎥
        ⎢ 4.0  1.0  3.0  ⎥
        ⎣  .    .    .   ⎦
```

Output

```
        ⎡                    ⎤
        ⎢ 13.0  6.0   5.0 ⎥
        ⎢ 11.0  7.0  10.0 ⎥
Y    =  ⎢  9.0  8.0  20.0 ⎥
        ⎢  7.0  9.0  15.0 ⎥
        ⎢   .    .     .  ⎥
        ⎣                    ⎦
```

Example 4: This example shows vectors, contained in matrices, with a negative stride, *inca*, for vector ***a***. For vector ***a***, processing begins at element `A(5)`, which is 3.0.

Call Statement and Input

```
             N   M   A   INCA  X  INCXI INCXO  Y  INCYI INCYO
             |   |   |    |    |    |     |    |    |     |
CALL SNAXPY( 3 , 4 , A , -2  , X ,  1  , 10  , Y ,  1  ,  5  )

A        =  (4.0, . , 2.0, . , 3.0)

        ⎡                  ⎤
        ⎢ 1.0  4.0  3.0 ⎥
        ⎢ 2.0  3.0  4.0 ⎥
        ⎢ 3.0  2.0  2.0 ⎥
        ⎢ 4.0  1.0  1.0 ⎥
X    =  ⎢  .    .    .  ⎥
        ⎢  .    .    .  ⎥
        ⎢  .    .    .  ⎥
        ⎢  .    .    .  ⎥
        ⎢  .    .    .  ⎥
        ⎢  .    .    .  ⎥
        ⎣                  ⎦

        ⎡                  ⎤
        ⎢ 4.0  1.0  3.0 ⎥
        ⎢ 3.0  2.0  4.0 ⎥
Y    =  ⎢ 2.0  3.0  2.0 ⎥
        ⎢ 1.0  4.0  1.0 ⎥
        ⎢  .    .    .  ⎥
        ⎣                  ⎦
```

Output

```
Y          =(same as output Y in Example 1)
```

SNDOT and DNDOT—Compute Special Dot Products N Times

These subprograms compute one of the following special dot products n times:

$s_i \leftarrow \boldsymbol{x}_i \bullet \boldsymbol{y}_i$	Store positive dot product
$s_i \leftarrow -\boldsymbol{x}_i \bullet \boldsymbol{y}_i$	Store negative dot product
$s_i \leftarrow s_i + \boldsymbol{x}_i \bullet \boldsymbol{y}_i$	Accumulate positive dot product
$s_i \leftarrow s_i - \boldsymbol{x}_i \bullet \boldsymbol{y}_i$	Accumulate negative dot product

for $i = 1, n$

where each s_i is an element in vector $\boldsymbol{s}$, and each $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$ are vectors contained in vectors (or matrices) $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively.

Table 45. Data Types

$\boldsymbol{s}$, $\boldsymbol{x}$, $\boldsymbol{y}$	**Subprogram**
Short-precision real	SNDOT
Long-precision real	DNDOT

Syntax

Fortran	CALL SNDOT | DNDOT (*n*, *m*, *s*, *incs*, *isw*, *x*, *incxi*, *incxo*, *y*, *incyi*, *incyo*)
C and C++	sndot | dndot (*n*, *m*, *s*, *incs*, *isw*, *x*, *incxi*, *incxo*, *y*, *incyi*, *incyo*);
PL/I	CALL SNDOT | DNDOT (*n*, *m*, *s*, *incs*, *isw*, *x*, *incxi*, *incxo*, *y*, *incyi*, *incyo*);

On Entry

n
is the number of dot product computations to be performed and the number of elements in the vector $\boldsymbol{s}$. Specified as: a fullword integer; $n \geq 0$.

m
is the number of elements in vectors $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$ for each dot product computation. Specified as: a fullword integer; $m \geq 0$.

s
is the vector $\boldsymbol{s}$, containing the n scalar values s_i, where: If $isw = 1$ or 2, s_i is not used in the computation (no input value specified.)

If $isw = 3$ or 4, s_i is used in the computation (input value specified.)

Specified as: a one-dimensional array of (at least) length $1+(n-1)|incs|$, containing numbers of the data type indicated in Table 45.

incs
is the stride for vector $\boldsymbol{s}$. Specified as: a fullword integer; $incs > 0$ or $incs < 0$.

isw
indicates the type of computation to perform, depending on the value specified:

If $isw = 1$, $s_i \leftarrow \boldsymbol{x}_i \bullet \boldsymbol{y}_i$

If $isw = 2$, $s_i \leftarrow -\boldsymbol{x}_i \bullet \boldsymbol{y}_i$

If $isw = 3$, $s_i \leftarrow s_i + \boldsymbol{x}_i \bullet \boldsymbol{y}_i$

If $isw = 4$, $s_i \leftarrow s_i - \boldsymbol{x}_i \bullet \boldsymbol{y}_i$
where $i = 1, n$

Specified as: a fullword integer. Its value must be 1, 2, 3, or 4.

x
is the vector (or matrix) $\boldsymbol{x}$, containing the $\boldsymbol{x}_i$ vectors of length m, used in the n dot product computations. Specified as: a one- or two-dimensional array of (at least) length $(1+(n-1)(incxo))+(m-1)|incxi|$, containing numbers of the data type indicated in Table 45 on page 232.

incxi
is the stride for $\boldsymbol{x}$ in the inner loop—that is, the stride identifying the elements in each vector $\boldsymbol{x}_i$. Specified as: a fullword integer. It can have any value.

incxo
is the stride for $\boldsymbol{x}$ in the outer loop—that is, the stride identifying each vector $\boldsymbol{x}_i$ in $\boldsymbol{x}$. Specified as: a fullword integer; $incxo \geq 0$.

y
is the vector (or matrix) $\boldsymbol{y}$, containing the $\boldsymbol{y}_i$ vectors of length m, used in the n dot product computations. Specified as: a one- or two-dimensional array of (at least) length $(1+(n-1)(incyo)) + (m-1)|incyi|$, containing numbers of the data type indicated in Table 45 on page 232.

incyi
is the stride for $\boldsymbol{y}$ in the inner loop—that is, the stride identifying the elements in each vector $\boldsymbol{y}_i$. Specified as: a fullword integer. It can have any value.

incyo
is the stride for $\boldsymbol{y}$ in the outer loop—that is, the stride identifying each vector $\boldsymbol{y}_i$ in $\boldsymbol{y}$. Specified as: a fullword integer; $incyo \geq 0$.

On Return

s
is the vector $\boldsymbol{s}$ of length n, containing the results of the n dot product computations. The type of dot product computation depends of the value specified for *isw*.

If $isw = 1$, $s_i \leftarrow \boldsymbol{x}_i \bullet \boldsymbol{y}_i$

If $isw = 2$, $s_i \leftarrow -\boldsymbol{x}_i \bullet \boldsymbol{y}_i$

If $isw = 3$, $s_i \leftarrow s_i + \boldsymbol{x}_i \bullet \boldsymbol{y}_i$

If $isw = 4$, $s_i \leftarrow s_i - \boldsymbol{x}_i \bullet \boldsymbol{y}_i$
where $i = 1, n$

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 45 on page 232.

Function: The four possible computations that can be performed by these subprograms are:

$s_i \leftarrow \boldsymbol{x}_i \bullet \boldsymbol{y}_i$	Store positive dot product
$s_i \leftarrow -\boldsymbol{x}_i \bullet \boldsymbol{y}_i$	Store negative dot product
$s_i \leftarrow s_i+\boldsymbol{x}_i \bullet \boldsymbol{y}_i$	Accumulate positive dot product
$s_i \leftarrow s_i-\boldsymbol{x}_i \bullet \boldsymbol{y}_i$	Accumulate negative dot product

for $i = 1, n$

where each s_i is a scalar element in the vector $\boldsymbol{s}$ of length n, and each of the n $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$ vectors of length m are contained in vectors (or matrices) $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively. Each computation uses the dot product, which is expressed:

$$\boldsymbol{x}_i \bullet \boldsymbol{y}_i = u_1v_1 + u_2v_2 + ... + u_mv_m$$

where u_i and v_i are elements of $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$, respectively. To find the elements for the computation, it uses:

- The strides for the inner loops, *incxi* and *incyi*, to locate the elements in vectors $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$, respectively.
- The strides for the outer loops, *incs, incxo*, and *incyo*, to locate the element s_i in vector $\boldsymbol{s}$ and the vectors $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$ in vectors (or matrices) $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively.

If m or n is 0, no computation is performed. For SNDOT, intermediate results are accumulated in long precision.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. $n < 0$
2. $m < 0$
3. $incs = 0$
4. $isw < 1$ or $isw > 4$
5. $incxo < 0$
6. $incyo < 0$

Example 1: This example shows a store positive dot product computation using vectors with positive strides.

Call Statement and Input

```
            N   M   S  INCS ISW  X  INCXI INCXO  Y  INCYI INCYO
            |   |   |   |    |   |    |     |    |    |     |
CALL SNDOT( 3 , 4 , S , 1  , 1 , X ,  1  ,  4  , Y ,  1  ,  4  )
```

```
          ⎡               ⎤
          ⎢ 1.0  2.0  3.0 ⎥
X    =    ⎢ 2.0  3.0  4.0 ⎥
          ⎢ 3.0  4.0  5.0 ⎥
          ⎢ 4.0  5.0  6.0 ⎥
          ⎣               ⎦
```

```
          ⎡               ⎤
          ⎢ 4.0  3.0  2.0 ⎥
Y    =    ⎢ 3.0  2.0  1.0 ⎥
          ⎢ 2.0  1.0  4.0 ⎥
          ⎢ 1.0  4.0  3.0 ⎥
          ⎣               ⎦
```

Output

```
S         =  (20.0, 36.0, 48.0)
```

Example 2: This example shows a store negative dot product computation using vectors with positive and negative strides.

Call Statement and Input

```
           N   M   S   INCS ISW  X  INCXI INCXO  Y  INCYI INCYO
           |   |   |    |    |  |    |     |    |    |     |
CALL SNDOT( 3 , 4 , S , -1  , 2 , X ,  2  , 10  , Y , -1  ,  6  )
```

```
        ⎡                ⎤
        ⎢ 1.0  2.0  3.0  ⎥
        ⎢  .    .    .   ⎥
        ⎢ 2.0  3.0  4.0  ⎥
        ⎢  .    .    .   ⎥
X   =   ⎢ 3.0  4.0  5.0  ⎥
        ⎢  .    .    .   ⎥
        ⎢ 4.0  5.0  6.0  ⎥
        ⎢  .    .    .   ⎥
        ⎢  .    .    .   ⎥
        ⎢  .    .    .   ⎥
        ⎣                ⎦
```

```
        ⎡                ⎤
        ⎢ 4.0  3.0  2.0  ⎥
        ⎢ 3.0  2.0  1.0  ⎥
Y   =   ⎢ 2.0  1.0  4.0  ⎥
        ⎢ 1.0  4.0  3.0  ⎥
        ⎢  .    .    .   ⎥
        ⎢  .    .    .   ⎥
        ⎣                ⎦
```

Output

```
S       =  (-42.0, -34.0, -30.0)
```

Example 3: This example shows an accumulative positive dot product using vectors with positive and negative strides.

Call Statement and Input

```
           N   M   S  INCS ISW  X  INCXI INCXO  Y  INCYI INCYO
           |   |   |   |    |  |    |     |    |    |     |
CALL SNDOT( 3 , 4 , S , 1  , 3 , X , -2  , 10  , Y ,  2  , 10  )
```

```
S       =  (2.0, 5.0, 8.0)
```

```
        ⎡                ⎤
        ⎢ 1.0  2.0  3.0  ⎥
        ⎢  .    .    .   ⎥
        ⎢ 2.0  3.0  4.0  ⎥
        ⎢  .    .    .   ⎥
X   =   ⎢ 3.0  4.0  5.0  ⎥
        ⎢  .    .    .   ⎥
        ⎢ 4.0  5.0  6.0  ⎥
        ⎢  .    .    .   ⎥
        ⎢  .    .    .   ⎥
        ⎢  .    .    .   ⎥
        ⎣                ⎦
```

```
        ⎡                ⎤
        ⎢ 4.0  3.0  2.0 ⎥
        ⎢  .    .    .  ⎥
        ⎢ 3.0  2.0  1.0 ⎥
        ⎢  .    .    .  ⎥
Y    =  ⎢ 2.0  1.0  4.0 ⎥
        ⎢  .    .    .  ⎥
        ⎢ 1.0  4.0  3.0 ⎥
        ⎢  .    .    .  ⎥
        ⎢  .    .    .  ⎥
        ⎣  .    .    .  ⎦
```

Output

```
S        =  (32.0, 39.0, 50.0)
```

Example 4: This example shows an accumulative negative dot product using vectors with positive and negative strides.

Call Statement and Input

```
            N   M   S   INCS ISW  X  INCXI INCXO  Y  INCYI INCYO
            |   |   |    |    |   |    |     |    |    |     |
CALL SNDOT( 3 , 4 , S , -1  , 4 , X ,  1  ,  6  , Y ,  2  , 10  )
```

```
S        =  (3.0, 6.0, 9.0)
        ⎡                ⎤
        ⎢ 1.0  2.0  3.0 ⎥
        ⎢ 2.0  3.0  4.0 ⎥
X    =  ⎢ 3.0  4.0  5.0 ⎥
        ⎢ 4.0  5.0  6.0 ⎥
        ⎢  .    .    .  ⎥
        ⎣  .    .    .  ⎦
```

```
        ⎡                ⎤
        ⎢ 4.0  3.0  2.0 ⎥
        ⎢  .    .    .  ⎥
        ⎢ 3.0  2.0  1.0 ⎥
        ⎢  .    .    .  ⎥
Y    =  ⎢ 2.0  1.0  4.0 ⎥
        ⎢  .    .    .  ⎥
        ⎢ 1.0  4.0  3.0 ⎥
        ⎢  .    .    .  ⎥
        ⎢  .    .    .  ⎥
        ⎣  .    .    .  ⎦
```

Output

```
S        =  (-45.0, -30.0, -11.0)
```

SNRM2, DNRM2, SCNRM2, and DZNRM2—Euclidean Length of a Vector with Scaling of Input to Avoid Destructive Underflow and Overflow

These subprograms compute the Euclidean length (l_2 norm) of vector ***x***, with scaling of input to avoid destructive underflow and overflow.

Table 46. Data Types

x	**Result**	**Subprogram**
Short-precision real	Short-precision real	SNRM2
Long-precision real	Long-precision real	DNRM2
Short-precision complex	Short-precision real	SCNRM2
Long-precision complex	Long-precision real	DZNRM2

Note: If there is a possibility that your data will cause the computation to overflow or underflow, you should use these subroutines instead of SNORM2, DNORM2, CNORM2, and ZNORM2, because the intermediate computational results may exceed the maximum or minimum limits of the machine. "Notes" on page 240 explains how to estimate whether your data will cause an overflow or underflow.

Syntax

Fortran	SNRM2 | DNRM2 | SCNRM2 | DZNRM2 (*n*, *x*, *incx*)
C and C++	snrm2 | dnrm2 | scnrm2 | dznrm2 (*n*, *x*, *incx*);
PL/I	SNRM2 | DNRM2 | SCNRM2 | DZNRM2 (*n*, *x*, *incx*);

On Entry

n
is the number of elements in vector ***x***. Specified as: a fullword integer; $n \geq 0$.

x
is the vector ***x*** of length *n*, whose Euclidean length is to be computed. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 46.

incx
is the stride for vector ***x***. Specified as: a fullword integer. It can have any value.

On Return

Function value
is the Euclidean length (l_2 norm) of the vector ***x***. Returned as: a number of the data type indicated in Table 46.

Note: Declare this function in your program as returning a value of the data type indicated in Table 46.

Function: The Euclidean length (l_2 norm) of vector ***x*** is expressed as follows, with scaling of input to avoid destructive underflow and overflow:

$$\sqrt{|x_1|^2 + |x_2|^2 + \ldots + |x_n|^2}$$

See reference [73]. The result is returned as the function value. If *n* is 0, then 0.0 is returned as the value of the function.

For SNRM2 and SCNRM2, the sum of the squares of the absolute values of the elements is accumulated in long precision. The square root of this long-precision sum is then computed and, if necessary, is unscaled.

Although these subroutines eliminate destructive underflow, nondestructive underflows may occur if the input elements differ greatly in magnitude. This does not affect accuracy, but it degrades performance. The system default is to mask underflow, which improves the performance of these subroutines.

Error Conditions

Computational Errors: None

Input-Argument Errors: $n < 0$

Important Information About the Following Examples: Workstations use workstation architecture precisions: ANSI/IEEE 32-bit and 64-bit binary floating-point format. The ranges are:

- For short-precision: 3.37×10^{-38} to 3.37×10^{38}
- For long-precision: 1.67×10^{-308} to 1.67×10^{308}

Example 1: This example shows a vector, ***x***, whose elements must be scaled to prevent overflow.

Function Reference and Input

```
               N   X  INCX
               |   |   |
DNORM = DNRM2( 6 , X , 1  )

X      = (0.68056D+200, 0.25521D+200, 0.34028D+200,
          0.85071D+200, 0.25521D+200, 0.85071D+200)
```

Output

```
DNORM    =  0.1469D+201
```

Example 2: This example shows a vector, ***x***, whose elements must be scaled to prevent destructive underflow.

Function Reference and Input

```
               N   X  INCX
               |   |   |
DNORM = DNRM2( 4 , X , 2  )

X     = (0.10795D-200, . , 0.10795D-200, . , 0.10795D-200,
         . , 0.10795D-200)
```

Output

```
DNORM    =  0.21590D-200
```

Example 3: This example shows a vector, ***x***, with a stride of 0. The result in SNORM is:

$$\sqrt{nx_1^2}$$

Function Reference and Input

```
                N   X  INCX
                |   |   |
SNORM = SNRM2( 4 , X , 0  )

X        =  (4.0)
```

Output

```
SNORM    =  8.0
```

Example 4: This example shows a vector, ***x***, containing complex numbers, and whose elements must be scaled to prevent overflow.

Function Reference and Input

```
                 N   X  INCX
                 |   |   |
DZNORM = DZNRM2( 3 , X , 1  )

X     = ((0.68056D+200, 0.25521D+200), (0.34028D+200, 0.85071D+200),
         (0.25521D+200, 0.85071D+200))
```

Output

```
DZNORM   =  0.1469D+201
```

Example 5: This example shows a vector, ***x***, containing complex numbers, and whose elements must be scaled to prevent destructive underflow.

Function Reference and Input

```
                 N   X  INCX
                 |   |   |
DZNORM = DZNRM2( 2 , X , 2  )

X     = ((0.10795D-200, 0.10795D-200), . ,
         (0.10795D-200, 0.10795D-200))
```

Output

```
DZNORM   =  0.2159D-200
```

SNORM2, DNORM2, CNORM2, and ZNORM2—Euclidean Length of a Vector with No Scaling of Input

These subprograms compute the euclidean length (l_2 norm) of vector ***x*** with no scaling of input.

Table 47. Data Types

x	**Result**	**Subprogram**
Short-precision real	Short-precision real	SNORM2
Long-precision real	Long-precision real	DNORM2
Short-precision complex	Short-precision real	CNORM2
Long-precision complex	Long-precision real	ZNORM2

Syntax

Fortran	SNORM2 | DNORM2 | CNORM2 | ZNORM2 (*n*, *x*, *incx*)
C and C++	snorm2 | dnorm2 | cnorm2 | znorm2 (*n*, *x*, *incx*);
PL/I	SNORM2 | DNORM2 | CNORM2 | ZNORM2 (*n*, *x*, *incx*);

On Entry

n
is the number of elements in vector ***x***. Specified as: a fullword integer; $n \geq 0$.

x
is the vector ***x*** of length *n*, whose euclidean length is to be computed. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 47.

incx
is the stride for vector ***x***. Specified as: a fullword integer. It can have any value.

On Return

Function value
is the euclidean length (l_2 norm) of the vector ***x***. Returned as: a number of the data type indicated in Table 47.

Notes

1. This subroutine does not underflow or overflow if the values of the elements in vector ***x*** conform to the following conditions. If these conditions are violated, overflow or destructive underflow may occur:
   - For short-precision numbers:

     Any valid short-precision number.
   - For long-precision numbers:

     $|x_i| = 0$ or $0.10010E{-}145 < |x_i| < 0.13408E{+}155$ for $i = 1, n$
2. Declare this function in your program as returning a value of the data type indicated in Table 47.

Function: The euclidean length (l_2 norm) of vector ***x*** is expressed as follows with no scaling of input:

$$\sqrt{|x_1|^2 + |x_2|^2 + \ldots + |x_n|^2}$$

See reference [73]. The result is returned as the function value. If *n* is 0, then 0.0 is returned as the value of the function.

For SNORM2 and CNORM2, the sum of the squares of the absolute values of the elements is accumulated in long-precision. The square root of this long-precision sum is then computed.

This subroutine should not be used if the values in vector ***x*** do not conform to the restriction given in "Notes" on page 240.

Error Conditions

Computational Errors: None

Input-Argument Errors: $n < 0$

Example 1: This example shows a vector, ***x***, with a stride of 1.

Function Reference and Input

```
                N   X  INCX
                |   |   |
SNORM = SNORM2( 6 , X , 1  )

X       =  (3.0, 4.0, 1.0, 8.0, 1.0, 3.0)
```

Output

```
SNORM   =  10.0
```

Example 2: This example shows a vector, ***x***, with a stride greater than 1.

Function Reference and Input

```
                N   X  INCX
                |   |   |
SNORM = SNORM2( 6 , X , 2  )

X       =  (3.0, . , 4.0, . , 1.0, . , 8.0, . , 1.0, . , 3.0)
```

Output

```
SNORM   =  10.0
```

Example 3: This example shows a vector, ***x***, with a stride of 0. The result in SNORM is:

$$\sqrt{nx_1^2}$$

Function Reference and Input

```
                N   X  INCX
                |   |   |
SNORM = SNORM2( 4 , X , 0  )

X        =  (4.0)
```

Output

```
SNORM    =  8.0
```

Example 4: This example shows a vector, ***x***, containing complex numbers and having a stride of 1.

Function Reference and Input

```
                N   X  INCX
                |   |   |
CNORM = CNORM2( 3 , X , 1  )

X        =  ((3.0, 4.0), (1.0, 8.0), (-1.0, 3.0))
```

Output

```
CNORM    =  10.0
```

SROTG, DROTG, CROTG, and ZROTG—Construct a Givens Plane Rotation

SROTG and DROTG construct a real Givens plane rotation, and CROTG and ZROTG construct a complex Givens plane rotation. The computations use rotational elimination parameters *a* and *b*. Values are returned for *r*, as well as the cosine *c* and the sine *s* of the angle of rotation. SROTG and DROTG also return a value for *z*.

Note: Throughout this description, the symbols *r* and *z* are used to represent two of the output values returned for this computation. It is important to note that the values for *r* and *z* are actually returned in the input-output arguments *a* and *b*, respectively, overwriting the original values passed in *a* and *b*.

Table 48. Data Types

a, *b*, *r*, *s*	*c*	*z*	**Subprogram**
Short-precision real	Short-precision real	Short-precision real	SROTG
Long-precision real	Long-precision real	Long-precision real	DROTG
Short-precision complex	Short-precision real	(No value returned)	CROTG
Long-precision complex	Long-precision real	(No value returned)	ZROTG

Syntax

Fortran	CALL SROTG | DROTG | CROTG | ZROTG (*a*, *b*, *c*, *s*)
C and C++	srotg | drotg | crotg | zrotg (*a*, *b*, *c*, *s*);
PL/I	CALL SROTG | DROTG | CROTG | ZROTG (*a*, *b*, *c*, *s*);

On Entry

a
is the rotational elimination parameter *a*. Specified as: a number of the data type indicated in Table 48.

b
is the rotational elimination parameter *b*. Specified as: a number of the data type indicated in Table 48.

c
See On Return.

s
See On Return.

On Return

a
is the value computed for *r*.

For SROTG and DROTG:

$$r = \sigma\sqrt{a^2 + b^2}$$

where:
σ = SIGN(a) if $|a| > |b|$
σ = SIGN(b) if $|a| \leq |b|$

For CROTG and ZROTG:

$$r = \psi\sqrt{|a|^2 + |b|^2} \quad \text{if } |a| \neq 0$$

$$r = b \quad \text{if } |a| = 0$$

where:
ψ = $a/|a|$

Returned as: a number of the data type indicated in Table 48 on page 243.

b

is the value computed for z.

For SROTG and DROTG:
$z = s$ if $|a| > |b|$
$z = 1/c$ if $|a| \leq |b|$ and $c \neq 0$ and $r \neq 0$
$z = 1$ if $|a| \leq |b|$ and $c = 0$ and $r \neq 0$
$z = 0$ if $r = 0$

For CROTG and ZROTG: no value is returned, and the input value is not changed.

Returned as: a number of the data type indicated in Table 48 on page 243.

c

is the cosine c of the angle of (Givens) rotation. For SROTG and DROTG:
$c = a/r$ if $r \neq 0$
$c = 1$ if $r = 0$

For CROTG and ZROTG:

$$c = \frac{|a|}{\sqrt{|a|^2 + |b|^2}} \quad \text{if } |a| \neq 0$$

$$c = 0 \quad \text{if } |a| = 0$$

Returned as: a number of the data type indicated in Table 48 on page 243.

s

is the sine s of the angle of (Givens) rotation.

For SROTG and DROTG:
$s = b/r$ if $r \neq 0$
$s = 0$ if $r = 0$

For CROTG and ZROTG:

$$s = \frac{\psi \bar{b}}{\sqrt{|a|^2 + |b|^2}} \quad \text{if } |a| \neq 0$$

$$s = (1.0, 0.0) \quad \text{if } |a| = 0$$

where $\psi = a/|a|$

Returned as: a number of the data type indicated in Table 48 on page 243.

Note: In your C program, arguments *a*, *b*, *c*, and *s* must be passed by reference.

Function

SROTG and DROTG: A real Givens plane rotation is constructed for values *a* and *b* by computing values for *r, c, s*, and *z*, where:

$$r = \sigma\sqrt{a^2 + b^2}$$

where:

σ = SIGN(*a*) if $|a| > |b|$
σ = SIGN(*b*) if $|a| \leq |b|$

$c = a/r$ if $r \neq 0$

$c = 1$ if $r = 0$

$s = b/r$ if $r \neq 0$

$s = 0$ if $r = 0$

$z = s$ if $|a| > |b|$

$z = 1/c$ if $|a| \leq |b|$ and $c \neq 0$ and $r \neq 0$

$z = 1$ if $|a| \leq |b|$ and $c = 0$ and $r \neq 0$

$z = 0$ if $r = 0$

See reference [73].

Following are some important points about the computation:

1. The numbers for *c*, *s*, and *r* satisfy:

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

2. Where necessary, scaling is used to avoid overflow and destructive underflow in the computation of *r*, which is expressed as follows:

$$r = \sigma(|a|+|b|)\sqrt{\left(\frac{a}{|a|+|b|}\right)^2+\left(\frac{b}{|a|+|b|}\right)^2}$$

3. σ is not essential to the computation of a Givens rotation matrix, but its use permits later stable reconstruction of *c* and *s* from just one stored number, *z*. See reference [85]. *c* and *s* are reconstructed from *z* as follows:

$$\text{For } z = 1,\ c = 0 \text{ and } s = 1$$
$$\text{For } |z| < 1,\ c = \sqrt{1-z^2} \text{ and } s = z$$
$$\text{For } |z| > 1,\ c = 1/z \text{ and } s = \sqrt{1-c^2}$$

CROTG and ZROTG: A complex Givens plane rotation is constructed for values *a* and *b* by computing values for *r, c*, and *s*, where:

$$r = \psi\sqrt{|a|^2+|b|^2} \qquad \text{if } |a| \neq 0$$

$$r = b \qquad \text{if } |a| = 0$$

where:

$$\psi = a/|a|$$

$$c = \frac{|a|}{\sqrt{|a|^2+|b|^2}} \qquad \text{if } |a| \neq 0$$

$$c = 0 \qquad \text{if } |a| = 0$$

$$s = \frac{\psi\,\overline{b}}{\sqrt{|a|^2 + |b|^2}} \qquad \text{if } |a| \neq 0$$

$$s = (1.0, 0.0) \qquad \text{if } |a| = 0$$

See reference [73].

Following are some important points about the computation:

1. The numbers for *c, s*, and *r* satisfy:

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

2. Where necessary, scaling is used to avoid overflow and destructive underflow in the computation of *r*, which is expressed as follows:

$$r = \psi\,(|a| + |b|)\sqrt{\left|\frac{a}{|a| + |b|}\right|^2 + \left|\frac{b}{|a| + |b|}\right|^2}$$

Error Conditions

Computational Errors: None

Input-Argument Errors: None

Example 1: This example shows the construction of a real Givens plane rotation, where *r* is 0.

Call Statement and Input

```
             A     B    C   S
             |     |    |   |
CALL SROTG( 0.0 , 0.0 , C , S )
```

Output

```
A        =  0.0
B        =  0.0
C        =  1.0
S        =  0.0
```

Example 2: This example shows the construction of a real Givens plane rotation, where c is 0.

Call Statement and Input

```
             A     B    C   S
             |     |    |   |
CALL SROTG( 0.0 , 2.0 , C , S )
```

Output

```
A        =  2.0
B        =  1.0
C        =  0.0
S        =  1.0
```

Example 3: This example shows the construction of a real Givens plane rotation, where $|b| > |a|$.

Call Statement and Input

```
             A      B    C   S
             |      |    |   |
CALL SROTG( 6.0 , -8.0 , C , S )
```

Output

```
A        =  -10.0

B        =  -1.66
C        =  -0.6
S        =  0.8
```

Example 4: This example shows the construction of a real Givens plane rotation, where $|a| > |b|$.

Call Statement and Input

```
             A     B    C   S
             |     |    |   |
CALL SROTG( 8.0 , 6.0 , C , S )
```

Output

```
A        =  10.0
B        =  0.6
C        =  0.8
S        =  0.6
```

Example 5: This example shows the construction of a complex Givens plane rotation, where $|a| = 0$.

Call Statement and Input

```
            A   B   C   S
            |   |   |   |
CALL CROTG( A , B , C , S )

A        =  (0.0, 0.0)
B        =  (1.0, 0.0)
```

Output

```
A        =  (1.0, 0.0)
C        =  0.0
S        =  (1.0, 0.0)
```

Example 6: This example shows the construction of a complex Givens plane rotation, where $|a| \neq 0$.

Call Statement and Input

```
            A   B   C   S
            |   |   |   |
CALL CROTG( A , B , C , S )

A        =  (3.0, 4.0)
B        =  (4.0, 6.0)
```

Output

```
A        =  (5.26, 7.02)
C        =  0.57
S        =  (0.82, -0.05)
```

SROT, DROT, CROT, ZROT, CSROT, and ZDROT—Apply a Plane Rotation

SROT and DROT apply a real plane rotation to real vectors; CROT and ZROT apply a complex plane rotation to complex vectors; and CSROT and ZDROT apply a real plane rotation to complex vectors. The plane rotation is applied to *n* points, where the points to be rotated are contained in vectors ***x*** and ***y***, and where the cosine and sine of the angle of rotation are *c* and *s*, respectively.

Table 49. Data Types

x, ***y***	*c*	*s*	**Subprogram**
Short-precision real	Short-precision real	Short-precision real	SROT
Long-precision real	Long-precision real	Long-precision real	DROT
Short-precision complex	Short-precision real	Short-precision complex	CROT
Long-precision complex	Long-precision real	Long-precision complex	ZROT
Short-precision complex	Short-precision real	Short-precision real	CSROT
Long-precision complex	Long-precision real	Long-precision real	ZDROT

Syntax

Fortran	CALL SROT | DROT | CROT | ZROT | CSROT | ZDROT (*n*, *x*, *incx*, *y*, *incy*, *c*, *s*)
C and C++	srot | drot | crot | zrot | csrot | zdrot (*n*, *x*, *incx*, *y*, *incy*, *c*, *s*);
PL/I	CALL SROT | DROT | CROT | ZROT | CSROT | ZDROT (*n*, *x*, *incx*, *y*, *incy*, *c*, *s*);

On Entry

n
is the number of points to be rotated—that is, the number of elements in vectors ***x*** and ***y***. Specified as: a fullword integer; $n \geq 0$.

x
is the vector ***x*** of length *n*, containing the x_i coordinates of the points to be rotated. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 49.

incx
is the stride for vector ***x***. Specified as: a fullword integer. It can have any value.

y
is the vector ***y*** of length *n*, containing the y_i coordinates of the points to be rotated. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incy|$, containing numbers of the data type indicated in Table 49.

incy
is the stride for vector ***y***. Specified as: a fullword integer. It can have any value.

c
the cosine, *c*, of the angle of rotation. Specified as: a number of the data type indicated in Table 49.

s
the sine, *s*, of the angle of rotation. Specified as: a number of the data type indicated in Table 49.

On Return

x

is the vector ***x*** of length *n*, containing the rotated x_i coordinates, where:

$x_i \leftarrow cx_i+sy_i \quad$ for $i = 1,$

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 49 on page 250.

y

is the vector ***y*** of length *n*, containing the rotated y_i coordinates, where:

For SROT, DROT, CSROT, and ZDROT:

$y_i \leftarrow -sx_i+cy_i \quad$ for $i = 1, n$

For CROT and ZROT:

$$y_i \leftarrow -\bar{s}x_i + cy_i \quad \text{for } i = 1,n$$

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 49 on page 250.

Note: The vectors ***x*** and ***y*** must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.

Function: Applying a plane rotation to *n* points, where the points to be rotated are contained in vectors ***x*** and ***y***, is expressed as follows, where *c* and *s* are the cosine and sine of the angle of rotation, respectively. For SROT, DROT, CSROT, and ZDROT:

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

For CROT and ZROT:

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

See references [54] and [73]. No computation is performed if *n* is 0 or if *c* is 1.0 and *s* is zero. For SROT, CROT, and CSROT, intermediate results are accumulated in long precision.

Error Conditions

Computational Errors: None

Input-Argument Errors: $n < 0$

Example 1: This example shows how to apply a real plane rotation to real vectors ***x*** and ***y*** having positive strides.

Call Statement and Input

```
          N   X  INCX  Y  INCY   C    S
          |   |   |    |   |     |    |
CALL SROT( 5 , X , 1  , Y , 2  , 0.5 , S )

X       =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y       =  (-1.0, . , -2.0, . , -3.0, . , -4.0, . , -5.0)
```

$$S = \frac{\sqrt{3.0}}{2.0}$$

Output

```
X       =  (-0.366, -0.732, -1.098, -1.464, -1.830)
Y       =  (-1.366, -2.732, -4.098, -5.464, -6.830)
```

Example 2: This example shows how to apply a real plane rotation to real vectors ***x*** and ***y*** having strides of opposite sign.

Call Statement and Input

```
          N   X  INCX  Y   INCY   C    S
          |   |   |    |    |     |    |
CALL SROT( 5 , X , 1  , Y , -1  , 0.5 , S )

X       =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y       =  (-5.0, -4.0, -3.0, -2.0, -1.0)
```

$$S = \frac{\sqrt{3.0}}{2.0}$$

Output

```
X       =(same as output X in Example 1)
Y       =  (-6.830, -5.464, -4.098, -2.732, -1.366)
```

Example 3: This example shows how scalar values in vectors ***x*** and ***y*** can be processed by specifying 0 strides and the number of elements to be processed, *n*, equal to 1.

Call Statement and Input

```
          N   X  INCX  Y  INCY   C    S
          |   |   |    |   |     |    |
CALL SROT( 1 , X , 0  , Y , 0  , 0.5 , S )

X       =  (1.0)
Y       =  (-1.0)
```

$$S = \frac{\sqrt{3.0}}{2.0}$$

Output

```
X       =  (-0.366)
Y       =  (-1.366)
```

Example 4: This example shows how to apply a complex plane rotation to complex vectors ***x*** and ***y*** having positive strides.

Call Statement and Input

```
          N   X  INCX  Y  INCY   C    S
          |   |   |    |   |     |    |
CALL CROT( 3 , X , 1  , Y , 2  , 0.5 , S )

X       =  ((1.0, 2.0), (2.0, 3.0), (3.0, 4.0))
Y       =  ((-1.0, 5.0), . , (-2.0, 4.0), . , (-3.0, 3.0))
S       =  (0.75, 0.50)
```

Output

```
X       =  ((-2.750, 4.250), (-2.500, 3.500), (-2.250, 2.750))
Y       =  ((-2.250, 1.500), . , (-4.000, 0.750), . ,
            (-5.750, 0.000))
```

Example 5: This example shows how to apply a real plane rotation to complex vectors ***x*** and ***y*** having positive strides.

Call Statement and Input

```
           N   X  INCX  Y  INCY   C    S
           |   |   |    |   |     |    |
CALL CSROT( 3 , X , 1  , Y , 2  , 0.5 , S )

X       =  ((1.0, 2.0), (2.0, 3.0), (3.0, 4.0))
Y       =  ((-1.0, 5.0), . , (-2.0, 4.0), . , (-3.0, 3.0))
```

$$S = \frac{\sqrt{3.0}}{2.0}$$

Output

```
X       =  ((-0.366, 5.330), (-0.732, 4.964), (-1.098, 4.598))
Y       =  ((-1.366, 0.768), . , (-2.732, -0.598), . ,
            (-4.098, -1.964))
```

SSCAL, DSCAL, CSCAL, ZSCAL, CSSCAL, and ZDSCAL—Multiply a Vector X by a Scalar and Store in the Vector X

These subprograms perform the following computation, using the scalar α and the vector $\boldsymbol{x}$:

$$\boldsymbol{x} \leftarrow \alpha\boldsymbol{x}$$

Table 50. Data Types

α	$\boldsymbol{x}$	Subprogram
Short-precision real	Short-precision real	SSCAL
Long-precision real	Long-precision real	DSCAL
Short-precision complex	Short-precision complex	CSCAL
Long-precision complex	Long-precision complex	ZSCAL
Short-precision real	Short-precision complex	CSSCAL
Long-precision real	Long-precision complex	ZDSCAL

Syntax

Fortran	CALL SSCAL | DSCAL | CSCAL | ZSCAL | CSSCAL | ZDSCAL (*n*, *alpha*, *x*, *incx*)
C and C++	sscal | dscal | cscal | zscal | csscal | zdscal (*n*, *alpha*, *x*, *incx*);
PL/I	CALL SSCAL | DSCAL | CSCAL | ZSCAL | CSSCAL | ZDSCAL (*n*, *alpha*, *x*, *incx*);

On Entry

n
is the number of elements in vector $\boldsymbol{x}$. Specified as: a fullword integer; $n \geq 0$.

alpha
is the scalar α. Specified as: a number of the data type indicated in Table 50.

x
is the vector $\boldsymbol{x}$ of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 50.

incx
is the stride for vector $\boldsymbol{x}$. Specified as: a fullword integer. It can have any value.

On Return

x
is the vector $\boldsymbol{x}$ of length *n*, containing the result of the computation $\alpha\boldsymbol{x}$. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 50.

Note: The fastest way in ESSL to zero out contiguous (stride 1) arrays is to call SSCAL or DSCAL, specifying $incx = 1$ and $\alpha = 0$.

Function: The computation is expressed as follows:

$$\begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix} \leftarrow \alpha \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix}$$

See reference [73]. If n is 0, no computation is performed. For CSCAL, intermediate results are accumulated in long precision.

Error Conditions

Computational Errors: None

Input-Argument Errors: $n < 0$

Example 1: This example shows a vector, ***x***, with a stride of 1.

Call Statement and Input

```
            N  ALPHA  X  INCX
            |    |    |   |
CALL SSCAL( 5 , 2.0 , X , 1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
```

Output

```
X        =  (2.0, 4.0, 6.0, 8.0, 10.0)
```

Example 2: This example shows vector, ***x***, with a stride greater than 1.

Call Statement and Input

```
            N  ALPHA  X  INCX
            |    |    |   |
CALL SSCAL( 5 , 2.0 , X , 2  )

X        =  (1.0, . , 2.0, . , 3.0, . , 4.0, . , 5.0)
```

Output

```
X        =  (2.0, . , 4.0, . , 6.0, . , 8.0, . , 10.0)
```

Example 3: This example illustrates that when the strides for two similar computations (Example 1 and Example 3) have the same absolute value but have opposite signs, the output is the same. This example is the same as Example 1, except the stride for ***x*** is negative (–1). For performance reasons, it is better to specify the positive stride. For ***x***, processing begins at element `X(5)`, which is 5.0, and results are stored beginning at the same element.

Call Statement and Input

```
            N  ALPHA  X   INCX
            |    |    |    |
CALL SSCAL( 5 , 2.0 , X , -1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
```

Output

```
X        =  (2.0, 4.0, 6.0, 8.0, 10.0)
```

Example 4: This example shows how SSCAL can be used to compute a scalar value. In this case, input vector ***x*** contains a scalar value, and the stride is 0. The number of elements to be processed, *n*, is 1.

Call Statement and Input

```
            N  ALPHA  X  INCX
            |    |    |   |
CALL SSCAL( 1 , 2.0 , X , 0  )

X        =  (1.0)
```

Output

```
X        =  (2.0)
```

Example 5: This example shows a scalar, α, and a vector, ***x***, containing complex numbers, where vector ***x*** has a stride of 1.

Call Statement and Input

```
            N  ALPHA  X  INCX
            |    |    |   |
CALL CSCAL( 3 ,ALPHA, X , 1  )

ALPHA    =  (2.0, 3.0)
X        =  ((1.0, 2.0), (2.0, 0.0), (3.0, 5.0))
```

Output

```
X        =  ((-4.0, 7.0), (4.0, 6.0), (-9.0, 19.0))
```

Example 6: This example shows a scalar, α, containing a real number, and a vector, ***x***, containing complex numbers, where vector ***x*** has a stride of 1.

Call Statement and Input

```
             N  ALPHA  X  INCX
             |    |    |   |
CALL CSSCAL( 3 , 2.0 , X , 1  )

X        =  ((1.0, 2.0), (2.0, 0.0), (3.0, 5.0))
```

Output

```
X        =  ((2.0, 4.0), (4.0, 0.0), (6.0, 10.0))
```

SSWAP, DSWAP, CSWAP, and ZSWAP—Interchange the Elements of Two Vectors

These subprograms interchange the elements of vectors ***x*** and ***y***:

$$y \longleftrightarrow x$$

Table 51. Data Types

x, ***y***	**Subprogram**
Short-precision real	SSWAP
Long-precision real	DSWAP
Short-precision complex	CSWAP
Long-precision complex	ZSWAP

Syntax

Fortran	CALL SSWAP | DSWAP | CSWAP | ZSWAP (*n*, *x*, *incx*, *y*, *incy*)
C and C++	sswap | dswap | cswap | zswap (*n*, *x*, *incx*, *y*, *incy*);
PL/I	CALL SSWAP | DSWAP | CSWAP | ZSWAP (*n*, *x*, *incx*, *y*, *incy*);

On Entry

n
is the number of elements in vectors ***x*** and ***y***. Specified as: a fullword integer; $n \geq 0$.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 51.

incx
is the stride for vector ***x***. Specified as: a fullword integer. It can have any value.

y
is the vector ***y*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incy|$, containing numbers of the data type indicated in Table 51.

incy
is the stride for vector ***y***. Specified as: a fullword integer. It can have any value.

On Return

x
is the vector ***x*** of length *n*, containing the elements that were swapped from vector ***y***. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 51.

y
is the vector ***y*** of length *n*, containing the elements that were swapped from vector ***x***. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 51.

Notes

1. If you specify the same vector for ***x*** and ***y***, then *incx* and *incy* must be equal; otherwise, results are unpredictable.
2. If you specify different vectors for ***x*** and ***y***, they must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.

Function: The elements of vectors ***x*** and ***y*** are interchanged as follows:

$$\begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix} \leftarrow \rightarrow \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix}$$

See reference [73]. If n is 0, no elements are interchanged.

Error Conditions

Computational Errors: None

Input-Argument Errors: $n < 0$

Example 1: This example shows vectors ***x*** and ***y*** with positive strides.

Call Statement and Input

```
            N   X  INCX  Y  INCY
            |   |   |    |   |
CALL SSWAP( 5 , X , 1  , Y , 2  )

X       =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y       =  (-1.0, . , -2.0, . , -3.0, . , -4.0, . , -5.0)
```

Output

```
X       =  (-1.0, -2.0, -3.0, -4.0, -5.0)
Y       =  (1.0, . , 2.0, . , 3.0, . , 4.0, . , 5.0)
```

Example 2: This example shows how to obtain output vectors ***x*** and ***y*** that are reverse copies of the input vectors ***y*** and ***x***. You must specify strides with the same absolute value, but with opposite signs. For ***y***, which has negative stride, processing begins at element `Y(5)`, which is –5.0, and the results of the swap are stored beginning at the same element.

Call Statement and Input

```
            N   X  INCX  Y   INCY
            |   |   |    |    |
CALL SSWAP( 5 , X , 1  , Y , -1  )

X       =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y       =  (-1.0, -2.0, -3.0, -4.0, -5.0)
```

Output

```
X        =  (-5.0, -4.0, -3.0, -2.0, -1.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Example 3: This example shows how SSWAP can be used to interchange scalar values in vectors ***x*** and ***y*** by specifying 0 strides and the number of elements to be processed as 1.

Call Statement and Input

```
            N   X  INCX  Y  INCY
            |   |   |    |   |
CALL SSWAP( 1 , X , 0  , Y , 0  )

X        =  (1.0)
Y        =  (-4.0)
```

Output

```
X        =  (-4.0)
Y        =  (1.0)
```

Example 4: This example shows vectors ***x*** and ***y***, containing complex numbers and having positive strides.

Call Statement and Input

```
            N   X  INCX  Y  INCY
            |   |   |    |   |
CALL CSWAP( 4 , X , 1  , Y , 2  )

X        =  ((1.0, 6.0), (2.0, 7.0), (3.0, 8.0), (4.0, 9.0))
Y        =  ((-1.0, -1.0), . , (-2.0, -2.0), . , (-3.0, -3.0), . ,
             (-4.0, -4.0))
```

Output

```
X        =  ((-1.0, -1.0), (-2.0, -2.0), (-3.0, -3.0), (-4.0, -4.0))
Y        =  ((1.0, 6.0), . , (2.0, 7.0), . , (3.0, 8.0), . ,
             (4.0, 9.0))
```

SVEA, DVEA, CVEA, and ZVEA—Add a Vector X to a Vector Y and Store in a Vector Z

These subprograms perform the following computation, using vectors ***x, y,*** and ***z***:

$z \leftarrow x+y$

Table 52. Data Types

x, ***y***, ***z***	**Subprogram**
Short-precision real	SVEA
Long-precision real	DVEA
Short-precision complex	CVEA
Long-precision complex	ZVEA

Syntax

Fortran	CALL SVEA | DVEA | CVEA | ZVEA (*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*)
C and C++	svea | dvea | cvea | zvea (*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*);
PL/I	CALL SVEA | DVEA | CVEA | ZVEA (*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*);

On Entry

n
is the number of elements in vectors ***x, y,*** and ***z***. Specified as: a fullword integer; $n \geq 0$.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 52.

incx
is the stride for vector ***x***. Specified as: a fullword integer. It can have any value.

y
is the vector ***y*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incy|$, containing numbers of the data type indicated in Table 52.

incy
is the stride for vector ***y***. Specified as: a fullword integer. It can have any value.

z
See On Return.

incz
is the stride for vector ***z***. Specified as: a fullword integer. It can have any value.

On Return

z
is the vector ***z*** of length *n*, containing the result of the computation. Returned as: a one-dimensional array of (at least) length $1+(n-1)|incz|$, containing numbers of the data type indicated in Table 52.

Notes

1. If you specify the same vector for ***x*** and ***z***, then *incx* and *incz* must be equal; otherwise, results are unpredictable. The same is true for ***y*** and ***z***.
2. If you specify different vectors for ***x*** and ***z***, they must have no common elements; otherwise, results are unpredictable. The same is true for ***y*** and ***z***. See "Concepts" on page 55.

Function: The computation is expressed as follows:

$$\begin{bmatrix} z_1 \\ . \\ . \\ . \\ z_n \end{bmatrix} \leftarrow \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix}$$

If n is 0, no computation is performed.

Error Conditions

Computational Errors: None

Input-Argument Errors: $n < 0$

Example 1: This example shows vectors ***x***, ***y***, and ***z***, with positive strides.

Call Statement and Input

```
           N   X  INCX  Y  INCY  Z  INCZ
           |   |   |    |   |    |   |
CALL SVEA( 5 , X , 1  , Y , 2  , Z , 1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (1.0, . , 1.0, . , 1.0, . , 1.0, . , 1.0)
```

Output

```
Z        =  (2.0, 3.0, 4.0, 5.0, 6.0)
```

Example 2: This example shows vectors ***x*** and ***y*** having strides of opposite sign, and an output vector ***z*** having a positive stride. For ***y***, which has negative stride, processing begins at element `Y(5)`, which is 1.0.

Call Statement and Input

```
           N   X  INCX  Y   INCY  Z  INCZ
           |   |   |    |    |    |   |
CALL SVEA( 5 , X , 1  , Y , -1  , Z , 2  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output

```
Z        =  (2.0, . , 4.0, . , 6.0, . , 8.0, . , 10.0)
```

Example 3: This example shows a vector, ***x***, with 0 stride and a vector, *z*, with negative stride. ***x*** is treated like a vector of length *n*, all of whose elements are the same as the single element in ***x***. For vector *z*, results are stored beginning in element `Z(5)`.

Call Statement and Input

```
           N   X  INCX  Y  INCY  Z   INCZ
           |   |   |    |   |    |    |
CALL SVEA( 5 , X , 0  , Y , 1  , Z , -1  )

X        =  (1.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output

```
Z        =  (2.0, 3.0, 4.0, 5.0, 6.0)
```

Example 4: This example shows a vector, ***y***, with 0 stride. ***y*** is treated like a vector of length *n*, all of whose elements are the same as the single element in ***y***.

Call Statement and Input

```
           N   X  INCX  Y  INCY  Z  INCZ
           |   |   |    |   |    |   |
CALL SVEA( 5 , X , 1  , Y , 0  , Z , 1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0)
```

Output

```
Z        =  (6.0, 7.0, 8.0, 9.0, 10.0)
```

Example 5: This example shows the output vector, *z*, with 0 stride, where the vector ***x*** has positive stride, and the vector ***y*** has 0 stride. The number of elements to be processed, *n*, is greater than 1.

Call Statement and Input

```
           N   X  INCX  Y  INCY  Z  INCZ
           |   |   |    |   |    |   |
CALL SVEA( 5 , X , 1  , Y , 0  , Z , 0  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0)
```

Output

```
Z        =  (10.0)
```

Example 6: This example shows the output vector ***z***, with 0 stride, where the vector ***x*** has 0 stride, and the vector ***y*** has negative stride. The number of elements to be processed, *n*, is greater than 1.

Call Statement and Input

```
           N   X  INCX  Y   INCY  Z  INCZ
           |   |   |    |    |    |   |
CALL SVEA( 5 , X , 0  , Y , -1  , Z , 0  )

X        =  (1.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output

```
Z        =  (6.0)
```

Example 7: This example shows how SVEA can be used to compute a scalar value. In this case, vectors ***x*** and ***y*** contain scalar values. The strides of all vectors, ***x***, ***y***, and ***z***, are 0. The number of elements to be processed, *n*, is 1.

Call Statement and Input

```
           N   X  INCX  Y  INCY  Z  INCZ
           |   |   |    |   |    |   |
CALL SVEA( 1 , X , 0  , Y , 0  , Z , 0  )

X        =  (1.0)
Y        =  (5.0)
```

Output

```
Z        =  (6.0)
```

Example 8: This example shows vectors ***x*** and ***y***, containing complex numbers and having positive strides.

Call Statement and Input

```
           N   X  INCX  Y  INCY  Z  INCZ
           |   |   |    |   |    |   |
CALL CVEA( 3 , X , 1  , Y , 2  , Z , 1  )

X        =  ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
Y        =  ((7.0, 8.0), . , (9.0, 10.0), . , (11.0, 12.0))
```

Output

```
Z        =  ((8.0, 10.0), (12.0, 14.0), (16.0, 18.0))
```

SVES, DVES, CVES, and ZVES—Subtract a Vector Y from a Vector X and Store in a Vector Z

These subprograms perform the following computation, using vectors ***x, y,*** and ***z***:

$z \leftarrow x-y$

Table 53. Data Types

x, ***y***, ***z***	**Subprogram**
Short-precision real	SVES
Long-precision real	DVES
Short-precision complex	CVES
Long-precision complex	ZVES

Syntax

Fortran	CALL SVES | DVES | CVES | ZVES (*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*)
C and C++	sves | dves | cves | zves (*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*);
PL/I	CALL SVES | DVES | CVES | ZVES (*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*);

On Entry

n
is the number of elements in vectors ***x, y,*** and ***z***. Specified as: a fullword integer; $n \geq 0$.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 53.

incx
is the stride for vector ***x***. Specified as: a fullword integer. It can have any value.

y
is the vector ***y*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incy|$, containing numbers of the data type indicated in Table 53.

incy
is the stride for vector ***y***. Specified as: a fullword integer. It can have any value.

z
See On Return.

incz
is the stride for vector ***z***. Specified as: a fullword integer. It can have any value.

On Return

z
is the vector ***z*** of length *n*, containing the result of the computation. Returned as: a one-dimensional array of (at least) length $1+(n-1)|incz|$, containing numbers of the data type indicated in Table 53.

Notes

1. If you specify the same vector for ***x*** and ***z***, then *incx* and *incz* must be equal; otherwise, results are unpredictable. The same is true for ***y*** and ***z***.
2. If you specify different vectors for ***x*** and ***z***, they must have no common elements; otherwise, results are unpredictable. The same is true for ***y*** and ***z***. See "Concepts" on page 55.

Function: The computation is expressed as follows:

$$\begin{bmatrix} z_1 \\ . \\ . \\ . \\ z_n \end{bmatrix} \leftarrow \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix} - \begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix}$$

If *n* is 0, no computation is performed.

Error Conditions

Computational Errors: None

Input-Argument Errors: $n < 0$

Example 1: This example shows vectors ***x***, ***y***, and ***z***, with positive strides.

Call Statement and Input

```
           N   X  INCX  Y  INCY  Z  INCZ
           |   |   |    |   |    |   |
CALL SVES( 5 , X , 1  , Y , 2  , Z , 1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (1.0, . , 1.0, . , 1.0, . , 1.0, . , 1.0)
```

Output

```
Z        =  (0.0, 1.0, 2.0, 3.0, 4.0)
```

Example 2: This example shows vectors ***x*** and ***y*** having strides of opposite sign, and an output vector ***z*** having a positive stride. For ***y***, which has negative stride, processing begins at element `Y(5)`, which is 1.0.

Call Statement and Input

```
           N   X  INCX  Y   INCY  Z  INCZ
           |   |   |    |    |    |   |
CALL SVES( 5 , X , 1  , Y , -1  , Z , 2  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output

```
Z        =  (0.0, . , 0.0, . , 0.0, . , 0.0, . , 0.0)
```

Example 3: This example shows a vector, ***x***, with 0 stride, and a vector, ***z***, with negative stride. ***x*** is treated like a vector of length *n*, all of whose elements are the same as the single element in ***x***. For vector *z*, results are stored beginning in element `Z(5)`.

Call Statement and Input

```
           N   X  INCX  Y  INCY  Z   INCZ
           |   |   |    |   |    |    |
CALL SVES( 5 , X , 0  , Y , 1  , Z , -1  )

X        =  (1.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output

```
Z        =  (0.0, -1.0, -2.0, -3.0, -4.0)
```

Example 4: This example shows a vector, ***y***, with 0 stride. ***y*** is treated like a vector of length *n*, all of whose elements are the same as the single element in ***y***.

Call Statement and Input

```
           N   X  INCX  Y  INCY  Z  INCZ
           |   |   |    |   |    |   |
CALL SVES( 5 , X , 1  , Y , 0  , Z , 1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0)
```

Output

```
Z        =  (-4.0, -3.0, -2.0, -1.0, 0.0)
```

Example 5: This example shows the output vector ***z***, with 0 stride, where the vector ***x*** has positive stride, and the vector ***y*** has 0 stride. The number of elements to be processed, *n*, is greater than 1.

Call Statement and Input

```
           N   X  INCX  Y  INCY  Z  INCZ
           |   |   |    |   |    |   |
CALL SVES( 5 , X , 1  , Y , 0  , Z , 0  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0)
```

Output

```
Z        =  (0.0)
```

Example 6: This example shows the output vector ***z***, with 0 stride, where the vector ***x*** has 0 stride, and the vector ***y*** has negative stride. The number of elements to be processed, *n*, is greater than 1.

Call Statement and Input

```
           N   X  INCX  Y   INCY  Z  INCZ
           |   |   |    |    |    |   |
CALL SVES( 5 , X , 0  , Y , -1  , Z , 0  )

X        =  (1.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output

```
Z        =  (-4.0)
```

Example 7: This example shows how SVES can be used to compute a scalar value. In this case, vectors ***x*** and ***y*** contain scalar values. The strides of all vectors, ***x***, ***y***, and ***z***, are 0. The number of elements to be processed, *n*, is 1.

Call Statement and Input

```
           N   X  INCX  Y  INCY  Z  INCZ
           |   |   |    |   |    |   |
CALL SVES( 1 , X , 0  , Y , 0  , Z , 0  )

X        =  (1.0)
Y        =  (5.0)
```

Output

```
Z        =  (-4.0)
```

Example 8: This example shows vectors ***x*** and ***y***, containing complex numbers and having positive strides.

Call Statement and Input

```
           N   X  INCX  Y  INCY  Z  INCZ
           |   |   |    |   |    |   |
CALL CVES( 3 , X , 1  , Y , 2  , Z , 1  )

X        =  ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
Y        =  ((7.0, 8.0), . , (9.0, 10.0), . , (11.0, 12.0))
```

Output

```
Z        =  ((-6.0, -6.0), (-6.0, -6.0), (-6.0, -6.0))
```

SVEM, DVEM, CVEM, and ZVEM—Multiply a Vector X by a Vector Y and Store in a Vector Z

These subprograms perform the following computation, using vectors ***x, y,*** and ***z***:

$z \leftarrow xy$

Table 54. Data Types

x, ***y***, ***z***	**Subprogram**
Short-precision real	SVEM
Long-precision real	DVEM
Short-precision complex	CVEM
Long-precision complex	ZVEM

Syntax

Fortran	CALL SVEM | DVEM | CVEM | ZVEM (*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*)
C and C++	svem | dvem | cvem | zvem (*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*);
PL/I	CALL SVEM | DVEM | CVEM | ZVEM (*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*);

On Entry

n
is the number of elements in vectors ***x, y,*** and ***z***. Specified as: a fullword integer; $n \geq 0$.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 54.

incx
is the stride for vector ***x***. Specified as: a fullword integer. It can have any value.

y
is the vector ***y*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incy|$, containing numbers of the data type indicated in Table 54.

incy
is the stride for vector ***y***. Specified as: a fullword integer. It can have any value.

z
See On Return.

incz
is the stride for vector ***z***. Specified as: a fullword integer. It can have any value.

On Return

z
is the vector ***z*** of length *n*, containing the result of the computation. Returned as: a one-dimensional array of (at least) length $1+(n-1)|incz|$, containing numbers of the data type indicated in Table 54.

Notes

1. If you specify the same vector for ***x*** and ***z***, then *incx* and *incz* must be equal; otherwise, results are unpredictable. The same is true for ***y*** and ***z***.
2. If you specify different vectors for ***x*** and ***z***, they must have no common elements; otherwise, results are unpredictable. The same is true for ***y*** and ***z***. See "Concepts" on page 55.

Function: The computation is expressed as follows:

$z_i \leftarrow x_i y_i \quad$ for $i = 1, n$

If n is 0, no computation is performed. For CVEM, intermediate results are accumulated in long precision (short-precision Multiply followed by a long-precision Add), with the final result truncated to short precision.

Error Conditions

Computational Errors: None

Input-Argument Errors: $n < 0$

Example 1: This example shows vectors ***x***, ***y***, and ***z***, with positive strides.

Call Statement and Input

```
           N   X  INCX  Y  INCY  Z  INCZ
           |   |   |    |   |    |   |
CALL SVEM( 5 , X , 1  , Y , 2  , Z , 1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (1.0, . , 1.0, . , 1.0, . , 1.0, . , 1.0)
```

Output

```
Z        =  (1.0, 2.0, 3.0, 4.0, 5.0)
```

Example 2: This example shows vectors ***x*** and ***y*** having strides of opposite sign, and an output vector ***z*** having a positive stride. For ***y***, which has negative stride, processing begins at element `Y(5)`, which is 1.0.

Call Statement and Input

```
           N   X  INCX  Y   INCY  Z  INCZ
           |   |   |    |    |    |   |
CALL SVEM( 5 , X , 1  , Y , -1  , Z , 2  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output

```
Z        =  (1.0, . , 4.0, . , 9.0, . , 16.0, . , 25.0)
```

Example 3: This example shows a vector, ***x***, with 0 stride, and a vector, ***z***, with negative stride. ***x*** is treated like a vector of length n, all of whose elements are the same as the single element in ***x***. For vector ***z***, results are stored beginning in element `Z(5)`.

Call Statement and Input

```
           N   X  INCX  Y  INCY  Z   INCZ
           |   |   |    |   |    |    |
CALL SVEM( 5 , X , 0  , Y , 1  , Z , -1  )

X        =  (1.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output

```
Z        =  (1.0, 2.0, 3.0, 4.0, 5.0)
```

Example 4: This example shows a vector, ***y***, with 0 stride. ***y*** is treated like a vector of length *n*, all of whose elements are the same as the single element in ***y***.

Call Statement and Input

```
           N   X  INCX  Y  INCY  Z  INCZ
           |   |   |    |   |    |   |
CALL SVEM( 5 , X , 1  , Y , 0  , Z , 1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0)
```

Output

```
Z        =  (5.0, 10.0, 15.0, 20.0, 25.0)
```

Example 5: This example shows the output vector, ***z***, with 0 stride, where the vector ***x*** has positive stride, and the vector ***y*** has 0 stride. The number of elements to be processed, *n*, is greater than 1.

Call Statement and Input

```
           N   X  INCX  Y  INCY  Z  INCZ
           |   |   |    |   |    |   |
CALL SVEM( 5 , X , 1  , Y , 0  , Z , 0  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0)
```

Output

```
Z        =  (25.0)
```

Example 6: This example shows the output vector ***z***, with 0 stride, where the vector ***x*** has 0 stride, and the vector ***y*** has negative stride. The number of elements to be processed, *n*, is greater than 1.

Call Statement and Input

```
           N   X  INCX  Y   INCY  Z  INCZ
           |   |   |    |    |    |   |
CALL SVEM( 5 , X , 0  , Y , -1  , Z , 0  )

X        =  (1.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output

```
Z        =  (5.0)
```

Example 7: This example shows how SVEM can be used to compute a scalar value. In this case, vectors ***x*** and ***y*** contain scalar values. The strides of all vectors, ***x***, ***y***, and *z*, are 0. The number of elements to be processed, *n*, is 1.

Call Statement and Input

```
           N   X  INCX  Y  INCY  Z  INCZ
           |   |   |    |   |    |   |
CALL SVEM( 1 , X , 0  , Y , 0  , Z , 0  )

X        =  (1.0)
Y        =  (5.0)
```

Output

```
Z        =  (5.0)
```

Example 8: This example shows vectors ***x*** and ***y***, containing complex numbers and having positive strides.

Call Statement and Input

```
           N   X  INCX  Y  INCY  Z  INCZ
           |   |   |    |   |    |   |
CALL CVEM( 3 , X , 1  , Y , 2  , Z , 1  )

X        =  ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
Y        =  ((7.0, 8.0), . , (9.0, 10.0), . , (11.0, 12.0))
```

Output

```
Z        =  ((-9.0, 22.0), (-13.0, 66.0), (-17.0, 126.0))
```

SYAX, DYAX, CYAX, ZYAX, CSYAX, and ZDYAX—Multiply a Vector X by a Scalar and Store in a Vector Y

These subprograms perform the following computation, using the scalar α and vectors ***x*** and ***y***:

$$\boldsymbol{y} \leftarrow \alpha\boldsymbol{x}$$

Table 55. Data Types

α	***x***, ***y***	**Subprogram**
Short-precision real	Short-precision real	SYAX
Long-precision real	Long-precision real	DYAX
Short-precision complex	Short-precision complex	CYAX
Long-precision complex	Long-precision complex	ZYAX
Short-precision real	Short-precision complex	CSYAX
Long-precision real	Long-precision complex	ZDYAX

Syntax

Fortran	CALL SYAX | DYAX | CYAX | ZYAX | CSYAX | ZDYAX (*n*, *alpha*, *x*, *incx*, *y*, *incy*)
C and C++	syax | dyax | cyax | zyax | csyax | zdyax (*n*, *alpha*, *x*, *incx*, *y*, *incy*);
PL/I	CALL SYAX | DYAX | CYAX | ZYAX | CSYAX | ZDYAX (*n*, *alpha*, *x*, *incx*, *y*, *incy*);

On Entry

n
is the number of elements in vector ***x*** and ***y***. Specified as: a fullword integer; $n \geq 0$.

alpha
is the scalar α. Specified as: a number of the data type indicated in Table 55.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 55.

incx
is the stride for vector ***x***. Specified as: a fullword integer. It can have any value.

y
See On Return.

incy
is the stride for vector ***y***. Specified as: a fullword integer. It can have any value.

On Return

y
is the vector ***y*** of length *n*, containing the result of the computation α***x***. Returned as: a one-dimensional array of (at least) length $1+(n-1)|incy|$, containing numbers of the data type indicated in Table 55.

Notes

1. If you specify the same vector for ***x*** and ***y***, then *incx* and *incy* must be equal; otherwise, results are unpredictable.
2. If you specify different vectors for ***x*** and ***y***, they must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.

Function: The computation is expressed as follows:

$$\begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix} \leftarrow \alpha \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix}$$

See reference [73]. If n is 0, no computation is performed. For CYAX, intermediate results are accumulated in long precision.

Error Condition

Computational Errors: None

Input-Argument Errors: $n < 0$

Example 1: This example shows vectors ***x*** and ***y*** with positive strides.

Call Statement and Input

```
           N  ALPHA  X  INCX  Y  INCY
           |    |    |   |    |   |
CALL SYAX( 5 , 2.0 , X , 1  , Y , 2  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
```

Output

```
Y        =  (2.0, . , 4.0, . , 6.0, . , 8.0, . , 10.0)
```

Example 2: This example shows vectors ***x*** and ***y*** that have strides of opposite signs. For ***y***, which has negative stride, results are stored beginning in element `Y(5)`.

Call Statement and Input

```
           N  ALPHA  X  INCX  Y   INCY
           |    |    |   |    |    |
CALL SYAX( 5 , 2.0 , X , 1  , Y , -1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
```

Output

```
Y        =  (10.0, 8.0, 6.0, 4.0, 2.0)
```

Example 3: This example shows a vector, ***x***, with 0 stride. ***x*** is treated like a vector of length n, all of whose elements are the same as the single element in ***x***.

Call Statement and Input

```
           N  ALPHA  X  INCX  Y  INCY
           |    |    |   |    |   |
CALL SYAX( 5 , 2.0 , X , 0  , Y , 1  )

X       =  (1.0)
```

Output

```
Y       =  (2.0, 2.0, 2.0, 2.0, 2.0)
```

Example 4: This example shows how SYAX can be used to compute a scalar value. In this case both vectors ***x*** and ***y*** contain scalar values, and the strides for both vectors are 0. The number of elements to be processed, *n*, is 1.

Call Statement and Input

```
           N  ALPHA  X  INCX  Y  INCY
           |    |    |   |    |   |
CALL SYAX( 1 , 2.0 , X , 0  , Y , 0  )

X       =  (1.0)
```

Output

```
Y       =  (2.0)
```

Example 5: This example shows a scalar, α, and vectors ***x*** and ***y***, containing complex numbers, where both vectors have a stride of 1.

Call Statement and Input

```
           N  ALPHA  X  INCX  Y  INCY
           |    |    |   |    |   |
CALL CYAX( 3 ,ALPHA, X , 1  , Y , 1  )

ALPHA   =  (2.0, 3.0)
X       =  ((1.0, 2.0), (2.0, 0.0), (3.0, 5.0))
```

Output

```
Y       =  ((-4.0, 7.0), (4.0, 6.0), (-9.0, 19.0))
```

Example 6: This example shows a scalar, α, containing a real number, and vectors ***x*** and ***y***, containing complex numbers, where both vectors have a stride of 1.

Call Statement and Input

```
            N  ALPHA  X  INCX  Y  INCY
            |    |    |   |    |   |
CALL CSYAX( 3 , 2.0 , X , 1  , Y , 1  )

X       =  ((1.0, 2.0), (2.0, 0.0), (3.0, 5.0))
```

Output

```
Y       =  ((2.0, 4.0), (4.0, 0.0), (6.0, 10.0))
```

SZAXPY, DZAXPY, CZAXPY, and ZZAXPY—Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in a Vector Z

These subprograms perform the following computation, using the scalar α and vectors ***x***, ***y***, and *z*:

$z \leftarrow y+\alpha x$

Table 56. Data Types

α, ***x***, ***y***, *z*	**Subprogram**
Short-precision real	SZAXPY
Long-precision real	DZAXPY
Short-precision complex	CZAXPY
Long-precision complex	ZZAXPY

Syntax

Fortran	CALL SZAXPY | DZAXPY | CZAXPY | ZZAXPY (*n*, *alpha*, *x*, *incx*, *y*, *incy*, *z*, *incz*)
C and C++	szaxpy | dzaxpy | czaxpy | zzaxpy (*n*, *alpha*, *x*, *incx*, *y*, *incy*, *z*, *incz*);
PL/I	CALL SZAXPY | DZAXPY | CZAXPY | ZZAXPY (*n*, *alpha*, *x*, *incx*, *y*, *incy*, *z*, *incz*);

On Entry

n
is the number of elements in vectors ***x, y,*** and ***z***. Specified as: a fullword integer; $n \geq 0$.

alpha
is the scalar α. Specified as: a number of the data type indicated in Table 56.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 56.

incx
is the stride for vector ***x***. Specified as: a fullword integer. It can have any value.

y
is the vector ***y*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incy|$, containing numbers of the data type indicated in Table 56.

incy
is the stride for vector ***y***. Specified as: a fullword integer. It can have any value.

z
See On Return.

incz
is the stride for vector ***z***. Specified as: a fullword integer. It can have any value.

On Return

z
is the vector ***z*** of length *n*, containing the result of the computation $y+\alpha x$. Returned as: a one-dimensional array of (at least) length $1+(n-1)|incz|$, containing numbers of the data type indicated in Table 56.

Notes

1. If you specify the same vector for ***x*** and ***z***, then *incx* and *incz* must be equal; otherwise, results are unpredictable. The same is true for ***y*** and ***z***.
2. If you specify different vectors for ***x*** and ***z***, they must have no common elements; otherwise, results are unpredictable. The same is true for ***y*** and ***z***. See "Concepts" on page 55.

Function: The computation is expressed as follows:

$$\begin{bmatrix} z_1 \\ . \\ . \\ . \\ z_n \end{bmatrix} \leftarrow \begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix} + \alpha \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix}$$

See reference [73]. If n is 0, no computation is performed. For CZAXPY, intermediate results are accumulated in long precision.

Error Conditions

Computational Errors: None

Input-Argument Errors: $n < 0$

Example 1: This example shows vectors ***x*** and ***y*** with positive strides.

Call Statement and Input

```
             N  ALPHA  X  INCX  Y  INCY  Z  INCZ
             |    |    |   |    |   |    |   |
CALL SZAXPY( 5 , 2.0 , X , 1  , Y , 2  , Z , 1  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (1.0, . , 1.0, . , 1.0, . , 1.0, . , 1.0)
```

Output

```
Z        =  (3.0, 5.0, 7.0, 9.0, 11.0)
```

Example 2: This example shows vectors ***x*** and ***y*** having strides of opposite sign, and an output vector ***z*** having a positive stride. For ***y***, which has negative stride, processing begins at element `Y(5)`, which is 1.0.

Call Statement and Input

```
             N  ALPHA  X  INCX  Y   INCY  Z  INCZ
             |    |    |   |    |    |    |   |
CALL SZAXPY( 5 , 2.0 , X , 1  , Y , -1  , Z , 2  )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output

```
Z       =  (3.0, . , 6.0, . , 9.0, . , 12.0, . , 15.0)
```

Example 3: This example shows a vector, ***x***, with 0 stride, and a vector, ***z***, with negative stride. ***x*** is treated like a vector of length *n*, all of whose elements are the same as the single element in ***x***. For vector ***z***, results are stored beginning in element `Z(5)`.

Call Statement and Input

```
            N  ALPHA  X  INCX  Y  INCY  Z   INCZ
            |    |    |   |    |   |    |    |
CALL SZAXPY( 5 , 2.0 , X , 0  , Y , 1  , Z , -1  )

X       =  (1.0)
Y       =  (5.0, 4.0, 3.0, 2.0, 1.0)
```

Output

```
Z       =  (3.0, 4.0, 5.0, 6.0, 7.0)
```

Example 4: This example shows a vector, ***y***, with 0 stride. ***y*** is treated like a vector of length *n*, all of whose elements are the same as the single element in ***y***.

Call Statement and Input

```
            N  ALPHA  X  INCX  Y  INCY  Z  INCZ
            |    |    |   |    |   |    |   |
CALL SZAXPY( 5 , 2.0 , X , 1  , Y , 0  , Z , 1  )

X       =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y       =  (5.0)
```

Output

```
Z       =  (7.0, 9.0, 11.0, 13.0, 15.0)
```

Example 5: This example shows how SZAXPY can be used to compute a scalar value. In this case, vectors ***x*** and ***y*** contain scalar values. The strides of all vectors, ***x***, ***y***, and ***z***, are 0. The number of elements to be processed, *n*, is 1.

Call Statement and Input

```
            N  ALPHA  X  INCX  Y  INCY  Z  INCZ
            |    |    |   |    |   |    |   |
CALL SZAXPY( 1 , 2.0 , X , 0  , Y , 0  , Z , 0  )

X       =  (1.0)
Y       =  (5.0)
```

Output

```
Z       =  (7.0)
```

Example 6: This example shows vectors ***x*** and ***y***, containing complex numbers and having positive strides.

Call Statement and Input

```
             N  ALPHA  X  INCX  Y  INCY  Z  INCZ
             |    |    |   |    |   |    |   |
CALL CZAXPY( 3 ,ALPHA, X , 1  , Y , 2  , Z , 1  )

ALPHA    =  (2.0, 3.0)
X        =  ((1.0, 2.0), (2.0, 0.0), (3.0, 5.0))
Y        =  ((1.0, 1.0), . , (0.0, 2.0), . , (5.0, 4.0))
```

Output

```
Z        =  ((-3.0, 8.0), (4.0, 8.0), (-4.0, 23.0))
```

Sparse Vector-Scalar Subprograms

This section contains the sparse vector-scalar subprogram descriptions.

SSCTR, DSCTR, CSCTR, ZSCTR—Scatter the Elements of a Sparse Vector X in Compressed-Vector Storage Mode into Specified Elements of a Sparse Vector Y in Full-Vector Storage Mode

These subprograms scatter the elements of sparse vector ***x***, stored in compressed-vector storage mode, into specified elements of sparse vector ***y***, stored in full-vector storage mode.

Table 57. Data Types

x, ***y***	**Subprogram**
Short-precision real	SSCTR
Long-precision real	DSCTR
Short-precision complex	CSCTR
Long-precision complex	ZSCTR

Syntax

Fortran	CALL SSCTR | DSCTR | CSCTR | ZSCTR (*nz*, *x*, *indx*, *y*)
C and C++	ssctr | dsctr | csctr | zsctr (*nz*, *x*, *indx*, *y*);
PL/I	CALL SSCTR | DSCTR | CSCTR | ZSCTR (*nz*, *x*, *indx*, *y*);

On Entry

nz
is the number of elements in sparse vector ***x***, stored in compressed-vector storage mode. Specified as: a fullword integer; $nz \geq 0$.

x
is the sparse vector ***x***, containing *nz* elements, stored in compressed-vector storage mode in an array, referred to as `X`. Specified as: a one-dimensional array of (at least) length *nz*, containing numbers of the data type indicated in Table 57.

indx
is the array, referred to as `INDX`, containing the *nz* indices that indicate the positions of the elements of the sparse vector ***x*** when in full-vector storage mode. They also indicate the positions in vector ***y*** into which the elements are copied.

Specified as: a one-dimensional array of (at least) length *nz*, containing fullword integers.

y
See On Return.

On Return

y
is the sparse vector ***y***, stored in full-vector storage mode, of (at least) length max(`INDX`(*i*)) for $i = 1, nz$, into which *nz* elements of vector ***x*** are copied at positions indicated by the indices array `INDX`.

Returned as: a one-dimensional array of (at least) length max(`INDX`(*i*)) for $i = 1, nz$, containing numbers of the data type indicated in Table 57.

Notes

1. Each value specified in array INDX must be unique; otherwise, results are unpredictable.
2. Vectors ***x*** and ***y*** must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.
3. For a description of how sparse vectors are stored, see "Sparse Vector" on page 61.

Function: The copy is expressed as follows:

$$y_{\mathtt{INDX}(i)} \leftarrow x_i \qquad \text{for } i = 1, nz$$

where:

x is a sparse vector, stored in compressed-vector storage mode.
INDX is the indices array for sparse vector ***x***.
y is a sparse vector, stored in full-vector storage mode.

See reference [29]. If *nz* is 0, no copy is performed.

Error Conditions

Computational Errors: None

Input-Argument Errors: $nz < 0$

Example 1: This example shows how to use SSCTR to copy a sparse vector ***x*** of length 5 into the following vector ***y***, where the elements of array INDX are in ascending order:

```
Y = (6.0, 2.0, 4.0, 7.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0 )
```

Call Statement and Input

```
            NZ  X   INDX   Y
            |   |    |     |
CALL SSCTR( 5 , X , INDX , Y )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
INDX     =  (1, 3, 4, 7, 10)
```

Output

```
Y        =  (1.0, 2.0, 2.0, 3.0, 6.0, 10.0, 4.0, 8.0, 9.0, 5.0)
```

Example 2: This example shows how to use SSCTR to copy a sparse vector ***x*** of length 5 into the following vector ***y***, where the elements of array INDX are in random order:

```
Y = (6.0, 2.0, 4.0, 7.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0 )
```

Call Statement and Input

```
            NZ  X   INDX   Y
            |   |    |     |
CALL SSCTR( 5 , X , INDX , Y )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
INDX     =  (4, 3, 1, 10, 7)
```

Output

```
Y        =  (3.0, 2.0, 2.0, 1.0, 6.0, 10.0, 5.0, 8.0, 9.0, 4.0)
```

Example 3: This example shows how to use CSCTR to copy a sparse vector ***x*** of length 3 into the following vector ***y***, where the elements of array INDX are in random order:

```
Y = ((6.0, 5.0), (-2.0, 3.0), (15.0, 4.0), (9.0, 0.0))
```

Call Statement and Input

```
            NZ  X   INDX   Y
            |   |    |     |
CALL CSCTR( 3 , X , INDX , Y )

X        =  ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
INDX     =  (4, 1, 3)
```

Output

```
Y        =  ((3.0, 4.0), (-2.0, 3.0), (5.0, 6.0), (1.0, 2.0))
```

SGTHR, DGTHR, CGTHR, and ZGTHR—Gather Specified Elements of a Sparse Vector Y in Full-Vector Storage Mode into a Sparse Vector X in Compressed-Vector Storage Mode

These subprograms gather specified elements of vector ***y***, stored in full-vector storage mode, into sparse vector ***x***, stored in compressed-vector storage mode.

Table 58. Data Types

x, ***y***	**Subprogram**
Short-precision real	SGTHR
Long-precision real	DGTHR
Short-precision complex	CGTHR
Long-precision complex	ZGTHR

Syntax

Fortran	CALL SGTHR | DGTHR | CGTHR | ZGTHR (*nz*, *y*, *x*, *indx*)
C and C++	sgthr | dgthr | cgthr | zgthr (*nz*, *y*, *x*, *indx*);
PL/I	CALL SGTHR | DGTHR | CGTHR | ZGTHR (*nz*, *y*, *x*, *indx*);

On Entry

nz
is the number of elements in sparse vector ***x***, stored in compressed-vector storage mode. Specified as: a fullword integer; $nz \geq 0$.

y
is the sparse vector ***y***, stored in full-vector storage mode, of (at least) length max(INDX(*i*)) for $i = 1, nz$, from which *nz* elements are copied from positions indicated by the indices array INDX.

Specified as: a one-dimensional array of (at least) length max(INDX(*i*)) for $i = 1, nz$, containing numbers of the data type indicated in Table 58.

x
See On Return.

indx
is the array, referred to as INDX, containing the *nz* indices that indicate the positions of the elements of the sparse vector ***x*** when in full-vector storage mode. They also indicate the positions in vector ***y*** from which elements are copied.

Specified as: a one-dimensional array of (at least) length *nz*, containing fullword integers.

On Return

x
is the sparse vector ***x***, containing *nz* elements, stored in compressed-vector storage mode in an array, referred to as X, into which are copied the elements of vector ***y*** from positions indicated by the indices array INDX.

Returned as: a one-dimensional array of (at least) length *nz*, containing numbers of the data type indicated in Table 58.

Notes

1. Vectors ***x*** and ***y*** must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.
2. For a description of how sparse vectors are stored, see "Sparse Vector" on page 61.

Function: The copy is expressed as follows:

$$x_i \leftarrow y_{\mathtt{INDX}(i)} \quad \text{for } i = 1, nz$$

where:

x is a sparse vector, stored in compressed-vector storage mode.
`INDX` is the indices array for sparse vector ***x***.
y is a sparse vector, stored in full-vector storage mode.

See reference [29]. If *nz* is 0, no copy is performed.

Error Conditions

Computational Errors: None

Input-Argument Errors: $nz < 0$

Example 1: This example shows how to use SGTHR to copy specified elements of a vector ***y*** into a sparse vector ***x*** of length 5, where the elements of array `INDX` are in ascending order.

Call Statement and Input

```
            NZ  Y   X   INDX
            |   |   |    |
CALL SGTHR( 5 , Y , X , INDX )

Y       =  (6.0, 2.0, 4.0, 7.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0)
INDX    =  (1, 3, 4, 7, 9)
```

Output

```
X       =  (6.0, 4.0, 7.0, -2.0, 9.0)
```

Example 2: This example shows how to use SGTHR to copy specified elements of a vector ***y*** into a sparse vector ***x*** of length 5, where the elements of array `INDX` are in random order. (Note that the element 0.0 occurs in output vector ***x***. This does not produce an error.)

Call Statement and Input

```
            NZ  Y   X   INDX
            |   |   |    |
CALL SGTHR( 5 , Y , X , INDX )

Y       =  (6.0, 2.0, 4.0, 7.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0)
INDX    =  (4, 3, 1, 10, 7)
```

Output

```
X        =  (7.0, 4.0, 6.0, 0.0, -2.0)
```

Example 3: This example shows how to use CGTHR to copy specified elements of a vector, ***y***, into a sparse vector, ***x***, of length 3, where the elements of array INDX are in random order.

Call Statement and Input

```
            NZ  Y   X   INDX
            |   |   |    |
CALL CGTHR( 3 , Y , X , INDX )

Y        =  ((6.0, 5.0), (-2.0, 3.0), (15.0, 4.0), (9.0, 0.0))
INDX     =  (4, 1, 3)
```

Output

```
X        =  ((9.0, 0.0), (6.0, 5.0), (15.0, 4.0))
```

SGTHRZ, DGTHRZ, CGTHRZ, and ZGTHRZ—Gather Specified Elements of a Sparse Vector Y in Full-Vector Mode into a Sparse Vector X in Compressed-Vector Mode, and Zero the Same Specified Elements of Y

These subprograms gather specified elements of sparse vector ***y***, stored in full-vector storage mode, into sparse vector ***x***, stored in compressed-vector storage mode, and zero the same specified elements of vector ***y***.

Table 59. Data Types

x, ***y***	**Subprogram**
Short-precision real	SGTHRZ
Long-precision real	DGTHRZ
Short-precision complex	CGTHRZ
Long-precision complex	ZGTHRZ

Syntax

Fortran	CALL SGTHRZ | DGTHRZ | CGTHRZ | ZGTHRZ (*nz*, *y*, *x*, *indx*)
C and C++	sgthrz | dgthrz | cgthrz | zgthrz (*nz*, *y*, *x*, *indx*);
PL/I	CALL SGTHRZ | DGTHRZ | CGTHRZ | ZGTHRZ (*nz*, *y*, *x*, *indx*);

On Entry

nz
is the number of elements in sparse vector ***x***, stored in compressed-vector storage mode. Specified as: a fullword integer; $nz \geq 0$.

y
is the sparse vector ***y***, stored in full-vector storage mode, of (at least) length max(INDX(*i*)) for $i = 1, nz$, from which *nz* elements are copied from positions indicated by the indices array INDX.

Specified as: a one-dimensional array of (at least) length max(INDX(*i*)) for $i = 1, nz$, containing numbers of the data type indicated in Table 59.

x
See On Return.

indx
is the array, referred to as INDX, containing the *nz* indices that indicate the positions of the elements of the sparse vector ***x*** when in full-vector storage mode. They also indicate the positions in vector ***y*** from which elements are copied then set to zero.

Specified as: a one-dimensional array of (at least) length *nz*, containing fullword integers.

On Return

y
is the sparse vector ***y***, stored in full-vector storage mode, of (at least) length max(INDX(*i*)) for $i = 1, nz$, whose elements are set to zero at positions indicated by the indices array INDX.

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 59.

x

is the sparse vector ***x***, containing *nz* elements stored in compressed-vector storage mode in an array, referred to as X, into which are copied the elements of vector ***y*** from positions indicated by the indices array INDX.

Returned as: a one-dimensional array of (at least) length *nz*, containing numbers of the data type indicated in Table 59 on page 286.

Notes

1. Each value specified in array INDX must be unique; otherwise, results are unpredictable.
2. Vectors ***x*** and ***y*** must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.
3. For a description of how sparse vectors are stored, see "Sparse Vector" on page 61.

Function: The copy is expressed as follows:

$x_i \leftarrow y_{\mathtt{INDX}(i)}$
$y_{\mathtt{INDX}(i)} \leftarrow 0.0$ (for SGTHRZ and DGTHRZ)
$y_{\mathtt{INDX}(i)} \leftarrow (0.0,0.0)$ (for CGTHRZ and ZGTHRZ)
for $i = 1,nz$

where:

x is a sparse vector, stored in compressed-vector storage mode.
INDX is the indices array for sparse vector ***x***.
y is a sparse vector, stored in full-vector storage mode.

See reference [29]. If *nz* is 0, no computation is performed.

Error Conditions

Computational Errors: None

Input-Argument Errors: $nz < 0$

Example 1: This example shows how to use SGTHRZ to copy specified elements of a vector ***y*** into a sparse vector ***x*** of length 5, where the elements of array INDX are in ascending order.

Call Statement and Input

```
             NZ  Y   X   INDX
             |   |   |    |
CALL SGTHRZ( 5 , Y , X , INDX )

Y        =  (6.0, 2.0, 4.0, 7.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0)
INDX     =  (1, 3, 4, 7, 9)
```

Output

```
Y        =  (0.0, 2.0, 0.0, 0.0, 6.0, 10.0, 0.0, 8.0, 0.0, 0.0)
X        =  (6.0, 4.0, 7.0, -2.0, 9.0)
```

Example 2: This example shows how to use SGTHRZ to copy specified elements of a vector ***y*** into a sparse vector ***x*** of length 5, where the elements of array INDX are in random order. (Note that the element 0.0 occurs in output vector ***x***. This does not produce an error.)

Call Statement and Input

```
             NZ  Y   X   INDX
             |   |   |    |
CALL SGTHRZ( 5 , Y , X , INDX )

Y        =  (6.0, 2.0, 4.0, 7.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0)
INDX     =  (4, 3, 1, 10, 7)
```

Output

```
Y        =  (0.0, 2.0, 0.0, 0.0, 6.0, 10.0, 0.0, 8.0, 9.0, 0.0)
X        =  (7.0, 4.0, 6.0, 0.0, -2.0)
```

Example 3: This example shows how to use CGTHRZ to copy specified elements of a vector ***y*** into a sparse vector ***x*** of length 3, where the elements of array INDX are in random order.

Call Statement and Input

```
             NZ  Y   X   INDX
             |   |   |    |
CALL CGTHRZ( 3 , Y , X , INDX )

Y        =  ((6.0, 5.0), (-2.0, 3.0), (15.0, 4.0), (9.0, 0.0))
INDX     =  (4, 1, 3)
```

Output

```
Y        =  ((0.0, 0.0), (-2.0, 3.0), (0.0, 0.0), (0.0, 0.0))
X        =  ((9.0, 0.0), (6.0, 5.0), (15.0, 4.0))
```

SAXPYI, DAXPYI, CAXPYI, and ZAXPYI—Multiply a Sparse Vector X in Compressed-Vector Storage Mode by a Scalar, Add to a Sparse Vector Y in Full-Vector Storage Mode, and Store in the Vector Y

These subprograms multiply sparse vector ***x***, stored in compressed-vector storage mode, by scalar α, add it to sparse vector ***y***, stored in full-vector storage mode, and store the result in vector ***y***.

Table 60. Data Types

α, ***x***, ***y***	**Subprogram**
Short-precision real	SAXPYI
Long-precision real	DAXPYI
Short-precision complex	CAXPYI
Long-precision complex	ZAXPYI

Syntax

Fortran	CALL SAXPYI | DAXPYI | CAXPYI | ZAXPYI (*nz*, *alpha*, *x*, *indx*, *y*)
C and C++	saxpyi | daxpyi | caxpyi | zaxpyi (*nz*, *alpha*, *x*, *indx*, *y*);
PL/I	CALL SAXPYI | DAXPYI | CAXPYI | ZAXPYI (*nz*, *alpha*, *x*, *indx*, *y*);

On Entry

nz
is the number of elements in sparse vector ***x***, stored in compressed-vector storage mode. Specified as: a fullword integer; $nz \geq 0$.

alpha
is the scalar α. Specified as: a number of the data type indicated in Table 60.

x
is the sparse vector ***x***, containing *nz* elements, stored in compressed-vector storage mode in an array, referred to as `X`. Specified as: a one-dimensional array of (at least) length *nz*, containing numbers of the data type indicated in Table 60.

indx
is the array, referred to as `INDX`, containing the *nz* indices that indicate the positions of the elements of the sparse vector ***x*** when in full-vector storage mode. They also indicate the positions of the elements in vector ***y*** that are used in the computation.

Specified as: a one-dimensional array of (at least) length *nz*, containing fullword integers.

y
is the sparse vector ***y***, stored in full-vector storage mode, of (at least) length max(`INDX`(*i*)) for $i = 1, nz$. Specified as: a one-dimensional array of (at least) length max(`INDX`(*i*)) for $i = 1, nz$, containing numbers of the data type indicated in Table 60.

On Return

y
is the sparse vector ***y***, stored in full-vector storage mode, of (at least) length max(INDX(*i*)) for $i = 1, nz$ containing the results of the computation, stored at positions indicated by the indices array INDX.

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 60 on page 289.

Notes

1. Each value specified in array INDX must be unique; otherwise, results are unpredictable.
2. Vectors ***x*** and ***y*** must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.
3. For a description of how sparse vectors are stored, see "Sparse Vector" on page 61.

Function: The computation is expressed as follows:

$$y_{INDX(i)} \leftarrow y_{INDX(i)} + \alpha x_i \quad \text{for } i = 1, nz$$

where:

x is a sparse vector, stored in compressed-vector storage mode.
INDX is the indices array for sparse vector ***x***.
y is a sparse vector, stored in full-vector storage mode.

See reference [29]. If α or *nz* is zero, no computation is performed. For SAXPYI and CAXPYI, intermediate results are accumulated in long-precision.

Error Conditions

Computational Errors: None

Input-Argument Errors: $nz < 0$

Example 1: This example shows how to use SAXPYI to perform a computation using a sparse vector ***x*** of length 5, where the elements of array INDX are in ascending order.

Call Statement and Input

```
             NZ ALPHA  X   INDX   Y
             |    |    |    |     |
CALL SAXPYI( 5 , 2.0 , X , INDX , Y )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
INDX     =  (1, 3, 4, 7, 10)
Y        =  (1.0, 5.0, 4.0, 3.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0)
```

Output

```
Y        =  (3.0, 5.0, 8.0, 9.0, 6.0, 10.0, 6.0, 8.0, 9.0, 10.0)
```

Example 2: This example shows how to use SAXPYI to perform a computation using a sparse vector ***x*** of length 5, where the elements of array INDX are in random order.

Call Statement and Input

```
             NZ ALPHA  X   INDX   Y
             |    |    |    |     |
CALL SAXPYI( 5 , 2.0 , X , INDX , Y )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
INDX     =  (4, 3, 1, 10, 7)
Y        =  (1.0, 5.0, 4.0, 3.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0)
```

Output

```
Y        =  (7.0, 5.0, 8.0, 5.0, 6.0, 10.0, 8.0, 8.0, 9.0, 8.0)
```

Example 3: This example shows how to use CAXPYI to perform a computation using a sparse vector ***x*** of length 3, where the elements of array INDX are in random order.

Call Statement and Input

```
             NZ  ALPHA   X   INDX   Y
             |     |     |    |     |
CALL CAXPYI( 3 , ALPHA , X , INDX , Y )

ALPHA    =  (2.0, 3.0)
X        =  ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
INDX     =  (4, 1, 3)
Y        =  ((6.0, 5.0), (-2.0, 3.0), (15.0, 4.0), (9.0, 0.0))
```

Output

```
Y        =  ((0.0, 22.0), (-2.0, 3.0), (7.0, 31.0), (5.0, 7.0))
```

SDOTI, DDOTI, CDOTUI, ZDOTUI, CDOTCI, and ZDOTCI—Dot Product of a Sparse Vector X in Compressed-Vector Storage Mode and a Sparse Vector Y in Full-Vector Storage Mode

SDOTI, DDOTI, CDOTUI, and ZDOTUI compute the dot product of sparse vector ***x***, stored in compressed-vector storage mode, and full vector ***y***, stored in full-vector storage mode.

CDOTCI and ZDOTCI compute the dot product of the complex conjugate of sparse vector ***x***, stored in compressed-vector storage mode, and full vector ***y***, stored in full-vector storage mode.

Table 61. Data Types

x, ***y***, **Result**	**Subprogram**
Short-precision real	SDOTI
Long-precision real	DDOTI
Short-precision complex	CDOTUI
Long-precision complex	ZDOTUI
Short-precision complex	CDOTCI
Long-precision complex	ZDOTCI

Syntax

Fortran	SDOTI | DDOTI | CDOTUI | ZDOTUI | CDOTCI | ZDOTCI (*nz*, *x*, *indx*, *y*)
C and C++	sdoti | ddoti | cdotui | zdotui | cdotci | zdotci (*nz*, *x*, *indx*, *y*);
PL/I	SDOTI | DDOTI | CDOTUI | ZDOTUI | CDOTCI | ZDOTCI (*nz*, *x*, *indx*, *y*);

On Entry

nz
is the number of elements in sparse vector ***x***, stored in compressed-vector storage mode. Specified as: a fullword integer; $nz \geq 0$.

x
is the sparse vector ***x***, containing *nz* elements, stored in compressed-vector storage mode in an array, referred to as `X`. Specified as: a one-dimensional array of (at least) length *nz*, containing numbers of the data type indicated in Table 61.

indx
is the array, referred to as `INDX`, containing the *nz* indices that indicate the positions of the elements of the sparse vector ***x*** when in full-vector storage mode. They also indicate the positions of elements in vector ***y*** that are used in the computation.

Specified as: a one-dimensional array of (at least) length *nz*, containing fullword integers.

y
is the sparse vector ***y***, stored in full-vector storage mode, of (at least) length max(`INDX`(*i*)) for $i = 1, nz$. Specified as: a one-dimensional array of (at least) length

max(INDX(*i*)) for *i* = 1, *nz*, containing numbers of the data type indicated in Table 61.

On Return

Function value

is the result of the dot product computation.

Returned as: a number of the data type indicated in Table 61 on page 292.

Note

1. Declare this function in your program as returning a value of the data type indicated in Table 61 on page 292.
2. For a description of how sparse vectors are stored, see "Sparse Vector" on page 61.

Function: For SDOTI, DDOTI, CDOTUI, and ZDOTUI, the dot product computation is expressed as follows:

$$\sum_{i=1}^{nz} x_i y_{\mathrm{INDX}(i)} = x_1 y_{\mathrm{INDX}(1)} + x_2 y_{\mathrm{INDX}(2)} + \ldots + x_{nz} y_{\mathrm{INDX}(nz)}$$

For CDOTCI and ZDOTCI, the dot product computation is expressed as follows:

$$\sum_{i=1}^{nz} \bar{x}_i y_{\mathrm{INDX}(i)} = \bar{x}_1 y_{\mathrm{INDX}(1)} + \bar{x}_2 y_{\mathrm{INDX}(2)} + \ldots + \bar{x}_{nz} y_{\mathrm{INDX}(nz)}$$

where:

x is a sparse vector, stored in compressed-vector storage mode.

$\bar{x}$ is the complex conjugate of a sparse vector, stored in compressed-vector storage mode.

INDX is the indices array for sparse vector ***x***.

y is a sparse vector, stored in full-vector storage mode.

See reference [29]. The result is returned as the function value. If *nz* is 0, then zero is returned as the value of the function.

For SDOTI, CDOTUI, and CDOTCI, intermediate results are accumulated in long-precision.

Error Conditions

Computational Errors: None

Input-Argument Errors: $nz < 0$

Example 1: This example shows how to use SDOTI to compute a dot product using a sparse vector ***x*** of length 5, where the elements of array INDX are in ascending order.

Function Reference and Input

```
              NZ  X   INDX   Y
              |   |    |     |
DOTT = SDOTI( 5 , X , INDX , Y )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
INDX     =  (1, 3, 4, 7, 10)
Y        =  (1.0, 5.0, 4.0, 3.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0)
```

Output

```
DOTT     =  (1.0 + 8.0 + 9.0 -8.0 + 0.0) = 10.0
```

Example 2: This example shows how to use SDOTI to compute a dot product using a sparse vector ***x*** of length 5, where the elements of array INDX are in random order.

Function Reference and Input

```
              NZ  X   INDX   Y
              |   |    |     |
DOTT = SDOTI( 5 , X , INDX , Y )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
INDX     =  (4, 3, 1, 10, 7)
Y        =  (1.0, 5.0, 4.0, 3.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0)
```

Output

```
DOTT     =  (3.0 + 8.0 + 3.0 + 0.0 -10.0) = 4.0
```

Example 3: This example shows how to use CDOTUI to compute a dot product using a sparse vector ***x*** of length 3, where the elements of array INDX are in ascending order.

Function Reference and Input

```
               NZ  X   INDX   Y
               |   |    |     |
DOTT = CDOTUI( 3 , X , INDX , Y )

X        =  ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
INDX     =  (1, 3, 4)
Y        =  ((6.0, 5.0), (-2.0, 3.0), (15.0, 4.0), (9.0, 0.0))
```

Output

```
DOTT     =  (70.0, 143.0)
```

Example 4: This example shows how to use CDOTCI to compute a dot product using the complex conjugate of a sparse vector ***x*** of length 3, where the elements of array INDX are in random order.

Function Reference and Input

```
               NZ  X   INDX   Y
               |   |    |     |
DOTT = CDOTCI( 3 , X , INDX , Y )

X        =  ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
INDX     =  (4, 1, 3)
Y        =  ((6.0, 5.0), (-2.0, 3.0), (15.0, 4.0), (9.0, 0.0))
```

Output

```
DOTT     =  (146.0, -97.0)
```

Matrix-Vector Subprograms

This section contains the matrix-vector subprogram descriptions.

SGEMV, DGEMV, CGEMV, ZGEMV, SGEMX, DGEMX, SGEMTX, and DGEMTX—Matrix-Vector Product for a General Matrix, Its Transpose, or Its Conjugate Transpose

SGEMV and DGEMV compute the matrix-vector product for either a real general matrix or its transpose, using the scalars α and β, vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, and matrix $\boldsymbol{A}$ or its transpose:

$$\boldsymbol{y} \leftarrow \beta\boldsymbol{y}+\alpha\boldsymbol{A}\boldsymbol{x}$$

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

CGEMV and ZGEMV compute the matrix-vector product for either a complex general matrix, its transpose, or its conjugate transpose, using the scalars α and β, vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, and matrix $\boldsymbol{A}$, its transpose, or its conjugate transpose:

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

SGEMX and DGEMX compute the matrix-vector product for a real general matrix, using the scalar α, vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, and matrix $\boldsymbol{A}$:

$$\boldsymbol{y} \leftarrow \boldsymbol{y}+\alpha\boldsymbol{A}\boldsymbol{x}$$

SGEMTX and DGEMTX compute the matrix-vector product for the transpose of a real general matrix, using the scalar α, vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, and the transpose of matrix $\boldsymbol{A}$:

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

Table 62. Data Types

α, β, $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{A}$	**Subprogram**
Short-precision real	SGEMV, SGEMX, and SGEMTX
Long-precision real	DGEMV, DGEMX, and DGEMTX
Short-precision complex	CGEMV
Long-precision complex	ZGEMV

Note: SGEMV and DGEMV are Level 2 BLAS subroutines. It is suggested that these subroutines be used instead of SGEMX, DGEMX, SGEMTX, and DGEMTX, which are provided only for compatibility with earlier releases of ESSL.

Syntax

Fortran	CALL SGEMV | DGEMV | CGEMV | ZGEMV (*transa*, *m*, *n*, *alpha*, *a*, *lda*, *x*, *incx*, *beta*, *y*, *incy*) CALL SGEMX | DGEMX | SGEMTX | DGEMTX (*m*, *n*, *alpha*, *a*, *lda*, *x*, *incx*, *y*, *incy*)
C and C++	sgemv | dgemv | cgemv | zgemv (*transa*, *m*, *n*, *alpha*, *a*, *lda*, *x*, *incx*, *beta*, *y*, *incy*); sgemx | dgemx | sgemtx | dgemtx (*m*, *n*, *alpha*, *a*, *lda*, *x*, *incx*, *y*, *incy*);

PL/I	CALL SGEMV | DGEMV | CGEMV | ZGEMV (*transa*, *m*, *n*, *alpha*, *a*, *lda*, *x*, *incx*, *beta*, *y*, *incy*); CALL SGEMX | DGEMX | SGEMTX | DGEMTX (*m*, *n*, *alpha*, *a*, *lda*, *x*, *incx*, *y*, *incy*);

On Entry

transa
indicates the form of matrix ***A*** to use in the computation, where:

If *transa* = 'N', ***A*** is used in the computation.

If *transa* = 'T', $\boldsymbol{A}^{T}$ is used in the computation.

If *transa* = 'C', $\boldsymbol{A}^{H}$ is used in the computation.

Specified as: a single character. It must be 'N', 'T', or 'C'.

m
is the number of rows in matrix ***A***, and:

For SGEMV, DGEMV, CGEMV, and ZGEMV:
If *transa* = 'N', it is the length of vector ***y***.
If *transa* = 'T' or 'C', it is the length of vector ***x***.

For SGEMX and DGEMX, it is the length of vector ***y***.

For SGEMTX and DGEMTX, it is the length of vector ***x***.

Specified as: a fullword integer; $0 \le m \le lda$.

n
is the number of columns in matrix ***A***, and:

For SGEMV, DGEMV, CGEMV, and ZGEMV:
If *transa* = 'N', it is the length of vector ***x***.
If *transa* = 'T' or 'C', it is the length of vector ***y***.

For SGEMX and DGEMX, it is the length of vector ***x***.

For SGEMTX and DGEMTX, it is the length of vector ***y***.

Specified as: a fullword integer; $n \ge 0$.

alpha
is the scaling constant α. Specified as: a number of the data type indicated in Table 62 on page 297.

a
is the *m* by *n* matrix ***A***, where:

For SGEMV, DGEMV, CGEMV, and ZGEMV:
If *transa* = 'N', ***A*** is used in the computation.
If *transa* = 'T', $\boldsymbol{A}^{T}$ is used in the computation.
If *transa* = 'C', $\boldsymbol{A}^{H}$ is used in the computation.

For SGEMX and DGEMX, ***A*** is used in the computation.

For SGEMTX and DGEMTX, $\boldsymbol{A}^{T}$ is used in the computation.

Note: No data should be moved to form $\boldsymbol{A}^{T}$ or $\boldsymbol{A}^{H}$; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 62 on page 297.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; *lda* > 0 and *lda* ≥ *m*.

x
is the vector ***x***, where:

For SGEMV, DGEMV, CGEMV, and ZGEMV:
If *transa* = 'N', it has length *n*.
If *transa* = 'T' or 'C', it has length *m*.

For SGEMX and DGEMX, it has length *n*.

For SGEMTX and DGEMTX, it has length *m*.

Specified as: a one-dimensional array, containing numbers of the data type indicated in Table 62 on page 297, where:

For SGEMV, DGEMV, CGEMV, and ZGEMV:
If *transa* = 'N', it must have at least $1+(n-1)|incx|$ elements.
If *transa* = 'T' or 'C', it must have at least $1+(m-1)|incx|$ elements.

For SGEMX and DGEMX, it must have at least $1+(n-1)|incx|$ elements.

For SGEMTX and DGEMTX, it must have at least $1+(m-1)|incx|$ elements.

incx
is the stride for vector ***x***. Specified as: a fullword integer; It can have any value.

beta
is the scaling constant β. Specified as: a number of the data type indicated in Table 62 on page 297.

y
is the vector ***y***, where:

For SGEMV, DGEMV, CGEMV, and ZGEMV:
If *transa* = 'N', it has length *m*.
If *transa* = 'T' or 'C', it has length *n*.

For SGEMX and DGEMX, it has length *m*.

For SGEMTX and DGEMTX, it has length *n*.

Specified as: a one-dimensional array, containing numbers of the data type indicated in Table 62 on page 297, where:

For SGEMV, DGEMV, CGEMV, and ZGEMV:
If *transa* = 'N', it must have at least $1+(m-1)|incy|$ elements.
If *transa* = 'T' or 'C', it must have at least $1+(n-1)|incy|$ elements.

For SGEMX and DGEMX, it must have at least $1+(m-1)|incy|$ elements.

For SGEMTX and DGEMTX, it must have at least $1+(n-1)|incy|$ elements.

incy
is the stride for vector ***y***. Specified as: a fullword integer; *incy* > 0 or *incy* < 0.

On Return

y
is the vector ***y***, containing the result of the computation, where:

For SGEMV, DGEMV, CGEMV, and ZGEMV:
If *transa* = 'N', it has length *m*.
If *transa* = 'T' or 'C', it has length *n*.

For SGEMX and DGEMX, it has length *m*.

For SGEMTX and DGEMTX, it has length *n*.

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 62 on page 297.

Notes

1. For SGEMV and DGEMV, if you specify 'C' for the *transa* argument, it is interpreted as though you specified 'T'.
2. The SGEMV, DGEMV, CGEMV, and ZGEMV subroutines accept lowercase letters for the *transa* argument.
3. In the SGEMV, DGEMV, CGEMV, and ZGEMV subroutines, *incx* = 0 is valid; however, the Level 2 BLAS standard considers *incx* = 0 to be invalid. See references [34] and [35].
4. Vector ***y*** must have no common elements with matrix ***A*** or vector ***x***; otherwise, results are unpredictable. See "Concepts" on page 55.

Function: The possible computations that can be performed by these subroutines are described in the following sections. Varying implementation techniques are used for this computation to improve performance. As a result, accuracy of the computational result may vary for different computations.

For SGEMV, CGEMV, SGEMX, and SGEMTX, intermediate results are accumulated in long precision. Occasionally, for performance reasons, these intermediate results are stored.

See references [34], [35], [38], [46], and [73]. No computation is performed if *m* or *n* is 0 or if α is zero and β is one.

General Matrix: For SGEMV, DGEMV, CGEMV, and ZGEMV, the matrix-vector product for a general matrix:

$$\boldsymbol{y} \leftarrow \beta\boldsymbol{y}+\alpha\boldsymbol{A}\boldsymbol{x}$$

is expressed as follows:

$$\begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_m \end{bmatrix} \leftarrow \beta \begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_m \end{bmatrix} + \alpha \begin{bmatrix} a_{11} & \ldots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{m1} & \ldots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix}$$

For SGEMX and DGEMX, the matrix-vector product for a real general matrix:

$$\boldsymbol{y} \leftarrow \boldsymbol{y}+\alpha\boldsymbol{A}\boldsymbol{x}$$

is expressed as follows:

$$\begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_m \end{bmatrix} \leftarrow \begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_m \end{bmatrix} + \alpha \begin{bmatrix} a_{11} & \ldots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{m1} & \ldots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix}$$

In these expressions:

$\boldsymbol{y}$ is a vector of length m.
α is a scalar.
β is a scalar.
$\boldsymbol{A}$ is an m by n matrix.
$\boldsymbol{x}$ is a vector of length n.

Transpose of a General Matrix: For SGEMV, DGEMV, CGEMV and ZGEMV, the matrix-vector product for the transpose of a general matrix:

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

is expressed as follows:

$$\begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix} \leftarrow \beta \begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix} + \alpha \begin{bmatrix} a_{11} & \ldots & a_{m1} \\ . & & . \\ . & & . \\ . & & . \\ a_{1n} & \ldots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_m \end{bmatrix}$$

For SGEMTX and DGEMTX, the matrix-vector product for the transpose of a real general matrix:

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

is expressed as follows:

$$\begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix} \leftarrow \begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix} + \alpha \begin{bmatrix} a_{11} & \ldots & a_{m1} \\ . & & . \\ . & & . \\ . & & . \\ a_{1n} & \ldots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_m \end{bmatrix}$$

In these expressions:

$\boldsymbol{y}$ is a vector of length n.
α is a scalar.

β is a scalar.
A^T is the transpose of matrix A, where A is an m by n matrix.
x is a vector of length m.

Conjugate Transpose of a General Matrix: For CGEMV and ZGEMV, the matrix-vector product for the conjugate transpose of a general matrix:

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

is expressed as follows:

$$\begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix} \leftarrow \beta \begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix} + \alpha \begin{bmatrix} \overline{a}_{11} & \ldots & \overline{a}_{m1} \\ . & & . \\ . & & . \\ . & & . \\ \overline{a}_{1n} & \ldots & \overline{a}_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_m \end{bmatrix}$$

where:

y is a vector of length n.
α is a scalar.
β is a scalar.
A^H is the conjugate transpose of matrix A, where A is an m by n matrix.
x is a vector of length m.

Error Conditions

Resource Errors: Unable to allocate internal work area (for SGEMV, DGEMV, CGEMV, and ZGEMV).

Computational Errors: None

Input-Argument Errors

1. *transa* ≠ 'N', 'T', or 'C'
2. $m < 0$
3. $m > lda$
4. $n < 0$
5. $lda \leq 0$
6. $incy = 0$

Example 1: This example shows the computation for `TRANSA` equal to `'N'`, where the real general matrix A is used in the computation. Because *lda* is 10 and *n* is 3, array `A` must be declared as `A(E1:E2,F1:F2)`, where `E2-E1+1=10` and `F2-F1+1` ≥ 3. In this example, array `A` is declared as `A(1:10,0:2)`.

Call Statement and Input

```
           TRANSA M   N  ALPHA    A      LDA  X  INCX BETA   Y  INCY
             |    |   |    |      |       |   |   |     |    |   |
CALL SGEMV( 'N' , 4 , 3 , 1.0 , A(1,0) , 10 , X , 1  , 1.0 , Y , 2  )
```

```
     ⎡                 ⎤
     ⎢ 1.0  2.0  3.0 ⎥
     ⎢ 2.0  2.0  4.0 ⎥
     ⎢ 3.0  2.0  2.0 ⎥
     ⎢ 4.0  2.0  1.0 ⎥
A =  ⎢  .    .    .  ⎥
     ⎢  .    .    .  ⎥
     ⎢  .    .    .  ⎥
     ⎢  .    .    .  ⎥
     ⎢  .    .    .  ⎥
     ⎣  .    .    .  ⎦

X        =  (3.0, 2.0, 1.0)
Y        =  (4.0, . , 5.0, . , 2.0, . , 3.0)
```

Output

```
Y        =  (14.0, . , 19.0, . , 17.0, . , 20.0)
```

Example 2: This example shows the computation for `TRANSA` equal to `'T'`, where the transpose of the real general matrix ***A*** is used in the computation. Array `A` must follow the same rules as given in Example 1. In this example, array `A` is declared as `A(-1:8,1:3)`.

Call Statement and Input

```
           TRANSA M   N  ALPHA     A     LDA   X  INCX  BETA  Y  INCY
             |    |   |    |       |      |    |   |     |    |   |
CALL SGEMV( 'T' , 4 , 3 , 1.0 , A(-1,1) , 10 , X , 1  , 2.0 , Y , 2  )

A        =(same as input A in Example 1)
X        =  (3.0, 2.0, 1.0, 4.0)
Y        =  (1.0, . , 2.0, . , 3.0)
```

Output

```
Y        =  (28.0, . , 24.0, . , 29.0)
```

Example 3: This example shows the computation for `TRANSA` equal to `'N'`, where the complex general matrix ***A*** is used in the computation.

Call Statement and Input

```
           TRANSA M   N   ALPHA   A   LDA  X  INCX  BETA   Y  INCY
             |    |   |     |     |    |   |   |     |     |   |
CALL CGEMV( 'N' , 5 , 3 , ALPHA , A , 10 , X , 1  , BETA , Y , 1  )

ALPHA    =  (1.0, 0.0)
```

```
    ⎡                                    ⎤
    ⎢ (1.0, 2.0)  (3.0, 5.0)  (2.0, 0.0) ⎥
    ⎢ (2.0, 3.0)  (7.0, 9.0)  (4.0, 8.0) ⎥
    ⎢ (7.0, 4.0)  (1.0, 4.0)  (6.0, 0.0) ⎥
    ⎢ (8.0, 2.0)  (2.0, 5.0)  (8.0, 0.0) ⎥
A = ⎢ (9.0, 1.0)  (3.0, 6.0)  (1.0, 0.0) ⎥
    ⎢     .           .           .      ⎥
    ⎢     .           .           .      ⎥
    ⎢     .           .           .      ⎥
    ⎢     .           .           .      ⎥
    ⎣     .           .           .      ⎦

X        =  ((1.0, 2.0), (4.0, 0.0), (1.0, 1.0))
BETA     =  (1.0, 0.0)
Y        =  ((1.0, 2.0), (4.0, 0.0), (1.0, -1.0), (3.0, 4.0),
             (2.0, 0.0))
```

Output

```
Y        =  ((12.0, 28.0), (24.0, 55.0), (10.0, 39.0), (23.0, 50.0),
             (22.0, 44.0))
```

Example 4: This example shows the computation for TRANSA equal to 'T', where the transpose of complex general matrix $\boldsymbol{A}$ is used in the computation. Because β is zero, the result of the computation is $\alpha \boldsymbol{A}^T \boldsymbol{x}$

Call Statement and Input

```
           TRANSA M   N   ALPHA   A   LDA  X  INCX  BETA   Y  INCY
             |    |   |     |     |    |   |   |     |     |   |
CALL CGEMV( 'T' , 5 , 3 , ALPHA , A , 10 , X , 1  , BETA , Y , 1  )

ALPHA    =  (1.0, 0.0)
A        =(same as input A in Example 3)
X        =  ((1.0, 2.0), (4.0, 0.0), (1.0, 1.0), (3.0, 4.0),
             (2.0, 0.0))
BETA     =  (0.0, 0.0)
Y        =(not relevant)
```

Output

```
Y        =  ((42.0, 67.0), (10.0, 87.0), (50.0, 74.0))
```

Example 5: This example shows the computation for TRANSA equal to 'C', where the conjugate transpose of the complex general matrix $\boldsymbol{A}$ is used in the computation.

Call Statement and Input

```
           TRANSA M   N   ALPHA   A   LDA  X  INCX  BETA   Y  INCY
             |    |   |     |     |    |   |   |     |     |   |
CALL CGEMV( 'C' , 5 , 3 , ALPHA , A , 10 , X , 1  , BETA , Y , 1  )

ALPHA    =  (-1.0, 0.0)
A        =(same as input A in Example 3)
X        =  ((1.0, 2.0), (4.0, 0.0), (1.0, 1.0), (3.0, 4.0),
             (2.0, 0.0))
```

```
BETA     =  (1.0, 0.0)
Y        =  ((1.0, 2.0), (4.0, 0.0), (1.0, -1.0))
```

Output

```
Y        =  ((-73.0, -13.0), (-74.0, 57.0), (-49.0, -11.0))
```

Example 6: This example shows a matrix, ***A***, contained in a larger array, A. The strides of vectors ***x*** and ***y*** are positive. Because *lda* is 10 and *n* is 3, array A must be declared as A(E1:E2,F1:F2), where E2-E1+1=10 and F2-F1+1 ≥ 3. For this example, array A is declared as A(1:10,0:2).

Call Statement and Input

```
           M   N  ALPHA    A     LDA   X  INCX  Y  INCY
           |   |    |      |      |    |   |    |   |
CALL SGEMX( 4 , 3 , 1.0 , A(1,0) , 10 , X , 1  , Y , 2  )
```

```
    ⎡              ⎤
    ⎢ 1.0  2.0  3.0 ⎥
    ⎢ 2.0  2.0  4.0 ⎥
    ⎢ 3.0  2.0  2.0 ⎥
    ⎢ 4.0  2.0  1.0 ⎥
A = ⎢  .    .    .  ⎥
    ⎢  .    .    .  ⎥
    ⎢  .    .    .  ⎥
    ⎢  .    .    .  ⎥
    ⎢  .    .    .  ⎥
    ⎢  .    .    .  ⎥
    ⎣              ⎦
```

```
X        =  (3.0, 2.0, 1.0)
Y        =  (4.0, . , 5.0, . , 2.0, . , 3.0)
```

Output

```
Y        =  (14.0, . , 19.0, . , 17.0, . , 20.0)
```

Example 7: This example shows a matrix, ***A***, contained in a larger array, A. The strides of vectors ***x*** and ***y*** are of opposite sign. For ***y***, which has negative stride, processing begins at element Y(7), which is 4.0. Array A must follow the same rules as given in Example 6. For this example, array A is declared as A(-1:8,1:3).

Call Statement and Input

```
           M   N  ALPHA     A     LDA   X  INCX  Y   INCY
           |   |    |       |      |    |   |    |    |
CALL SGEMX( 4 , 3 , 1.0 , A(-1,1) , 10 , X , 1  , Y , -2  )
```

```
A        =(same as input A in Example 6)
X        =  (3.0, 2.0, 1.0)
Y        =  (3.0, . , 2.0, . , 5.0, . , 4.0)
```

Output

```
Y        =  (20.0, . , 17.0, . , 19.0, . , 14.0)
```

Example 8: This example shows a matrix, ***A***, contained in a larger array, A, and the first element of the matrix is not the first element of the array. Array A must follow the same rules as given in Example 6. For this example, array A is declared as A(1:10,1:3).

Call Statement and Input

```
           M   N  ALPHA    A     LDA   X  INCX  Y  INCY
           |   |    |      |      |    |   |    |   |
CALL SGEMX( 4 , 3 , 1.0 , A(5,1) , 10 , X , 1  , Y , 1  )
```

```
    ⌈               ⌉
    | .    .    .   |
    | .    .    .   |
    | .    .    .   |
    | .    .    .   |
A = | 1.0  2.0  3.0 |
    | 2.0  2.0  4.0 |
    | 3.0  2.0  2.0 |
    | 4.0  2.0  1.0 |
    |  .    .    .  |
    |  .    .    .  |
    ⌊               ⌋
```

```
X       =  (3.0, 2.0, 1.0)
Y       =  (4.0, 5.0, 2.0, 3.0)
```

Output

```
Y       =  (14.0, 19.0, 17.0, 20.0)
```

Example 9: This example shows a matrix, ***A***, and an array, A, having the same number of rows. For this case, *m* and *lda* are equal. Because *lda* is 4 and *n* is 3, array A must be declared as A(E1:E2,F1:F2), where E2-E1+1=4 and F2-F1+1 ≥ 3. For this example, array A is declared as A(1:4,0:2).

Call Statement and Input

```
           M   N  ALPHA    A     LDA  X  INCX  Y  INCY
           |   |    |      |      |   |   |    |   |
CALL SGEMX( 4 , 3 , 1.0 , A(1,0) , 4 , X , 1  , Y , 1  )
```

```
    ⌈               ⌉
    | 1.0  2.0  3.0 |
A = | 2.0  2.0  4.0 |
    | 3.0  2.0  2.0 |
    | 4.0  2.0  1.0 |
    ⌊               ⌋
```

```
X       =  (3.0, 2.0, 1.0)
Y       =  (4.0, 5.0, 2.0, 3.0)
```

Output

```
Y       =  (14.0, 19.0, 17.0, 20.0)
```

Example 10: This example shows a matrix, ***A***, and an array, A, having the same number of rows. For this case, *m* and *lda* are equal. Because *lda* is 4 and *n* is 3, array A must be declared as A(E1:E2,F1:F2), where E2-E1+1=4 and F2-F1+1 ≥ 3. For this example, array A is declared as A(1:4,0:2).

Call Statement and Input

```
            M   N  ALPHA    A     LDA  X  INCX  Y  INCY
            |   |    |      |      |   |   |    |   |
CALL SGEMTX( 4 , 3 , 1.0 , A(1,0) , 4 , X , 1  , Y , 1  )
```

```
    ⎡               ⎤
    ⎢ 1.0  2.0  3.0 ⎥
A = ⎢ 2.0  2.0  4.0 ⎥
    ⎢ 3.0  2.0  2.0 ⎥
    ⎢ 4.0  2.0  1.0 ⎥
    ⎣               ⎦
```

```
X       =  (3.0, 2.0, 1.0, 4.0)
Y       =  (1.0, 2.0, 3.0)
```

Output

```
Y       =  (27.0, 22.0, 26.0)
```

Example 11: This example shows a computation in which *alpha* is greater than 1. Array A must follow the same rules as given in Example 10. For this example, array A is declared as A(-1:2,1:3).

Call Statement and Input

```
            M   N  ALPHA     A     LDA  X  INCX  Y  INCY
            |   |    |       |      |   |   |    |   |
CALL SGEMTX( 4 , 3 , 2.0 , A(-1,1) , 4 , X , 1  , Y , 1  )
```

```
A       =(same as input A in Example 10)
X       =  (3.0, 2.0, 1.0, 4.0)
Y       =  (1.0, 2.0, 3.0)
```

Output

```
Y       =  (53.0, 42.0, 49.0)
```

SGER, DGER, CGERU, ZGERU, CGERC, and ZGERC—Rank-One Update of a General Matrix

SGER, DGER, CGERU, and ZGERU compute the rank-one update of a general matrix, using the scalar α, matrix $\boldsymbol{A}$, vector $\boldsymbol{x}$, and the transpose of vector $\boldsymbol{y}$:

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

CGERC and ZGERC compute the rank-one update of a general matrix, using the scalar α, matrix $\boldsymbol{A}$, vector $\boldsymbol{x}$, and the conjugate transpose of vector $\boldsymbol{y}$:

$$\boldsymbol{A} \leftarrow \boldsymbol{A}+\alpha\boldsymbol{x}\boldsymbol{y}^{H}$$

Table 63. Data Types

α, $\boldsymbol{A}$, $\boldsymbol{x}$, $\boldsymbol{y}$	**Subprogram**
Short-precision real	SGER
Long-precision real	DGER
Short-precision complex	CGERU and CGERC
Long-precision complex	ZGERU and ZGERC

Note: For compatibility with earlier releases of ESSL, you can use the names SGER1 and DGER1 for SGER and DGER, respectively.

Syntax

Fortran	CALL SGER | DGER | CGERU | ZGERU | CGERC | ZGERC (*m*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *a*, *lda*)
C and C++	sger | dger | cgeru | zgeru | cgerc | zgerc (*m*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *a*, *lda*);
PL/I	CALL SGER | DGER | CGERU | ZGERU | CGERC | ZGERC (*m*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *a*, *lda*);

On Entry

m
is the number of rows in matrix $\boldsymbol{A}$ and the number of elements in vector $\boldsymbol{x}$. Specified as: a fullword integer; $0 \le m \le lda$.

n
is the number of columns in matrix $\boldsymbol{A}$ and the number of elements in vector $\boldsymbol{y}$. Specified as: a fullword integer; $n \ge 0$.

alpha
is the scaling constant α. Specified as: a number of the data type indicated in Table 63.

x
is the vector $\boldsymbol{x}$ of length *m*. Specified as: a one-dimensional array of (at least) length $1+(m-1)|incx|$, containing numbers of the data type indicated in Table 63.

incx
is the stride for vector $\boldsymbol{x}$. Specified as: a fullword integer. It can have any value.

y
is the vector $\boldsymbol{y}$ of length *n*, whose transpose or conjugate transpose is used in the computation.

Note: No data should be moved to form $\boldsymbol{y}^{T}$ or $\boldsymbol{y}^{H}$; that is, the vector ***y*** should always be stored in its untransposed form.

Specified as: a one-dimensional array of (at least) length $1+(n-1)|incy|$, containing numbers of the data type indicated in Table 63 on page 308.

incy
is the stride for vector ***y***. Specified as: a fullword integer. It can have any value.

a
is the *m* by *n* matrix ***A***. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 63 on page 308.

lda
is the size of the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and $lda \geq m$.

On Return

a
is the *m* by *n* matrix ***A***, containing the result of the computation.

Returned as: a two-dimensional array, containing numbers of the data type indicated in Table 63 on page 308.

Notes

1. In these subroutines, $incx = 0$ and $incy = 0$ are valid; however, the Level 2 BLAS standard considers $incx = 0$ and $incy = 0$ to be invalid. See references [34] and [35].
2. Matrix ***A*** can have no common elements with vectors ***x*** and ***y***; otherwise, results are unpredictable. See "Concepts" on page 55.

Function: SGER, DGER, CGERU, and ZGERU compute the rank-one update of a general matrix:

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

where:

A is an *m* by *n* matrix.
α is a scalar.
x is a vector of length *m*.
$\boldsymbol{y}^{T}$ is the transpose of vector ***y*** of length *n*.

It is expressed as follows:

$$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{m1} & \dots & a_{mn} \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} & \dots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{m1} & \dots & a_{mn} \end{bmatrix} + \alpha \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_m \end{bmatrix} \begin{bmatrix} y_1 \dots y_n \end{bmatrix}$$

It can also be expressed as:

$$\begin{bmatrix} a_{11} & \ldots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{m1} & \ldots & a_{mn} \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} + \alpha x_1 y_1 & \ldots & a_{1n} + \alpha x_1 y_n \\ . & & . \\ . & & . \\ . & & . \\ a_{m1} + \alpha x_m y_1 & \ldots & a_{mn} + \alpha x_m y_n \end{bmatrix}$$

CGERC and ZGERC compute a slightly different rank-one update of a general matrix:

$\boldsymbol{A} \leftarrow \boldsymbol{A} + \alpha \boldsymbol{x} \boldsymbol{y}^{H}$

where:

$\boldsymbol{A}$ is an m by n matrix.
α is a scalar.
$\boldsymbol{x}$ is a vector of length m.
$\boldsymbol{y}^{H}$ is the conjugate transpose of vector $\boldsymbol{y}$ of length n.

It is expressed as follows:

$$\begin{bmatrix} a_{11} & \ldots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{m1} & \ldots & a_{mn} \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} & \ldots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{m1} & \ldots & a_{mn} \end{bmatrix} + \alpha \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_m \end{bmatrix} \begin{bmatrix} \bar{y}_1 & \ldots & \bar{y}_n \end{bmatrix}$$

It can also be expressed as:

$$\begin{bmatrix} a_{11} & \ldots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{m1} & \ldots & a_{mn} \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} + \alpha x_1 \bar{y}_1 & \ldots & a_{1n} + \alpha x_1 \bar{y}_n \\ . & & . \\ . & & . \\ . & & . \\ a_{m1} + \alpha x_m \bar{y}_1 & \ldots & a_{mn} + \alpha x_m \bar{y}_n \end{bmatrix}$$

See references [34], [35], and [73]. No computation is performed if m, n, or α is zero. For CGERU and CGERC, intermediate results are accumulated in long precision. For SGER, intermediate results are accumulated in long precision on some platforms.

Error Conditions

Resource Errors: Unable to allocate internal work area.

Computational Errors: None

Input-Argument Errors

1. $m < 0$
2. $n < 0$
3. $lda \leq 0$
4. $m > lda$

Example 1: This example shows a matrix, ***A***, contained in a larger array, A. The strides of vectors ***x*** and ***y*** are positive. Because *lda* is 10 and *n* is 3, array A must be declared as A(E1:E2,F1:F2), where E2-E1+1=10 and F2-F1+1 ≥ 3. For this example, array A is declared as A(1:10,0:2).

Call Statement and Input

```
           M   N  ALPHA  X  INCX  Y  INCY    A     LDA
           |   |    |    |   |    |   |      |      |
CALL SGER( 4 , 3 , 1.0 , X , 1  , Y , 2  , A(1,0) , 10 )

X        =  (3.0, 2.0, 1.0, 4.0)
Y        =  (1.0, . , 2.0, . , 3.0)

    ⎡                ⎤
    ⎢ 1.0  2.0  3.0  ⎥
    ⎢ 2.0  2.0  4.0  ⎥
    ⎢ 3.0  2.0  2.0  ⎥
    ⎢ 4.0  2.0  1.0  ⎥
A = ⎢  .    .    .   ⎥
    ⎢  .    .    .   ⎥
    ⎢  .    .    .   ⎥
    ⎢  .    .    .   ⎥
    ⎢  .    .    .   ⎥
    ⎢  .    .    .   ⎥
    ⎣                ⎦
```

Output

```
    ⎡                 ⎤
    ⎢ 4.0   8.0  12.0 ⎥
    ⎢ 4.0   6.0  10.0 ⎥
    ⎢ 4.0   4.0   5.0 ⎥
    ⎢ 8.0  10.0  13.0 ⎥
A = ⎢  .     .     .  ⎥
    ⎢  .     .     .  ⎥
    ⎢  .     .     .  ⎥
    ⎢  .     .     .  ⎥
    ⎢  .     .     .  ⎥
    ⎢  .     .     .  ⎥
    ⎣                 ⎦
```

Example 2: This example shows a matrix, ***A***, contained in a larger array, A. The strides of vectors ***x*** and ***y*** are of opposite sign. For ***y***, which has negative stride, processing begins at element Y(5), which is 1.0. Array A must follow the same rules as given in Example 1. For this example, array A is declared as A(-1:8,1:3).

Call Statement and Input

```
           M   N  ALPHA  X  INCX  Y   INCY     A     LDA
           |   |    |    |   |    |    |       |      |
CALL SGER( 4 , 3 , 1.0 , X , 1  , Y , -2  , A(-1,1) , 10 )

X        =  (3.0, 2.0, 1.0, 4.0)
Y        =  (3.0, . , 2.0, . , 1.0)
A        =(same as input A in Example 1)
```

Output

```
A          =(same as input A in Example 1)
```

Example 3: This example shows a matrix, ***A***, contained in a larger array, `A`, and the first element of the matrix is not the first element of the array. Array `A` must follow the same rules as given in Example 1. For this example, array `A` is declared as `A(1:10,1:3)`.

Call Statement and Input

```
           M   N  ALPHA  X  INCX  Y  INCY    A     LDA
           |   |    |    |   |    |   |      |      |
CALL SGER( 4 , 3 , 1.0 , X , 3  , Y , 1  , A(4,1) , 10 )

X        =  (3.0, . , . , 2.0, . , . , 1.0, . , . , 4.0)
Y        =  (1.0, 2.0, 3.0)
```

```
    ⎡                ⎤
    ⎢ .    .    .    ⎥
    ⎢ .    .    .    ⎥
    ⎢ .    .    .    ⎥
    ⎢ 1.0  2.0  3.0  ⎥
A = ⎢ 2.0  2.0  4.0  ⎥
    ⎢ 3.0  2.0  2.0  ⎥
    ⎢ 4.0  2.0  1.0  ⎥
    ⎢  .    .    .   ⎥
    ⎢  .    .    .   ⎥
    ⎢  .    .    .   ⎥
    ⎣                ⎦
```

Output

```
    ⎡                  ⎤
    ⎢ .     .     .    ⎥
    ⎢ .     .     .    ⎥
    ⎢ .     .     .    ⎥
    ⎢ 4.0   8.0  12.0  ⎥
A = ⎢ 4.0   6.0  10.0  ⎥
    ⎢ 4.0   4.0   5.0  ⎥
    ⎢ 8.0  10.0  13.0  ⎥
    ⎢  .     .     .   ⎥
    ⎢  .     .     .   ⎥
    ⎢  .     .     .   ⎥
    ⎣                  ⎦
```

Example 4: This example shows a matrix, ***A***, and array, `A`, having the same number of rows. For this case, *m* and *lda* are equal. Because *lda* is 4 and *n* is 3, array `A` must be declared as `A(E1:E2,F1:F2)`, where `E2-E1+1=4` and `F2-F1+1 ≥ 3`. For this example, array `A` is declared as `A(1:4,0:2)`.

Call Statement and Input

```
           M   N  ALPHA  X  INCX  Y  INCY    A     LDA
           |   |    |    |   |    |   |      |      |
CALL SGER( 4 , 3 , 1.0 , X , 1  , Y , 1  , A(1,0) , 4 )

X        =  (3.0, 2.0, 1.0, 4.0)
Y        =  (1.0, 2.0, 3.0)

     ⎡               ⎤
     │ 1.0  2.0  3.0 │
A =  │ 2.0  2.0  4.0 │
     │ 3.0  2.0  2.0 │
     │ 4.0  2.0  1.0 │
     ⎣               ⎦
```

Output

```
     ⎡               ⎤
     │ 4.0   8.0  12.0 │
A =  │ 4.0   6.0  10.0 │
     │ 4.0   4.0   5.0 │
     │ 8.0  10.0  13.0 │
     ⎣               ⎦
```

Example 5: This example shows a computation in which scalar value for *alpha* is greater than 1. Array `A` must follow the same rules as given in Example 4. For this example, array `A` is declared as `A(-1:2,1:3)`.

Call Statement and Input

```
           M   N  ALPHA  X  INCX  Y  INCY     A     LDA
           |   |    |    |   |    |   |       |      |
CALL SGER( 4 , 3 , 2.0 , X , 1  , Y , 1  , A(-1,1) , 4 )

X        =  (3.0, 2.0, 1.0, 4.0)
Y        =  (1.0, 2.0, 3.0)
A        =(same as input A in Example 4)
```

Output

```
     ⎡               ⎤
     │  7.0  14.0  21.0 │
A =  │  6.0  10.0  16.0 │
     │  5.0   6.0   8.0 │
     │ 12.0  18.0  25.0 │
     ⎣               ⎦
```

Example 6: This example shows a rank-one update in which all data items contain complex numbers, and the transpose $\boldsymbol{y}^T$ is used in the computation. Matrix $\boldsymbol{A}$ is contained in a larger array, `A`. The strides of vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ are positive. The Fortran DIMENSION statement for array `A` must follow the same rules as given in Example 1. For this example, array `A` is declared as `A(1:10,0:2)`.

Call Statement and Input

```
            M   N   ALPHA   X  INCX  Y  INCY    A     LDA
            |   |     |     |   |    |   |      |      |
CALL CGERU( 5 , 3 , ALPHA , X , 1  , Y , 1  , A(1,0) , 10 )

ALPHA    =  (1.0, 0.0)
X        =  ((1.0, 2.0), (4.0, 0.0), (1.0, 1.0), (3.0, 4.0),
             (2.0, 0.0))
Y        =  ((1.0, 2.0), (4.0, 0.0), (1.0, -1.0))

    ⎡                                  ⎤
    ⎢ (1.0, 2.0)  (3.0, 5.0)  (2.0, 0.0) ⎥
    ⎢ (2.0, 3.0)  (7.0, 9.0)  (4.0, 8.0) ⎥
    ⎢ (7.0, 4.0)  (1.0, 4.0)  (6.0, 0.0) ⎥
    ⎢ (8.0, 2.0)  (2.0, 5.0)  (8.0, 0.0) ⎥
A = ⎢ (9.0, 1.0)  (3.0, 6.0)  (1.0, 0.0) ⎥
    ⎢     .           .           .      ⎥
    ⎢     .           .           .      ⎥
    ⎢     .           .           .      ⎥
    ⎢     .           .           .      ⎥
    ⎢     .           .           .      ⎥
    ⎣                                  ⎦
```

Output

```
    ⎡                                                ⎤
    ⎢ (-2.0,   6.0)   (7.0,  13.0)   (5.0,   1.0) ⎥
    ⎢  (6.0,  11.0)  (23.0,   9.0)   (8.0,   4.0) ⎥
    ⎢  (6.0,   7.0)   (5.0,   8.0)   (8.0,   0.0) ⎥
    ⎢  (3.0,  12.0)  (14.0,  21.0)  (15.0,   1.0) ⎥
A = ⎢ (11.0,   5.0)  (11.0,   6.0)   (3.0,  -2.0) ⎥
    ⎢      .              .              .        ⎥
    ⎢      .              .              .        ⎥
    ⎢      .              .              .        ⎥
    ⎢      .              .              .        ⎥
    ⎢      .              .              .        ⎥
    ⎣                                                ⎦
```

Example 7: This example shows a rank-one update in which all data items contain complex numbers, and the conjugate transpose $\boldsymbol{y}^H$ is used in the computation. Matrix $\boldsymbol{A}$ is contained in a larger array, `A`. The strides of vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ are positive. The Fortran DIMENSION statement for array `A` must follow the same rules as given in Example 1. For this example, array `A` is declared as `A(1:10,0:2)`.

Call Statement and Input

```
            M   N   ALPHA   X  INCX  Y  INCY    A     LDA
            |   |     |     |   |    |   |      |      |
CALL CGERC( 5 , 3 , ALPHA , X , 1  , Y , 1  , A(1,0) , 10 )

ALPHA    =  (1.0, 0.0)
X        =  ((1.0, 2.0), (4.0, 0.0), (1.0, 1.0), (3.0, 4.0),
             (2.0, 0.0))
Y        =  ((1.0, 2.0), (4.0, 0.0), (1.0, -1.0))
A        =(same as input A in Example 6 )
```

Output

```
    ⌈                                                ⌉
    |  (6.0,   2.0)   (7.0,  13.0)  (1.0,   3.0)  |
    |  (6.0,  -5.0)  (23.0,   9.0)  (8.0,  12.0)  |
    | (10.0,   3.0)   (5.0,   8.0)  (6.0,   2.0)  |
    | (19.0,   0.0)  (14.0,  21.0)  (7.0,   7.0)  |
A = | (11.0,  -3.0)  (11.0,   6.0)  (3.0,   2.0)  |
    |      .              .             .         |
    |      .              .             .         |
    |      .              .             .         |
    |      .              .             .         |
    |      .              .             .         |
    ⌊                                                ⌋
```

SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV, SSLMX, and DSLMX—Matrix-Vector Product for a Real Symmetric or Complex Hermitian Matrix

SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, and ZHEMV compute the matrix-vector product for either a real symmetric matrix or a complex Hermitian matrix, using the scalars α and β, matrix $\boldsymbol{A}$, and vectors $\boldsymbol{x}$ and $\boldsymbol{y}$:

$$\boldsymbol{y} \leftarrow \beta\boldsymbol{y}+\alpha\boldsymbol{A}\boldsymbol{x}$$

SSLMX and DSLMX compute the matrix-vector product for a real symmetric matrix, using the scalar α, matrix $\boldsymbol{A}$, and vectors $\boldsymbol{x}$ and $\boldsymbol{y}$:

$$\boldsymbol{y} \leftarrow \boldsymbol{y}+\alpha\boldsymbol{A}\boldsymbol{x}$$

The following storage modes are used:

- For SSPMV, DSPMV, CHPMV, and ZHPMV, matrix $\boldsymbol{A}$ is stored in upper- or lower-packed storage mode.
- For SSYMV, DSYMV, CHEMV, and ZHEMV, matrix $\boldsymbol{A}$ is stored in upper or lower storage mode.
- For SSLMX and DSLMX, matrix $\boldsymbol{A}$ is stored in lower-packed storage mode.

Table 64. Data Types

α, β, $\boldsymbol{A}$, $\boldsymbol{x}$, $\boldsymbol{y}$	**Subprogram**
Short-precision real	SSPMV, SSYMV, and SSLMX
Long-precision real	DSPMV, DSYMV, and DSLMX
Short-precision complex	CHPMV and CHEMV
Long-precision complex	ZHPMV and ZHEMV

Note: SSPMV and DSPMV are Level 2 BLAS subroutines. You should use these subroutines instead of SSLMX and DSLMX, which are provided only for compatibility with earlier releases of ESSL.

Syntax

Fortran	CALL SSPMV | DSPMV | CHPMV | ZHPMV (*uplo*, *n*, *alpha*, *ap*, *x*, *incx*, *beta*, *y*, *incy*) CALL SSYMV | DSYMV | CHEMV | ZHEMV (*uplo*, *n*, *alpha*, *a*, *lda*, *x*, *incx*, *beta*, *y*, *incy*) CALL SSLMX | DSLMX (*n*, *alpha*, *ap*, *x*, *incx*, *y*, *incy*)
C and C++	sspmv | dspmv | chpmv | zhpmv (*uplo*, *n*, *alpha*, *ap*, *x*, *incx*, *beta*, *y*, *incy*); ssymv | dsymv | chemv | zhemv (*uplo*, *n*, *alpha*, *a*, *lda*, *x*, *incx*, *beta*, *y*, *incy*); sslmx | dslmx (*n*, *alpha*, *ap*, *x*, *incx*, *y*, *incy*);
PL/I	CALL SSPMV | DSPMV | CHPMV | ZHPMV (*uplo*, *n*, *alpha*, *ap*, *x*, *incx*, *beta*, *y*, *incy*); CALL SSYMV | DSYMV | CHEMV | ZHEMV (*uplo*, *n*, *alpha*, *a*, *lda*, *x*, *incx*, *beta*, *y*, *incy*); CALL SSLMX | DSLMX (*n*, *alpha*, *ap*, *x*, *incx*, *y*, *incy*);

On Entry

uplo
indicates the storage mode used for matrix ***A***, where:

If *uplo* = 'U', ***A*** is stored in upper-packed or upper storage mode.

If *uplo* = 'L', ***A*** is stored in lower-packed or lower storage mode.

Specified as: a single character. It must be 'U' or 'L'.

n
is the number of elements in vectors ***x*** and ***y*** and the order of matrix ***A***. Specified as: a fullword integer; $n \geq 0$.

alpha
is the scaling constant α. Specified as: a number of the data type indicated in Table 64 on page 316.

ap
has the following meaning:

For SSPMV and DSPMV, *ap* is the real symmetric matrix ***A*** of order *n*, stored in upper- or lower-packed storage mode.

For CHPMV and ZHPMV, *ap* is the complex Hermitian matrix ***A*** of order *n*, stored in upper- or lower-packed storage mode.

For SSLMX and DSLMX, *ap* is the real symmetric matrix ***A*** of order *n*, stored in lower-packed storage mode.

Specified as: a one-dimensional array of (at least) length $n(n+1)/2$, containing numbers of the data type indicated in Table 64 on page 316.

a
has the following meaning:

For SSYMV and DSYMV, *a* is the real symmetric matrix ***A*** of order *n*, stored in upper or lower storage mode.

For CHEMV and ZHEMV, *a* is the complex Hermitian matrix ***A*** of order *n*, stored in upper or lower storage mode.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 64 on page 316.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and $lda \geq n$.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 64 on page 316.

incx
is the stride for vector ***x***. Specified as: a fullword integer, where:

For SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, and ZHEMV, $incx < 0$ or $incx > 0$.

For SSLMX and DSLMX, *incx* can have any value.

beta
is the scaling constant β. Specified as: a number of the data type indicated in Table 64 on page 316.

y
is the vector ***y*** of length *n*. Specified as: a one-dimensional array of (at least) length 1+(*n*−1)|*incy*|, containing numbers of the data type indicated in Table 64 on page 316.

incy
is the stride for vector ***y***. Specified as: a fullword integer; *incy* > 0 or *incy* < 0.

On Return

y
is the vector ***y*** of length *n*, containing the result of the computation. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 64 on page 316.

Notes

1. All subroutines accept lowercase letters for the *uplo* argument.
2. The vector ***y*** must have no common elements with vector ***x*** or matrix ***A***; otherwise, results are unpredictable. See "Concepts" on page 55.
3. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values.
4. For a description of how symmetric matrices are stored in upper- or lower-packed storage mode and upper or lower storage mode, see "Symmetric Matrix" on page 65. For a description of how complex Hermitian matrices are stored in upper- or lower-packed storage mode and upper or lower storage mode, see "Complex Hermitian Matrix" on page 70.

Function: These subroutines perform the computations described in the two sections below. See references [34], [35], and [73]. For SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, and ZHEMV, if *n* is zero or if α is zero and β is one, no computation is performed. For SSLMX and DSLMX, if *n* or α is zero, no computation is performed.

For SSLMX, SSPMV, SSYMV, CHPMV, and CHEMV, intermediate results are accumulated in long precision. However, several intermediate stores may occur for each element of the vector ***y***.

For SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, and ZHEMV: These subroutines compute the matrix-vector product for either a real symmetric matrix or a complex Hermitian matrix:

$$\boldsymbol{y} \leftarrow \beta\boldsymbol{y}+\alpha\boldsymbol{A}\boldsymbol{x}$$

where:

y is a vector of length *n*.
α is a scalar.
β is a scalar.
A is a real symmetric or complex Hermitian matrix of order *n*.
x is a vector of length *n*.

It is expressed as follows:

$$\begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix} \leftarrow \beta \begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix} + \alpha \begin{bmatrix} a_{11} & \ldots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{n1} & \ldots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix}$$

For SSLMX and DSLMX: These subroutines compute the matrix-vector product for a real symmetric matrix stored in lower-packed storage mode:

$\boldsymbol{y} \leftarrow \boldsymbol{y}+\alpha\boldsymbol{A}\boldsymbol{x}$

where:

$\boldsymbol{y}$ is a vector of length n.
α is a scalar.
$\boldsymbol{A}$ is a real symmetric matrix of order n.
$\boldsymbol{x}$ is a vector of length n.

It is expressed as follows:

$$\begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix} \leftarrow \begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix} + \alpha \begin{bmatrix} a_{11} & \ldots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{n1} & \ldots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix}$$

Error Conditions

Computational Errors: None

Input-Argument Errors

1. *uplo* ≠ 'L' or 'U'
2. $n < 0$
3. $lda < n$
4. $lda \leq 0$
5. $incx = 0$
6. $incy = 0$

Example 1: This example shows vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ with positive strides and a real symmetric matrix $\boldsymbol{A}$ of order 3, stored in lower-packed storage mode. Matrix $\boldsymbol{A}$ is:

```
| 8.0  4.0  2.0 |
| 4.0  6.0  7.0 |
| 2.0  7.0  3.0 |
```

Call Statement and Input

```
            UPLO  N  ALPHA  AP   X  INCX BETA  Y  INCY
             |    |    |     |   |   |    |    |   |
CALL SSPMV( 'L' , 3 , 1.0 , AP , X , 1 , 1.0 , Y , 2 )

AP       =  (8.0, 4.0, 2.0, 6.0, 7.0, 3.0)
X        =  (3.0, 2.0, 1.0)
Y        =  (5.0, . , 3.0, . , 2.0)
```

Output

```
Y        =  (39.0, . , 34.0, . , 25.0)
```

Example 2: This example shows vector ***x*** and ***y*** having strides of opposite signs. For ***x***, which has negative stride, processing begins at element `X(5)`, which is 1.0. The real symmetric matrix ***A*** of order 3 is stored in upper-packed storage mode. It uses the same input matrix ***A*** as in Example 1.

Call Statement and Input

```
            UPLO  N  ALPHA  AP   X  INCX  BETA  Y  INCY
             |    |    |     |   |   |    |    |   |
CALL SSPMV( 'U' , 3 , 1.0 , AP , X , -2 , 2.0 , Y , 1 )

AP       =  (8.0, 4.0, 6.0, 2.0, 7.0, 3.0)
X        =  (4.0, . , 2.0, . , 1.0)
Y        =  (6.0, 5.0, 4.0)
```

Output

```
Y        =  (36.0, 54.0, 36.0)
```

Example 3: This example shows vector ***x*** and ***y*** with positive stride and a complex Hermitian matrix ***A*** of order 3, stored in lower-packed storage mode. Matrix ***A*** is:

```
⌈                                  ⌉
|  (1.0, 0.0)  (3.0, 5.0)  (2.0, -3.0) |
| (3.0, -5.0)  (7.0, 0.0)  (4.0, -8.0) |
|  (2.0, 3.0)  (4.0, 8.0)   (6.0, 0.0) |
⌊                                  ⌋
```

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values.

Call Statement and Input

```
            UPLO  N   ALPHA   AP   X  INCX BETA   Y  INCY
             |    |     |      |   |   |    |     |   |
CALL CHPMV( 'L' , 3 , ALPHA , AP , X , 1 , BETA , Y , 2 )

ALPHA    =  (1.0, 0.0)
AP       =  ((1.0, . ), (3.0, -5.0), (2.0, 3.0), (7.0, . ),
             (4.0, 8.0), (6.0, . ))
X        =  ((1.0, 2.0), (4.0, 0.0), (3.0, 4.0))
BETA     =  (1.0, 0.0)
Y        =  ((1.0, 0.0), . , (2.0, -1.0), . , (2.0, 1.0))
```

Output

```
Y       =  ((32.0, 21.0), . , (87.0, -8.0), . , (32.0, 64.0))
```

Example 4: This example shows vector ***x*** and ***y*** having strides of opposite signs. For ***x***, which has negative stride, processing begins at element `X(5)`, which is (1.0, 2.0). The complex Hermitian matrix ***A*** of order 3 is stored in upper-packed storage mode. It uses the same input matrix ***A*** as in Example 3.

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values.

Call Statement and Input

```
           UPLO  N   ALPHA   AP   X  INCX  BETA   Y INCY
            |    |     |     |    |    |    |     |   |
CALL CHPMV( 'U' , 3 , ALPHA , AP , X , -2 , BETA , Y , 2 )

ALPHA    =  (1.0, 0.0)
AP       =  ((1.0, . ), (3.0, 5.0), (7.0, . ), (2.0, -3.0),
             (4.0, -8.0), (6.0, . ))
X        =  ((3.0, 4.0), . , (4.0, 0.0), . , (1.0, 2.0))
BETA     =  (0.0, 0.0)
Y        =(not relevant)
```

Output

```
Y       =  ((31.0, 21.0), . , (85.0, -7.0), . , (30.0, 63.0))
```

Example 5: This example shows vectors ***x*** and ***y*** with positive strides and a real symmetric matrix ***A*** of order 3, stored in lower storage mode. It uses the same input matrix ***A*** as in Example 1.

Call Statement and Input

```
           UPLO  N  ALPHA  A  LDA  X  INCX BETA  Y INCY
            |    |    |    |   |   |   |    |    |   |
CALL SSYMV( 'L' , 3 , 1.0 , A , 3 , X , 1 , 1.0 , Y , 2 )

    ⎡              ⎤
    ⎢ 8.0  .    .  ⎥
A = ⎢ 4.0  6.0  .  ⎥
    ⎢ 2.0  7.0  3.0⎥
    ⎣              ⎦

X        =  (3.0, 2.0, 1.0)
Y        =  (5.0, . , 3.0, . , 2.0)
```

Output

```
Y       =  (39.0, . , 34.0, . , 25.0)
```

Example 6: This example shows vector ***x*** and ***y*** having strides of opposite signs. For ***x***, which has negative stride, processing begins at element X(5), which is 1.0. The real symmetric matrix ***A*** of order 3 is stored in upper storage mode. It uses the same input matrix ***A*** as in Example 1.

Call Statement and Input

```
           UPLO  N  ALPHA  A  LDA  X  INCX  BETA  Y  INCY
            |    |    |    |   |   |    |    |    |   |
CALL SSYMV( 'U' , 3 , 1.0 , A , 4 , X , -2 , 2.0 , Y , 1 )

    ┌                ┐
    | 8.0  4.0  2.0  |
A = |  .   6.0  7.0  |
    |  .    .   3.0  |
    |  .    .    .   |
    └                ┘

X       =  (4.0, . , 2.0, . , 1.0)
Y       =  (6.0, 5.0, 4.0)
```

Output

```
A       =  (36.0, 54.0, 36.0)
```

Example 7: This example shows vector ***x*** and ***y*** with positive stride and a complex Hermitian matrix ***A*** of order 3, stored in lower storage mode. It uses the same input matrix ***A*** as in Example 3.

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values.

Call Statement and Input

```
           UPLO  N   ALPHA   A  LDA  X  INCX BETA   Y INCY
            |    |     |     |   |   |   |    |     |   |
CALL CHEMV( 'L' , 3 , ALPHA , A , 3 , X , 1 , BETA , Y , 2 )

ALPHA    =  (1.0, 0.0)

       ┌                                    ┐
       | (1.0,  . )       .          .      |
A    = | (3.0, -5.0)  (7.0, .  )     .      |
       | (2.0,  3.0)  (4.0, 8.0) (6.0,  . ) |
       └                                    ┘

X       =  ((1.0, 2.0), (4.0, 0.0), (3.0, 4.0))
BETA    =  (1.0, 0.0)
Y       =  ((1.0, 0.0), . , (2.0, -1.0), . , (2.0, 1.0))
```

Output

```
Y       =  ((32.0, 21.0), . , (87.0, -8.0), . , (32.0, 64.0))
```

Example 8: This example shows vector ***x*** and ***y*** having strides of opposite signs. For ***x***, which has negative stride, processing begins at element `X(5)`, which is (1.0, 2.0). The complex Hermitian matrix ***A*** of order 3 is stored in upper storage mode. It uses the same input matrix ***A*** as in Example 3.

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values.

Call Statement and Input

```
            UPLO  N   ALPHA   A  LDA  X  INCX  BETA   Y INCY
             |    |     |     |   |   |    |    |     |   |
CALL CHEMV( 'U' , 3 , ALPHA , A , 3 , X , -2 , BETA , Y , 2 )

ALPHA    =  (1.0, 0.0)

        ⌈                                    ⌉
        | (1.0, . )  (3.0, 5.0)  (2.0, -3.0) |
A    =  |     .      (7.0,  . )  (4.0, -8.0) |
        |     .          .       (6.0,   . ) |
        ⌊                                    ⌋

X        =  ((3.0, 4.0), . , (4.0, 0.0), . , (1.0, 2.0))
BETA     =  (0.0, 0.0)
Y        =(not relevant)
```

Output

```
Y        =  ((31.0, 21.0), . , (85.0, -7.0), . , (30.0, 63.0))
```

Example 9: This example shows vectors ***x*** and ***y*** with positive strides and a real symmetric matrix ***A*** of order 3. Matrix ***A*** is:

```
                    ⌈              ⌉
                    | 8.0  4.0  2.0 |
                    | 4.0  6.0  7.0 |
                    | 2.0  7.0  3.0 |
                    ⌊              ⌋
```

Call Statement and Input

```
            N  ALPHA  AP   X  INCX  Y  INCY
            |    |    |    |   |    |   |
CALL SSLMX( 3 , 1.0 , AP , X , 1  , Y , 2  )

AP       =  (8.0, 4.0, 2.0, 6.0, 7.0, 3.0)
X        =  (3.0, 2.0, 1.0)
Y        =  (5.0, . , 3.0, . , 2.0)
```

Output

```
Y        =  (39.0, . , 34.0, . , 25.0)
```

SSPR, DSPR, CHPR, ZHPR, SSYR, DSYR, CHER, ZHER, SSLR1, and DSLR1 —Rank-One Update of a Real Symmetric or Complex Hermitian Matrix

SSPR, DSPR, SSYR, DSYR, SSLR1, and DSLR1 compute the rank-one update of a real symmetric matrix, using the scalar α, matrix $\boldsymbol{A}$, vector $\boldsymbol{x}$, and its transpose $\boldsymbol{x}^T$:

$$\boldsymbol{A} \leftarrow \boldsymbol{A}+\alpha\boldsymbol{x}\boldsymbol{x}^T$$

CHPR, ZHPR, CHER, and ZHER compute the rank-one update of a complex Hermitian matrix, using the scalar α, matrix $\boldsymbol{A}$, vector $\boldsymbol{x}$, and its conjugate transpose $\boldsymbol{x}^H$:

$$\boldsymbol{A} \leftarrow \boldsymbol{A}+\alpha\boldsymbol{x}\boldsymbol{x}^H$$

The following storage modes are used:

- For SSPR, DSPR, CHPR, and ZHPR, matrix $\boldsymbol{A}$ is stored in upper- or lower-packed storage mode.
- For SSYR, DSYR, CHER, and ZHER, matrix $\boldsymbol{A}$ is stored in upper or lower storage mode.
- For SSLR1 and DSLR1, matrix $\boldsymbol{A}$ is stored in lower-packed storage mode.

Table 65. Data Types

$\boldsymbol{A}$, $\boldsymbol{x}$	α	Subprogram
Short-precision real	Short-precision real	SSPR, SSYR, and SSLR1
Long-precision real	Long-precision real	DSPR, DSYR, and DSLR1
Short-precision complex	Short-precision real	CHPR and CHER
Long-precision complex	Long-precision real	ZHPR and ZHER

Note: SSPR and DSPR are Level 2 BLAS subroutines. You should use these subroutines instead of SSLR1 and DSLR1, which are only provided for compatibility with earlier releases of ESSL.

Syntax

Fortran	CALL SSPR | DSPR | CHPR | ZHPR (*uplo*, *n*, *alpha*, *x*, *incx*, *ap*) CALL SSYR | DSYR | CHER | ZHER (*uplo*, *n*, *alpha*, *x*, *incx*, *a*, *lda*) CALL SSLR1 | DSLR1 (*n*, *alpha*, *x*, *incx*, *ap*)
C and C++	sspr | dspr | chpr | zhpr (*uplo*, *n*, *alpha*, *x*, *incx*, *ap*); ssyr | dsyr | cher | zher (*uplo*, *n*, *alpha*, *x*, *incx*, *a*, *lda*); sslr1 | dslr1 (*n*, *alpha*, *x*, *incx*, *ap*);
PL/I	CALL SSPR | DSPR | CHPR | ZHPR (*uplo*, *n*, *alpha*, *x*, *incx*, *ap*); CALL SSYR | DSYR | CHER | ZHER (*uplo*, *n*, *alpha*, *x*, *incx*, *a*, *lda*); CALL SSLR1 | DSLR1 (*n*, *alpha*, *x*, *incx*, *ap*);

On Entry

uplo
indicates the storage mode used for matrix ***A***, where:

If *uplo* = 'U', ***A*** is stored in upper-packed or upper storage mode.

If *uplo* = 'L', ***A*** is stored in lower-packed or lower storage mode.

Specified as: a single character. It must be 'U' or 'L'.

n
is the number of elements in vector ***x*** and the order of matrix ***A***. Specified as: a fullword integer; $n \geq 0$.

alpha
is the scaling constant α. Specified as: a number of the data type indicated in Table 65 on page 324.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 65 on page 324.

incx
is the stride for vector ***x***. Specified as: a fullword integer, where:

For SSPR, DSPR, CHPR, ZHPR, SSYR, DSYR, CHER, and ZHER, $incx < 0$ or $incx > 0$.

For SSLR1 and DSLR1, *incx* can have any value.

ap
has the following meaning:

For SSPR and DSPR, *ap* is the real symmetric matrix ***A*** of order *n*, stored in upper- or lower-packed storage mode.

For CHPR and ZHPR, *ap* is the complex Hermitian matrix ***A*** of order *n*, stored in upper- or lower-packed storage mode.

For SSLR1 and DSLR1, *ap* is the real symmetric matrix ***A*** of order *n*, stored in lower-packed storage mode.

Specified as: a one-dimensional array of (at least) length $n(n+1)/2$, containing numbers of the data type indicated in Table 65 on page 324.

a
has the following meaning:

For SSYR and DSYR, *a* is the real symmetric matrix ***A*** of order *n*, stored in upper or lower storage mode.

For CHER and ZHER, *a* is the complex Hermitian matrix ***A*** of order *n*, stored in upper or lower storage mode.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 65 on page 324.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and $lda \geq n$.

On Return

ap
is the matrix ***A*** of order *n*, containing the results of the computation. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 65 on page 324.

a
is the matrix ***A*** of order *n*, containing the results of the computation. Returned as: a two-dimensional array, containing numbers of the data type indicated in Table 65 on page 324.

Notes

1. All subroutines accept lowercase letters for the *uplo* argument.
2. The vector ***x*** must have no common elements with matrix ***A***; otherwise, results are unpredictable. See "Concepts" on page 55.
3. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, if $\alpha \neq 0.0$, they are set to zero.
4. For a description of how symmetric matrices are stored in upper- or lower-packed storage mode and upper or lower storage mode, see "Symmetric Matrix" on page 65. For a description of how complex Hermitian matrices are stored in upper- or lower-packed storage mode and upper or lower storage mode, see "Complex Hermitian Matrix" on page 70.

Function: These subroutines perform the computations described in the two sections below. See references [34], [35], and [73]. If *n* or α is 0, no computation is performed.

For CHPR and CHER, intermediate results are accumulated in long precision. For SSPR, SSYR, and SSLR1, intermediate results are accumulated in long precision on some platforms.

For SSPR, DSPR, SSYR, DSYR, SSLR1, and DSLR1: These subroutines compute the rank-one update of a real symmetric matrix:

$$\boldsymbol{A} \leftarrow \boldsymbol{A}+\alpha\boldsymbol{x}\boldsymbol{x}^{\mathrm{T}}$$

where:

A is a real symmetric matrix of order *n*.
α is a scalar.
x is a vector of length *n*.
$\boldsymbol{x}^{\mathrm{T}}$ is the transpose of vector ***x***.

It is expressed as follows:

$$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{n1} & \dots & a_{nn} \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} & \dots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{n1} & \dots & a_{nn} \end{bmatrix} + \alpha \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix} \begin{bmatrix} x_1 & \dots & x_n \end{bmatrix}$$

For CHPR, ZHPR, CHER, and ZHER: These subroutines compute the rank-one update of a complex Hermitian matrix:

$$\boldsymbol{A} \leftarrow \boldsymbol{A}+\alpha\boldsymbol{x}\boldsymbol{x}^{H}$$

where:

A is a complex Hermitian matrix of order *n*.
α is a scalar.
x is a vector of length *n*.
$\boldsymbol{x}^{H}$ is the conjugate transpose of vector ***x***.

It is expressed as follows:

$$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{n1} & \dots & a_{nn} \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} & \dots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{n1} & \dots & a_{nn} \end{bmatrix} + \alpha \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix} \begin{bmatrix} \bar{x}_1 & \dots & \bar{x}_n \end{bmatrix}$$

Error Condition

Computational Errors: None

Input-Argument Errors

1. *uplo* ≠ 'L' or 'U'
2. *n* < 0
3. *incx* = 0
4. *lda* ≤ 0
5. *lda* < *n*

Example 1: This example shows a vector ***x*** with a positive stride, and a real symmetric matrix ***A*** of order 3, stored in lower-packed storage mode. Matrix ***A*** is:

```
| 8.0  4.0  2.0 |
| 4.0  6.0  7.0 |
| 2.0  7.0  3.0 |
```

Call Statement and Input

```
          UPLO   N  ALPHA  X  INCX  AP
            |    |    |    |   |    |
CALL SSPR( 'L' , 3 , 1.0 , X , 1  , AP )

X        =  (3.0, 2.0, 1.0)
AP       =  (8.0, 4.0, 2.0, 6.0, 7.0, 3.0)
```

Output

```
AP       =  (17.0, 10.0, 5.0, 10.0, 9.0, 4.0)
```

Example 2: This example shows a vector ***x*** with a negative stride, and a real symmetric matrix ***A*** of order 3, stored in upper-packed storage mode. It uses the same input matrix ***A*** as in Example 1.

Call Statement and Input

```
          UPLO   N  ALPHA  X  INCX  AP
            |    |    |    |    |   |
CALL SSPR( 'U' , 3 , 1.0 , X , -2 , AP )

X        =  (1.0, . , 2.0, . , 3.0)
AP       =  (8.0, 4.0, 6.0, 2.0, 7.0, 3.0)
```

Output

```
AP       =  (17.0, 10.0, 10.0, 5.0, 9.0, 4.0)
```

Example 3: This example shows a vector ***x*** with a positive stride, and a complex Hermitian matrix ***A*** of order 3, stored in lower-packed storage mode. Matrix ***A*** is:

$$\begin{bmatrix} (1.0, 0.0) & (3.0, 5.0) & (2.0, -3.0) \\ (3.0, -5.0) & (7.0, 0.0) & (4.0, -8.0) \\ (2.0, 3.0) & (4.0, 8.0) & (6.0, 0.0) \end{bmatrix}$$

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, if $\alpha \neq 0.0$, they are set to zero.

Call Statement and Input

```
           UPLO  N  ALPHA  X  INCX AP
             |   |    |    |   |   |
CALL CHPR( 'L' , 3 , 1.0 , X , 1 , AP )

X        =  ((1.0, 2.0), (4.0, 0.0), (3.0, 4.0))
AP       =  ((1.0, . ), (3.0, -5.0), (2.0, 3.0), (7.0, . ),
             (4.0, 8.0), (6.0, . ))
```

Output

```
AP       =  ((6.0, 0.0), (7.0, -13.0), (13.0, 1.0), (23.0, 0.0),
             (16.0, 24.0), (31.0, 0.0))
```

Example 4: This example shows a vector ***x*** with a negative stride, and a complex Hermitian matrix ***A*** of order 3, stored in upper-packed storage mode. It uses the same input matrix ***A*** as in Example 3.

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, if $\alpha \neq 0.0$, they are set to zero.

Call Statement and Input

```
           UPLO  N  ALPHA  X  INCX  AP
            |    |    |    |    |   |
CALL CHPR( 'U' , 3 , 1.0 , X , -2 , AP )

X        =  ((3.0, 4.0), . , (4.0, 0.0), . , (1.0, 2.0))
AP       =  ((1.0, . ), (3.0, 5.0), (7.0, . ), (2.0, -3.0),
             (4.0, -8.0), (6.0, . ))
```

Output

```
AP       =  ((6.0, 0.0), (7.0, 13.0), (23.0, 0.0), (13.0, -1.0),
             (16.0, -24.0), (31.0, 0.0))
```

Example 5: This example shows a vector ***x*** with a positive stride, and a real symmetric matrix ***A*** of order 3, stored in lower storage mode. It uses the same input matrix ***A*** as in Example 1.

Call Statement and Input

```
           UPLO  N  ALPHA  X  INCX A  LDA
            |    |    |    |   |   |   |
CALL SSYR( 'L' , 3 , 1.0 , X , 1 , A , 3 )

X        =  (3.0, 2.0, 1.0)

         ⎡               ⎤
         | 8.0   .    .  |
A    =   | 4.0  6.0   .  |
         | 2.0  7.0  3.0 |
         ⎣               ⎦
```

Output

```
         ⎡                ⎤
         | 17.0    .    .  |
A    =   | 10.0  10.0   .  |
         |  5.0   9.0  4.0 |
         ⎣                ⎦
```

Example 6: This example shows a vector ***x*** with a negative stride, and a real symmetric matrix ***A*** of order 3, stored in upper storage mode. It uses the same input matrix ***A*** as in Example 1.

Call Statement and Input

```
           UPLO  N  ALPHA  X  INCX  A  LDA
            |    |    |    |    |   |   |
CALL SSYR( 'U' , 3 , 1.0 , X , -2 , A , 4 )

X        =  (1.0, . , 2.0, . , 3.0)

         ⎡               ⎤
         | 8.0  4.0  2.0 |
A    =   |  .   6.0  7.0 |
         |  .    .   3.0 |
         |  .    .    .  |
         ⎣               ⎦
```

Output

```
        ⎡                  ⎤
        ⎢ 17.0  10.0  5.0 ⎥
A    =  ⎢   .   10.0  9.0 ⎥
        ⎢   .     .   4.0 ⎥
        ⎣   .     .    .  ⎦
```

Example 7: This example shows a vector ***x*** with a positive stride, and a complex Hermitian matrix ***A*** of order 3, stored in lower storage mode. It uses the same input matrix ***A*** as in Example 3.

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, if $\alpha \neq 0.0$, they are set to zero.

Call Statement and Input

```
           UPLO  N  ALPHA  X  INCX A  LDA
            |    |    |    |   |   |   |
CALL CHER( 'L' , 3 , 1.0 , X , 1 , A , 3 )

X        =  ((1.0, 2.0), (4.0, 0.0), (3.0, 4.0))

        ⎡                                  ⎤
        ⎢   (1.0, . )       .           .  ⎥
A    =  ⎢ (3.0, -5.0)   (7.0, . )        .  ⎥
        ⎢  (2.0, 3.0)  (4.0, 8.0)  (6.0, . ) ⎥
        ⎣                                  ⎦
```

Output

```
        ⎡                                        ⎤
        ⎢   (6.0, 0.0)        .            .     ⎥
A    =  ⎢ (7.0, -13.0)   (23.0, 0.0)       .     ⎥
        ⎢  (13.0, 1.0)  (16.0, 24.0)  (31.0, 0.0) ⎥
        ⎣                                        ⎦
```

Example 8: This example shows a vector ***x*** with a negative stride, and a complex Hermitian matrix ***A*** of order 3, stored in upper storage mode. It uses the same input matrix ***A*** as in Example 3.

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, if $\alpha \neq 0.0$, they are set to zero.

Call Statement and Input

```
          UPLO  N  ALPHA  X  INCX  A  LDA
           |    |    |    |    |   |   |
CALL CHER( 'U' , 3 , 1.0 , X , -2 , A , 3 )

X       =  ((3.0, 4.0), . , (4.0, 0.0), . , (1.0, 2.0))

       ⎡                                    ⎤
       | (1.0, . )  (3.0, 5.0)  (2.0, -3.0) |
A   =  |     .      (7.0,  . )  (4.0, -8.0) |
       |     .          .         (6.0, . ) |
       ⎣                                    ⎦
```

Output

```
       ⎡                                     ⎤
       | (6.0, 0.0)  (7.0, 13.0)   (13.0, -1.0) |
A   =  |     .       (23.0, 0.0)  (16.0, -24.0) |
       |     .            .         (31.0, 0.0) |
       ⎣                                     ⎦
```

Example 9: This example shows a vector ***x*** with a positive stride, and a real symmetric matrix ***A*** of order 3, stored in lower-packed storage mode. It uses the same input matrix ***A*** as in Example 1.

Call Statement and Input

```
           N  ALPHA  X  INCX  AP
           |    |    |   |    |
CALL SSLR1( 3 , 1.0 , X , 1  , AP )

X       =  (3.0, 2.0, 1.0)
AP      =  (8.0, 4.0, 2.0, 6.0, 7.0, 3.0)
```

Output

```
AP      =  (17.0, 10.0, 5.0, 10.0, 9.0, 4.0)
```

SSPR2, DSPR2, CHPR2, ZHPR2, SSYR2, DSYR2, CHER2, ZHER2, SSLR2, and DSLR2—Rank-Two Update of a Real Symmetric or Complex Hermitian Matrix

SSPR2, DSPR2, SSYR2, DSYR2, SSLR2, and DSLR2 compute the rank-two update of a real symmetric matrix, using the scalar α, matrix $\boldsymbol{A}$, vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, and their transposes $\boldsymbol{x}^T$ and $\boldsymbol{y}^T$:

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

CHPR2, ZHPR2, CHER2, and ZHER2, compute the rank-two update of a complex Hermitian matrix, using the scalar α, matrix $\boldsymbol{A}$, vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, and their conjugate transposes $\boldsymbol{x}^H$ and $\boldsymbol{y}^H$:

$$\boldsymbol{A} \leftarrow \boldsymbol{A} + \alpha \boldsymbol{x}\boldsymbol{y}^H + \overline{\alpha} \boldsymbol{y}\boldsymbol{x}^H$$

The following storage modes are used:

- For SSPR2, DSPR2, CHPR2, and ZHPR2, matrix $\boldsymbol{A}$ is stored in upper- or lower-packed storage mode.
- For SSYR2, DSYR2, CHER2, and ZHER2, matrix $\boldsymbol{A}$ is stored in upper or lower storage mode.
- For SSLR2 and DSLR2, matrix $\boldsymbol{A}$ is stored in lower-packed storage mode.

Table 66. Data Types

α, $\boldsymbol{A}$, $\boldsymbol{x}$, $\boldsymbol{y}$	**Subprogram**
Short-precision real	SSPR2, SSYR2, and SSLR2
Long-precision real	DSPR2, DSYR2, and DSLR2
Short-precision complex	CHPR2 and CHER2
Long-precision complex	ZHPR2 and ZHER2

Note: SSPR2 and DSPR2 are Level 2 BLAS subroutines. You should use these subroutines instead of SSLR2 and DSLR2, which are only provided for compatibility with earlier releases of ESSL.

Syntax

Fortran	CALL SSPR2 | DSPR2 | CHPR2 | ZHPR2 (*uplo*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *ap*) CALL SSYR2 | DSYR2 | CHER2 | ZHER2 (*uplo*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *a*, *lda*) CALL SSLR2 | DSLR2 (*n*, *alpha*, *x*, *incx*, *y*, *incy*, *ap*)
C and C++	sspr2 | dspr2 | chpr2 | zhpr2 (*uplo*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *ap*); ssyr2 | dsyr2 | cher2 | zher2 (*uplo*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *a*, *lda*); sslr2 | dslr2 (*n*, *alpha*, *x*, *incx*, *y*, *incy*, *ap*);

PL/I	CALL SSPR2 | DSPR2 | CHPR2 | ZHPR2 (*uplo*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *ap*); CALL SSYR2 | DSYR2 | CHER2 | ZHER2 (*uplo*, *n*, *alpha*, *x*, *incx*, *y*, *incy*, *a lda*); CALL SSLR2 | DSLR2 (*n*, *alpha*, *x*, *incx*, *y*, *incy*, *ap*);

On Entry

uplo
indicates the storage mode used for matrix ***A***, where:

If *uplo* = 'U', ***A*** is stored in upper-packed or upper storage mode.

If *uplo* = 'L', ***A*** is stored in lower-packed or lower storage mode.

Specified as: a single character. It must be ‘U&csq or 'L'.

n
is the number of elements in vectors ***x*** and ***y*** and the order of matrix ***A***. Specified as: a fullword integer; $n \geq 0$.

alpha
is the scaling constant α. Specified as: a number of the data type indicated in Table 66 on page 332.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 66 on page 332.

incx
is the stride for vector ***x***.

Specified as: a fullword integer, where:

For SSPR2, DSPR2, CHPR2, ZHPR2, SSYR2, DSYR2, CHER2, and ZHER2, $incx < 0$ or $incx > 0$.

For SSLR2 and DSLR2, *incx* can have any value.

y
is the vector ***y*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incy|$, containing numbers of the data type indicated in Table 66 on page 332.

incy
is the stride for vector ***y***. Specified as: a fullword integer, where:

For SSPR2, DSPR2, CHPR2, ZHPR2, SSYR2, DSYR2, CHER2, and ZHER2, $incy < 0$ or $incy > 0$.

For SSLR2 and DSLR2, *incy* can have any value.

ap
has the following meaning:

For SSPR2 and DSPR2, *ap* is the real symmetric matrix ***A*** of order *n*, stored in upper- or lower-packed storage mode.

For CHPR2 and ZHPR2, *ap* is the complex Hermitian matrix ***A*** of order *n*, stored in upper- or lower-packed storage mode.

For SSLR2 and DSLR2, *ap* is the real symmetric matrix ***A*** of order *n*, stored in lower-packed storage mode.

Specified as: a one-dimensional array of (at least) length $n(n+1)/2$, containing numbers of the data type indicated in Table 66 on page 332.

a
has the following meaning:

For SSYR2 and DSYR2, *a* is the real symmetric matrix ***A*** of order *n*, stored in upper or lower storage mode.

For CHER2 and ZHER2, *a* is the complex Hermitian matrix ***A*** of order *n*, stored in upper or lower storage mode.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 66 on page 332.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and $lda \geq n$.

On Return

ap
is the matrix ***A*** of order *n*, containing the results of the computation. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 66 on page 332.

a
is the matrix ***A*** of order *n*, containing the results of the computation. Returned as: a two-dimensional array, containing numbers of the data type indicated in Table 66 on page 332.

Notes

1. All subroutines accept lowercase letters for the *uplo* argument.
2. The vectors ***x*** and ***y*** must have no common elements with matrix ***A***; otherwise, results are unpredictable. See "Concepts" on page 55.
3. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, if $\alpha \neq$ zero, the imaginary parts of the diagonal elements are set to zero.
4. For a description of how symmetric matrices are stored in upper- or lower-packed storage mode and upper or lower storage mode, see "Symmetric Matrix" on page 65. For a description of how complex Hermitian matrices are stored in upper- or lower-packed storage mode and upper or lower storage mode, see "Complex Hermitian Matrix" on page 70.

Function: These subroutines perform the computation described in the two sections below. See references [34], [35], and [73]. If *n* or α is zero, no computation is performed.

For SSPR2, SSYR2, SSLR2, CHPR2, and CHER2, intermediate results are accumulated in long precision.

SSPR2, DSPR2, SSYR2, DSYR2, SSLR2, and DSLR2: These subroutines compute the rank-two update of a real symmetric matrix:

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

where:

A is a real symmetric matrix of order *n*.
α is a scalar.

x is a vector of length *n*.
$\boldsymbol{x}^{\mathrm{T}}$ is the transpose of vector ***x***.
y is a vector of length *n*.
$\boldsymbol{y}^{\mathrm{T}}$ is the transpose of vector ***y***.

It is expressed as follows:

$$\begin{bmatrix} a_{11} & \ldots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{n1} & \ldots & a_{nn} \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} & \ldots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{n1} & \ldots & a_{nn} \end{bmatrix} + \alpha \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix} \begin{bmatrix} y_1 & \ldots & y_n \end{bmatrix}$$

$$+ \alpha \begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix} \begin{bmatrix} x_1 & \ldots & x_n \end{bmatrix}$$

CHPR2, ZHPR2, CHER2, and ZHER2: These subroutines compute the rank-two update of a complex Hermitian matrix:

$$\boldsymbol{A} \leftarrow \boldsymbol{A} + \alpha \boldsymbol{x}\boldsymbol{y}^{\mathrm{H}} + \overline{\alpha}\boldsymbol{y}\boldsymbol{x}^{\mathrm{H}}$$

where:

A is a complex Hermitian matrix of order *n*.
α is a scalar.
x is a vector of length *n*.
$\boldsymbol{x}^{\mathrm{H}}$ is the conjugate transpose of vector ***x***.
y is a vector of length *n*.
$\boldsymbol{y}^{\mathrm{H}}$ is the conjugate transpose of vector ***y***.

It is expressed as follows:

$$\begin{bmatrix} a_{11} & \ldots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{n1} & \ldots & a_{nn} \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} & \ldots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{n1} & \ldots & a_{nn} \end{bmatrix} + \alpha \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix} \begin{bmatrix} \overline{y}_1 & \ldots & \overline{y}_n \end{bmatrix}$$

$$+ \overline{\alpha} \begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix} \begin{bmatrix} \overline{x}_1 & \ldots & \overline{x}_n \end{bmatrix}$$

Error Condition

Computational Errors: None

Input-Argument Errors

1. *uplo* ≠ 'L' or 'U'
2. $n < 0$
3. $incx = 0$
4. $incy = 0$
5. $lda \leq 0$
6. $lda < n$

Example 1: This example shows vectors ***x*** and ***y*** with positive strides and a real symmetric matrix ***A*** of order 3, stored in lower-packed storage mode. Matrix ***A*** is:

```
⎡               ⎤
⎢ 8.0  4.0  2.0 ⎥
⎢ 4.0  6.0  7.0 ⎥
⎢ 2.0  7.0  3.0 ⎥
⎣               ⎦
```

Call Statement and Input

```
            UPLO  N  ALPHA  X  INCX  Y  INCY AP
             |    |    |    |   |    |   |   |
CALL SSPR2( 'L' , 3 , 1.0 , X , 1 ,  Y , 2 , AP )

X        =  (3.0, 2.0, 1.0)
Y        =  (5.0, . , 3.0, . , 2.0)
AP       =  (8.0, 4.0, 2.0, 6.0, 7.0, 3.0)
```

Output

```
AP       =  (38.0, 23.0, 13.0, 18.0, 14.0, 7.0)
```

Example 2: This example shows vector ***x*** and ***y*** having strides of opposite signs. For ***x***, which has negative stride, processing begins at element `X(5)`, which is 3.0. The real symmetric matrix ***A*** of order 3 is stored in upper-packed storage mode. It uses the same input matrix ***A*** as in Example 1.

Call Statement and Input

```
            UPLO  N  ALPHA  X   INCX Y  INCY AP
             |    |    |    |    |   |   |   |
CALL SSPR2( 'U' , 3 , 1.0 , X , -2 , Y , 2 , AP )

X        =  (1.0, . , 2.0, . , 3.0)
Y        =  (5.0, . , 3.0, . , 2.0)
AP       =  (8.0, 4.0, 6.0, 2.0, 7.0, 3.0)
```

Output

```
AP       =  (38.0, 23.0, 18.0, 13.0, 14.0, 7.0)
```

Example 3: This example shows vector ***x*** and ***y*** with positive stride and a complex Hermitian matrix ***A*** of order 3, stored in lower-packed storage mode. Matrix ***A*** is:

```
┌                                  ┐
│  (1.0, 0.0)  (3.0, 5.0)  (2.0, -3.0) │
│ (3.0, -5.0)  (7.0, 0.0)  (4.0, -8.0) │
│  (2.0, 3.0)  (4.0, 8.0)   (6.0, 0.0) │
└                                  ┘
```

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, if $\alpha \neq$ zero, the imaginary parts of the diagonal elements are set to zero.

Call Statement and Input

```
            UPLO  N   ALPHA   X  INCX Y  INCY AP
             |    |     |     |   |   |   |   |
CALL CHPR2( 'L' , 3 , ALPHA , X , 1 , Y , 2 , AP )

ALPHA    =  (1.0, 0.0)
X        =  ((1.0, 2.0), (4.0, 0.0), (3.0, 4.0))
Y        =  ((1.0, 0.0), . , (2.0, -1.0), . , (2.0, 1.0))
AP       =  ((1.0, . ), (3.0, -5.0), (2.0, 3.0), (7.0, . ),
             (4.0, 8.0), (6.0, . ))
```

Output

```
AP       =  ((3.0, 0.0), (7.0, -10.0), (9.0, 4.0), (23.0, 0.0),
             (14.0, 23.0), (26.0, 0.0))
```

Example 4: This example shows vector ***x*** and ***y*** having strides of opposite signs. For ***x***, which has negative stride, processing begins at element `X(5)`, which is (1.0,2.0). The complex Hermitian matrix ***A*** of order 3 is stored in upper-packed storage mode. It uses the same input matrix ***A*** as in Example 3.

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, if $\alpha \neq$ zero, the imaginary parts of the diagonal elements are set to zero.

Call Statement and Input

```
            UPLO  N   ALPHA   X  INCX  Y  INCY AP
             |    |     |     |   |   |   |   |
CALL CHPR2( 'U' , 3 , ALPHA , X , -2 , Y , 2 , AP )

ALPHA    =  (1.0, 0.0)
X        =  ((3.0, 4.0), . , (4.0, 0.0), . , (1.0, 2.0))
Y        =  ((1.0, 0.0), . , (2.0, -1.0), . , (2.0, 1.0))
AP       =  ((1.0, . ), (3.0, 5.0), (7.0, . ), (2.0, -3.0),
             (4.0, -8.0), (6.0, . ))
```

Output

```
AP       =  ((3.0, 0.0), (7.0, 10.0), (23.0, 0.0), (9.0, -4.0),
             (14.0, -23.0), (26.0, 0.0))
```

Example 5: This example shows vectors ***x*** and ***y*** with positive strides, and a real symmetric matrix ***A*** of order 3, stored in lower storage mode. It uses the same input matrix ***A*** as in Example 1.

Call Statement and Input

```
           UPLO  N  ALPHA  X  INCX Y  INCY A  LDA
            |    |    |    |   |   |   |   |   |
CALL SSYR2( 'L' , 3 , 1.0 , X , 1 , Y , 2 , A , 3 )

X        =  (3.0, 2.0, 1.0)
Y        =  (5.0, . , 3.0, . , 2.0)

         ⎡               ⎤
         | 8.0   .    .  |
A    =   | 4.0  6.0   .  |
         | 2.0  7.0  3.0 |
         ⎣               ⎦
```

Output

```
         ⎡                ⎤
         | 38.0    .    .  |
A    =   | 23.0  18.0   .  |
         | 13.0  14.0  7.0 |
         ⎣                ⎦
```

Example 6: This example shows vector ***x*** and ***y*** having strides of opposite signs. For ***x***, which has negative stride, processing begins at element `X(5)`, which is 3.0. The real symmetric matrix ***A*** of order 3 is stored in upper storage mode. It uses the same input matrix ***A*** as in Example 1.

Call Statement and Input

```
           UPLO  N  ALPHA  X  INCX  Y  INCY A  LDA
            |    |    |    |    |   |   |   |   |
CALL SSYR2( 'U' , 3 , 1.0 , X , -2 , Y , 2 , A , 4 )

X        =  (1.0, . , 2.0, . , 3.0)
Y        =  (5.0, . , 3.0, . , 2.0)

         ⎡               ⎤
         | 8.0  4.0  2.0 |
A    =   |  .   6.0  7.0 |
         |  .    .   3.0 |
         |  .    .    .  |
         ⎣               ⎦
```

Output

```
         ⎡                ⎤
         | 38.0  23.0  13.0 |
A    =   |   .   18.0  14.0 |
         |   .     .    7.0 |
         |   .     .     .  |
         ⎣                ⎦
```

Example 7: This example shows vector ***x*** and ***y*** with positive stride, and a complex Hermitian matrix ***A*** of order 3, stored in lower storage mode. It uses the same input matrix ***A*** as in Example 3.

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, if $\alpha \neq$ zero, the imaginary parts of the diagonal elements are set to zero.

Call Statement and Input

```
            UPLO  N   ALPHA   X  INCX Y  INCY A  LDA
             |    |     |     |   |   |   |   |   |
CALL CHER2( 'L' , 3 , ALPHA , X , 1 , Y , 2 , A , 3 )

ALPHA    =  (1.0, 0.0)
X        =  ((1.0, 2.0), (4.0, 0.0), (3.0, 4.0))
Y        =  ((1.0, 0.0), . , (2.0, -1.0), . , (2.0, 1.0))

        ⌈                                    ⌉
        |   (1.0, . )       .          .     |
A    =  | (3.0, -5.0)   (7.0, . )      .     |
        |  (2.0, 3.0)  (4.0, 8.0)  (6.0, . ) |
        ⌊                                    ⌋
```

Output

```
        ⌈                                        ⌉
        |   (3.0, 0.0)       .             .       |
A    =  | (7.0, -10.0)  (23.0, 0.0 )       .       |
        |   (9.0, 4.0)  (14.0, 23.0)  (26.0, 0.0 ) |
        ⌊                                        ⌋
```

Example 8: This example shows vector ***x*** and ***y*** having strides of opposite signs. For ***x***, which has negative stride, processing begins at element `X(5)`, which is (1.0, 2.0). The complex Hermitian matrix ***A*** of order 3 is stored in upper storage mode. It uses the same input matrix ***A*** as in Example 3.

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, if $\alpha \neq$ zero, the imaginary parts of the diagonal elements are set to zero.

Call Statement and Input

```
            UPLO  N   ALPHA   X  INCX  Y  INCY A  LDA
             |    |     |     |    |   |   |   |   |
CALL CHER2( 'U' , 3 , ALPHA , X , -2 , Y , 2 , A , 3 )

ALPHA    =  (1.0, 0.0)
X        =  ((3.0, 4.0), . , (4.0, 0.0), . , (1.0, 2.0))
Y        =  ((1.0, 0.0), . , (2.0, -1.0), . , (2.0, 1.0))

        ⌈                                    ⌉
        | (1.0, . ) (3.0, 5.0)   (2.0, -3.0) |
A    =  |     .      (7.0, . )   (4.0, -8.0) |
        |     .          .         (6.0, . ) |
        ⌊                                    ⌋
```

Output

```
        ⌈                                        ⌉
        | (3.0, 0.0)  (7.0, 10.0)    (9.0, -4.0) |
A    =  |     .       (23.0, 0.0)  (14.0, -23.0) |
        |     .            .         (26.0, 0.0) |
        ⌊                                        ⌋
```

Example 9: This example shows vectors ***x*** and ***y*** with positive strides and a real symmetric matrix ***A*** of order 3, stored in lower-packed storage mode. It uses the same input matrix ***A*** as in Example 1.

Call Statement and Input

```
            N  ALPHA  X  INCX  Y  INCY  AP
            |    |    |  |    |  |    |
CALL SSLR2( 3 , 1.0 , X , 1  , Y , 2  , AP )

X        =  (3.0, 2.0, 1.0)
Y        =  (5.0, . , 3.0, . , 2.0)
AP       =  (8.0, 4.0, 2.0, 6.0, 7.0, 3.0)
```

Output

```
AP       =  (38.0, 23.0, 13.0, 18.0, 14.0, 7.0)
```

SGBMV, DGBMV, CGBMV, and ZGBMV—Matrix-Vector Product for a General Band Matrix, Its Transpose, or Its Conjugate Transpose

SGBMV and DGBMV compute the matrix-vector product for either a real general band matrix or its transpose, where the general band matrix is stored in BLAS-general-band storage mode. It uses the scalars α and β, vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, and general band matrix $\boldsymbol{A}$ or its transpose:

$$\boldsymbol{y} \leftarrow \beta\boldsymbol{y}+\alpha\boldsymbol{A}\boldsymbol{x}$$

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

CGBMV and ZGBMV compute the matrix-vector product for either a complex general band matrix, its transpose, or its conjugate transpose, where the general band matrix is stored in BLAS-general-band storage mode. It uses the scalars α and β, vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, and general band matrix $\boldsymbol{A}$, its transpose, or its conjugate transpose:

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

Table 67. Data Types

α, β, $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{A}$	**Subprogram**
Short-precision real	SGBMV
Long-precision real	DGBMV
Short-precision complex	CGBMV
Long-precision complex	ZGBMV

Syntax

Fortran	CALL SGBMV | DGBMV | CGBMV | ZGBMV (*transa*, *m*, *n*, *ml*, *mu*, *alpha*, *a*, *lda*, *x*, *incx*, *beta*, *y*, *incy*)
C and C++	sgbmv | dgbmv | cgbmv | zgbmv (*transa*, *m*, *n*, *ml*, *mu*, *alpha*, *a*, *lda*, *x*, *incx*, *beta*, *y*, *incy*);
PL/I	CALL SGBMV | DGBMV | CGBMV | ZGBMV (*transa*, *m*, *n*, *ml*, *mu*, *alpha*, *a*, *lda*, *x*, *incx*, *beta*, *y*, *incy*);

On Entry

transa

indicates the form of matrix $\boldsymbol{A}$ to use in the computation, where:

If *transa* = 'N', $\boldsymbol{A}$ is used in the computation.

If *transa* = 'T', $\boldsymbol{A}^{T}$ is used in the computation.

If *transa* = 'C', $\boldsymbol{A}^{H}$ is used in the computation.

Specified as: a single character. It must be 'N', 'T', or 'C'.

m

is the number of rows in matrix $\boldsymbol{A}$, and:

If *transa* = 'N', it is the length of vector $\boldsymbol{y}$.

If *transa* = 'T' or 'C', it is the length of vector ***x***.

Specified as: a fullword integer; $m \geq 0$.

n

is the number of columns in matrix ***A***, and:

If *transa* = 'N', it is the length of vector ***x***.

If *transa* = 'T' or 'C', it is the length of vector ***y***.

Specified as: a fullword integer; $n \geq 0$.

ml

is the lower band width *ml* of the matrix ***A***. Specified as: a fullword integer; $ml \geq 0$.

mu

is the upper band width *mu* of the matrix ***A***. Specified as: a fullword integer; $mu \geq 0$.

alpha

is the scaling constant α. Specified as: a number of the data type indicated in Table 67 on page 341.

a

is the *m* by *n* general band matrix ***A***, stored in BLAS-general-band storage mode. It has an upper band width *mu* and a lower band width *ml*. Also:

If *transa* = 'N', ***A*** is used in the computation.

If *transa* = 'T', $\boldsymbol{A}^T$ is used in the computation.

If *transa* = 'C', $\boldsymbol{A}^H$ is used in the computation.

Note: No data should be moved to form $\boldsymbol{A}^T$ or $\boldsymbol{A}^H$; that is, the matrix ***A*** should always be stored in its untransposed form in BLAS-general-band storage mode.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 67 on page 341, where $lda \geq ml+mu+1$.

lda

is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and $lda \geq ml+mu+1$.

x

is the vector ***x***, where:

If *transa* = 'N', it has length *n*.

If *transa* = 'T' or 'C', it has length *m*.

Specified as: a one-dimensional array, containing numbers of the data type indicated in Table 67 on page 341, where:

If *transa* = 'N', it must have at least $1+(n-1)|incx|$ elements.

If *transa* = 'T' or 'C', it must have at least $1+(m-1)|incx|$ elements.

incx

is the stride for vector ***x***. Specified as: a fullword integer; $incx > 0$ or $incx < 0$.

beta

is the scaling constant β. Specified as: a number of the data type indicated in Table 67 on page 341.

y

is the vector ***y***, where:

If *transa* = 'N', it has length *m*.

If *transa* = 'T' or 'C', it has length *n*.

Specified as: a one-dimensional array, containing numbers of the data type indicated in Table 67 on page 341, where:

If *transa* = 'N', it must have at least $1+(m-1)|incy|$ elements.

If *transa* = 'T' or 'C', it must have at least $1+(n-1)|incy|$ elements.

incy
is the stride for vector ***y***. Specified as: a fullword integer; *incy* > 0 or *incy* < 0.

On Return

y
is the vector ***y***, containing the result of the computation, where:

If *transa* = 'N', it has length *m*.

If *transa* = 'T' or 'C', it has length *n*.

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 67 on page 341.

Notes

1. For SGBMV and DGBMV, if you specify 'C' for the *transa* argument, it is interpreted as though you specified 'T'.
2. All subroutines accept lowercase letters for the *transa* argument.
3. Vector ***y*** must have no common elements with matrix ***A*** or vector ***x***; otherwise, results are unpredictable. See "Concepts" on page 55.
4. To achieve optimal performance, use *lda* = *mu*+*ml*+1.
5. For general band matrices, if you specify *ml* ≥ *m* or *mu* ≥ *n*, ESSL assumes, **only for purposes of the computation**, that the lower band width is *m*−1 or the upper band width is *n*−1, respectively. However, ESSL uses the original values for *ml* and *mu* **for the purposes of finding the locations** of element a_{11} and all other elements in the array specified for ***A***, as described in "General Band Matrix" on page 77. For an illustration of this technique, see "Example 4" on page 346.
6. For a description of how a general band matrix is stored in BLAS-general-band storage mode in an array, see "General Band Matrix" on page 77.

Function: The possible computations that can be performed by these subroutines are described in the following sections. Varying implementation techniques are used for this computation to improve performance. As a result, accuracy of the computational result may vary for different computations.

In all the computations, general band matrix ***A*** is stored in its untransposed form in an array, using BLAS-general-band storage mode.

For SGBMV and CGBMV, intermediate results are accumulated in long precision. Occasionally, for performance reasons, these intermediate results are truncated to short precision and stored.

See references [34], [35], [38], [46], and [73]. No computation is performed if *m* or *n* is 0 or if α is zero and β is one.

General Band Matrix: For SGBMV, DGBMV, CGBMV, and ZGBMV, the matrix-vector product for a general band matrix is expressed as follows:

$$\boldsymbol{y} \leftarrow \beta\boldsymbol{y}+\alpha\boldsymbol{A}\boldsymbol{x}$$

where:

$\boldsymbol{x}$ is a vector of length n.
$\boldsymbol{y}$ is a vector of length m.
α is a scalar.
β is a scalar.
$\boldsymbol{A}$ is an m by n general band matrix, having a lower band width of *ml* and an upper band width of *mu*.

Transpose of a General Band Matrix: For SGBMV, DGBMV, CGBMV, and ZGBMV, the matrix-vector product for the transpose of a general band matrix is expressed as:

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

where:

$\boldsymbol{x}$ is a vector of length m.
$\boldsymbol{y}$ is a vector of length n.
α is a scalar.
β is a scalar.
$\boldsymbol{A}^{T}$ is the transpose of an m by n general band matrix $\boldsymbol{A}$, having a lower band width of *ml* and an upper band width of *mu*.

Conjugate Transpose of a General Band Matrix: For CGBMV and ZGBMV, the matrix-vector product for the conjugate transpose of a general band matrix is expressed as follows:

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

where:

$\boldsymbol{x}$ is a vector of length m.
$\boldsymbol{y}$ is a vector of length n.
α is a scalar.
β is a scalar.
$\boldsymbol{A}^{H}$ is the conjugate transpose of an m by n general band matrix $\boldsymbol{A}$ of order n, having a lower band width of *ml* and an upper band width of *mu*.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. *transa* ≠ 'N', 'T', or 'C'
2. $m < 0$
3. $n < 0$
4. $ml < 0$
5. $mu < 0$
6. $lda \leq 0$
7. $lda < ml+mu+1$

8. *incx* = 0
9. *incy* = 0

Example 1: This example shows how to use DGBMV to perform the computation $y \leftarrow \beta y+\alpha Ax$, where TRANSA is equal to 'N', and the following real general band matrix A is used in the computation. Matrix A is:

```
| 1.0  1.0  1.0  0.0 |
| 2.0  2.0  2.0  2.0 |
| 3.0  3.0  3.0  3.0 |
| 4.0  4.0  4.0  4.0 |
| 0.0  5.0  5.0  5.0 |
```

Call Statement and Input

```
           TRANSA M   N   ML  MU  ALPHA  A  LDA  X  INCX  BETA   Y  INCY
             |    |   |   |   |     |    |   |   |   |     |     |   |
CALL SGBMV( 'N' , 5 , 4 , 3 , 2 ,  2.0 , A , 8 , X , 1  , 10.0 , Y , 2  )
```

```
        | .    .   1.0  2.0 |
        | .   1.0  2.0  3.0 |
        | 1.0  2.0  3.0  4.0 |
A   =   | 2.0  3.0  4.0  5.0 |
        | 3.0  4.0  5.0   .  |
        | 4.0  5.0   .    .  |
        | .    .    .    .  |
        | .    .    .    .  |
```

```
X       =  (1.0, 2.0, 3.0, 4.0)
Y       =  (1.0, . , 2.0, . , 3.0, . , 4.0, . , 5.0, . )
```

Output

```
Y       =  (22.0, . , 60.0, . , 90.0, . , 120.0, . , 140.0, . )
```

Example 2: This example shows how to use SGBMV to perform the computation $y \leftarrow \beta y+\alpha A^{T}x$, where TRANSA is equal to 'T', and the transpose of a real general band matrix A is used in the computation. It uses the same input as Example 1.

Call Statement and Input

```
           TRANSA M   N   ML  MU  ALPHA  A  LDA  X  INCX  BETA   Y  INCY
             |    |   |   |   |     |    |   |   |   |     |     |   |
CALL SGBMV( 'T' , 5 , 4 , 3 , 2 ,  2.0 , A , 8 , X , 1  , 10.0 , Y , 2  )
```

Output

```
Y       =  (70.0, . , 130.0, . , 140.0, . , 148.0, . )
```

Example 3: This example shows how to use CGBMV to perform the computation $y \leftarrow \beta y+\alpha A^{H}x$, where TRANSA is equal to 'C', and the complex conjugate of the following general band matrix A is used in the computation. Matrix A is:

```
┌                                                    ┐
│ (1.0, 1.0)  (1.0, 1.0)  (1.0, 1.0)  (0.0, 0.0) │
│ (2.0, 2.0)  (2.0, 2.0)  (2.0, 2.0)  (2.0, 2.0) │
│ (3.0, 3.0)  (3.0, 3.0)  (3.0, 3.0)  (3.0, 3.0) │
│ (4.0, 4.0)  (4.0, 4.0)  (4.0, 4.0)  (4.0, 4.0) │
│ (0.0, 0.0)  (5.0, 5.0)  (5.0, 5.0)  (0.0, 0.0) │
└                                                    ┘
```

Call Statement and Input

```
           TRANSA M   N   ML  MU  ALPHA   A  LDA  X  INCX  BETA   Y  INCY
             |    |   |   |   |     |     |   |   |   |     |     |   |
CALL CGBMV( 'C' , 5 , 4 , 3 , 2 , ALPHA , A , 8 , X , 1  , BETA , Y , 2  )
```

```
        ┌                                          ┐
        │     .           .       (1.0, 1.0)  (2.0, 2.0) │
        │     .       (1.0, 1.0)  (2.0, 2.0)  (3.0, 3.0) │
        │ (1.0, 1.0)  (2.0, 2.0)  (3.0, 3.0)  (4.0, 4.0) │
A    =  │ (2.0, 2.0)  (3.0, 3.0)  (4.0, 4.0)  (5.0, 5.0) │
        │ (3.0, 3.0)  (4.0, 4.0)  (5.0, 5.0)      .      │
        │ (4.0, 4.0)  (5.0, 5.0)      .           .      │
        │     .           .           .           .      │
        │     .           .           .           .      │
        └                                          ┘

X       =  ((1.0, 2.0), (2.0, 3.0), (3.0, 4.0), (4.0, 5.0),
            (5.0, 6.0))
ALPHA   =  (1.0, 1.0)
BETA    =  (10.0, 0.0)
Y       =  ((1.0, 2.0), . , (2.0, 3.0), . , (3.0, 4.0), . ,
            (4.0, 5.0), . )
```

Output

```
Y       =  ((70.0, 100.0), . , (130.0, 170.0), . ,
            (140.0, 180.0), . , (148.0, 186.0), . )
```

Example 4: This example shows how to use SGBMV to perform the computation $y \leftarrow \beta y+\alpha Ax$, where $ml \geq m$ and $mu \geq n$, TRANSA is equal to 'N', and the following real general band matrix A is used in the computation. Matrix A is:

```
┌                    ┐
│ 1.0  1.0  1.0  1.0  1.0 │
│ 2.0  2.0  2.0  2.0  2.0 │
│ 3.0  3.0  3.0  3.0  3.0 │
│ 4.0  4.0  4.0  4.0  4.0 │
└                    ┘
```

Call Statement and Input

```
           TRANSA M   N   ML  MU  ALPHA  A  LDA   X  INCX  BETA   Y  INCY
             |    |   |   |   |     |    |   |    |   |     |     |   |
CALL SGBMV( 'N' , 4 , 5 , 6 , 5 ,  2.0 , A , 12 , X , 1  , 10.0 , Y , 2  )
```

```
        ⎡                                ⎤
        ⎢  .    .    .    .    .        ⎥
        ⎢  .    .    .    .   1.0       ⎥
        ⎢  .    .    .   1.0  2.0       ⎥
        ⎢  .    .   1.0  2.0  3.0       ⎥
        ⎢  .   1.0  2.0  3.0  4.0       ⎥
A    =  ⎢ 1.0  2.0  3.0  4.0   .        ⎥
        ⎢ 2.0  3.0  4.0   .    .        ⎥
        ⎢ 3.0  4.0   .    .    .        ⎥
        ⎢ 4.0   .    .    .    .        ⎥
        ⎢  .    .    .    .    .        ⎥
        ⎢  .    .    .    .    .        ⎥
        ⎢  .    .    .    .    .        ⎥
        ⎣                                ⎦

X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (1.0, . , 2.0, . , 3.0, . , 4.0, . )
```

Output

```
Y        =  (40.0, . , 80.0, . , 120.0, . , 160.0, . )
```

SSBMV, DSBMV, CHBMV, and ZHBMV—Matrix-Vector Product for a Real Symmetric or Complex Hermitian Band Matrix

SSBMV and DSBMV compute the matrix-vector product for a real symmetric band matrix. CHBMV and ZHBMV compute the matrix-vector product for a complex Hermitian band matrix. The band matrix ***A*** is stored in either upper- or lower-band-packed storage mode. It uses the scalars α and β, vectors ***x*** and ***y***, and band matrix ***A***:

$$\boldsymbol{y} \leftarrow \beta\boldsymbol{y}+\alpha\boldsymbol{A}\boldsymbol{x}$$
$$\boldsymbol{y} \leftarrow \beta\boldsymbol{y}+\alpha\boldsymbol{A}\boldsymbol{x}$$

Table 68. Data Types

α, β, ***x***, ***y***, ***A***	**Subprogram**
Short-precision real	SSBMV
Long-precision real	DSBMV
Short-precision complex	CHBMV
Long-precision complex	ZHBMV

Syntax

Fortran	CALL SSBMV | DSBMV | CHBMV | ZHBMV (*uplo*, *n*, *k*, *alpha*, *a*, *lda*, *x*, *incx*, *beta*, *y*, *incy*)
C and C++	ssbmv | dsbmv | chbmv | zhbmv (*uplo*, *n*, *k*, *alpha*, *a*, *lda*, *x*, *incx*, *beta*, *y*, *incy*);
PL/I	CALL SSBMV | DSBMV | CHBMV | ZHBMV (*uplo*, *n*, *k*, *alpha*, *a*, *lda*, *x*, *incx*, *beta*, *y*, *incy*);

On Entry

uplo
indicates the storage mode used for matrix ***A***, where either the upper or lower triangle can be stored:

If *uplo* = 'U', ***A*** is stored in upper-band-packed storage mode.

If *uplo* = 'L', ***A*** is stored in lower-band-packed storage mode.

Specified as: a single character. It must be 'U' or 'L'.

n
is the order of matrix ***A*** and the number of elements in vectors ***x*** and ***y***. Specified as: a fullword integer; $n \geq 0$.

k
is the half band width *k* of the matrix ***A***. Specified as: a fullword integer; $k \geq 0$.

alpha
is the scaling constant α. Specified as: a number of the data type indicated in Table 68.

a
is the real symmetric or complex Hermitian band matrix ***A*** of order *n*, having a half band width of *k*, where:

If *uplo* = 'U', ***A*** is stored in upper-band-packed storage mode.

If *uplo* = 'L', ***A*** is stored in lower-band-packed storage mode.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 68, where $lda \geq k+1$.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and $lda \geq k+1$.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 68 on page 348.

incx
is the stride for vector ***x***. Specified as: a fullword integer; $incx > 0$ or $incx < 0$.

beta
is the scaling constant β. Specified as: a number of the data type indicated in Table 68 on page 348.

y
is the vector ***y*** of length *n*. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 68 on page 348.

incy
is the stride for vector ***y***. Specified as: a fullword integer; $incy > 0$ or $incy < 0$.

On Return

y
is the vector ***y*** of length *n*, containing the result of the computation. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 68 on page 348.

Notes

1. All subroutines accept lowercase letters for the *uplo* argument.
2. Vector ***y*** must have no common elements with matrix ***A*** or vector ***x***; otherwise, results are unpredictable. See "Concepts" on page 55.
3. To achieve optimal performance in these subroutines, use $lda = k+1$.
4. The imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values.
5. For real symmetric and complex Hermitian band matrices, if you specify $k \geq n$, ESSL assumes, **only for purposes of the computation**, that the half band width of matrix ***A*** is $n-1$; that is, it processes matrix ***A***, of order *n*, as though it is a (nonbanded) real symmetric or complex Hermitian matrix. However, ESSL uses the original value for *k* **for the purposes of finding the locations** of element a_{11} and all other elements in the array specified for ***A***, as described in the storage modes referenced in the next note. For an illustration of this technique, see "Example 3" on page 352.
6. For a description of how a real symmetric band matrix is stored, see "Upper-Band-Packed Storage Mode" on page 83 or "Lower-Band-Packed Storage Mode" on page 84. For a description of how a complex Hermitian band matrix is stored, see "Complex Hermitian Matrix" on page 70.

Function: These subroutines perform the following matrix-vector product, using a real symmetric or complex Hermitian band matrix ***A***, stored in either upper- or lower-band-packed storage mode:

$$\boldsymbol{y} \leftarrow \beta\boldsymbol{y}+\alpha\boldsymbol{A}\boldsymbol{x}$$

where:

$\boldsymbol{x}$ and $\boldsymbol{y}$ are vectors of length n.
α and β are scalars.
$\boldsymbol{A}$ is an real symmetric or complex Hermitian band matrix of order n, having a half band width of k.

For SSBMV and CHBMV, intermediate results are accumulated in long precision. Occasionally, for performance reasons, these intermediate results are truncated to short precision and stored.

See references [34], [38], [46], and [73]. No computation is performed if n is 0 or if α is zero and β is one.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. *uplo* ≠ 'U' or 'L'
2. $n < 0$
3. $k < 0$
4. $lda \leq 0$
5. $lda < k+1$
6. $incx = 0$
7. $incy = 0$

Example 1: This example shows how to use SSBMV to perform the matrix-vector product, where the real symmetric band matrix $\boldsymbol{A}$ of order 7 and half band width of 3 is stored in upper-band-packed storage mode. Matrix $\boldsymbol{A}$ is:

```
[                                    ]
| 1.0  1.0  1.0  1.0  0.0  0.0  0.0 |
| 1.0  2.0  2.0  2.0  2.0  0.0  0.0 |
| 1.0  2.0  3.0  3.0  3.0  3.0  0.0 |
| 1.0  2.0  3.0  4.0  4.0  4.0  4.0 |
| 0.0  2.0  3.0  4.0  5.0  5.0  5.0 |
| 0.0  0.0  3.0  4.0  5.0  6.0  6.0 |
| 0.0  0.0  0.0  4.0  5.0  6.0  7.0 |
[                                    ]
```

Call Statement and Input

```
            UPLO  N   K  ALPHA  A  LDA  X  INCX  BETA   Y  INCY
             |    |   |    |    |   |   |   |     |     |   |
CALL SSBMV( 'U' , 7 , 3 , 2.0 , A , 5 , X , 1  , 10.0 , Y , 2  )
```

$$A = \begin{bmatrix} . & . & . & 1.0 & 2.0 & 3.0 & 4.0 \\ . & . & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 \\ . & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 \\ 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ . & . & . & . & . & . & . \end{bmatrix}$$

```
X        =  (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0)
Y        =  (1.0, . , 2.0, . , 3.0, . , 4.0, . , 5.0, . , 6.0, . , 7.0)
```

Output

```
Y        =  (30.0, . , 78.0, . , 148.0, . , 244.0, . , 288.0, . ,
             316.0, . , 322.0)
```

Example 2: This example shows how to use CHBMV to perform the matrix-vector product, where the complex Hermitian band matrix ***A*** of order 7 and half band width of 3 is stored in lower-band-packed storage mode. Matrix ***A*** is:

$$\begin{bmatrix} (1.0, 0.0) & (1.0, 1.0) & (1.0, 1.0) & (1.0, 1.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) \\ (1.0, -1.0) & (2.0, 0.0) & (2.0, 2.0) & (2.0, 2.0) & (2.0, 2.0) & (0.0, 0.0) & (0.0, 0.0) \\ (1.0, -1.0) & (2.0, -2.0) & (3.0, 0.0) & (3.0, 3.0) & (3.0, 3.0) & (3.0, 3.0) & (0.0, 0.0) \\ (1.0, -1.0) & (2.0, -2.0) & (3.0, -3.0) & (4.0, 0.0) & (4.0, 4.0) & (4.0, 4.0) & (4.0, 4.0) \\ (0.0, 0.0) & (2.0, -2.0) & (3.0, -3.0) & (4.0, -4.0) & (5.0, 0.0) & (5.0, 5.0) & (5.0, 5.0) \\ (0.0, 0.0) & (0.0, 0.0) & (3.0, -3.0) & (4.0, -4.0) & (5.0, -5.0) & (6.0, 0.0) & (6.0, 6.0) \\ (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (4.0, -4.0) & (5.0, -5.0) & (6.0, -6.0) & (7.0, 0.0) \end{bmatrix}$$

Note: The imaginary parts of the diagonal elements of a complex Hermitian matrix are assumed to be zero, so you do not need to set these values.

Call Statement and Input

```
            UPLO  N   K   ALPHA   A  LDA  X  INCX  BETA   Y  INCY
             |    |   |     |     |   |   |   |     |     |   |
CALL CHBMV( 'L' , 7 , 3 , ALPHA , A , 5 , X , 1  , BETA , Y , 2  )

ALPHA    =  (2.0, 0.0)
BETA     =  (10.0, 0.0)
```

$$A = \begin{bmatrix} (1.0, .) & (2.0, .) & (3.0, .) & (4.0, .) & (5.0, .) & (6.0, .) & (7.0, .) \\ (1.0, 1.0) & (2.0, 2.0) & (3.0, 3.0) & (4.0, 4.0) & (5.0, 5.0) & (6.0, 6.0) & . \\ (1.0, 1.0) & (2.0, 2.0) & (3.0, 3.0) & (4.0, 4.0) & (5.0, 5.0) & . & . \\ (1.0, 1.0) & (2.0, 2.0) & (3.0, 3.0) & (4.0, 4.0) & . & . & . \\ . & . & . & . & . & . & . \end{bmatrix}$$

```
X        =  ((1.0, 1.0), (2.0, 2.0), (3.0, 3.0), (4.0, 4.0),
             (5.0, 5.0), (6.0, 6.0), (7.0, 7.0))
Y        =  ((1.0, 1.0), . , (2.0, 2.0), . , (3.0, 3.0), . ,
             (4.0, 4.0), . , (5.0, 5.0), . , (6.0, 6.0), . ,
             (7.0, 7.0))
```

Output

```
Y        =  ((48.0, 12.0), . , (124.0, 32.0), . , (228.0, 68.0), . ,
             (360.0, 128.0), . , (360.0, 216.0), . ,
             (300.0, 332.0), . , (168.0, 476.0))
```

Example 3: This example shows how to use SSBMV to perform the matrix-vector product, where $n \geq k$. Matrix $\boldsymbol{A}$ is a real 5 by 5 symmetric band matrix with a half band width of 5, stored in upper-band-packed storage mode. Matrix $\boldsymbol{A}$ is:

$$\begin{bmatrix} 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\ 1.0 & 2.0 & 2.0 & 2.0 & 2.0 \\ 1.0 & 2.0 & 3.0 & 3.0 & 3.0 \\ 1.0 & 2.0 & 3.0 & 4.0 & 4.0 \\ 1.0 & 2.0 & 3.0 & 4.0 & 5.0 \end{bmatrix}$$

Call Statement and Input

```
            UPLO  N   K  ALPHA  A  LDA  X  INCX  BETA   Y  INCY
             |    |   |    |    |   |   |   |     |     |   |
CALL SSBMV( 'U' , 5 , 5 , 2.0 , A , 7 , X , 1  , 10.0 , Y , 2  )
```

$$A = \begin{bmatrix} . & . & . & . & . \\ . & . & . & . & 1.0 \\ . & . & . & 1.0 & 2.0 \\ . & . & 1.0 & 2.0 & 3.0 \\ . & 1.0 & 2.0 & 3.0 & 4.0 \\ 1.0 & 2.0 & 3.0 & 4.0 & 5.0 \\ . & . & . & . & . \end{bmatrix}$$

```
X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
Y        =  (1.0, . , 2.0, . , 3.0, . , 4.0, . , 5.0, . )
```

Output

```
Y        =  (40.0, . , 78.0, . , 112.0, . , 140.0, . , 160.0, . )
```

STRMV, DTRMV, CTRMV, ZTRMV, STPMV, DTPMV, CTPMV, and ZTPMV—Matrix-Vector Product for a Triangular Matrix, Its Transpose, or Its Conjugate Transpose

STRMV, DTRMV, STPMV, and DTPMV compute one of the following matrix-vector products, using the vector $\boldsymbol{x}$ and triangular matrix $\boldsymbol{A}$ or its transpose:

$\boldsymbol{x} \leftarrow \boldsymbol{A}\boldsymbol{x}$
$\boldsymbol{x} \leftarrow \boldsymbol{A}^{T}\boldsymbol{x}$

CTRMV, ZTRMV, CTPMV, and ZTPMV compute one of the following matrix-vector products, using the vector $\boldsymbol{x}$ and triangular matrix $\boldsymbol{A}$, its transpose, or its conjugate transpose:

$\boldsymbol{x} \leftarrow \boldsymbol{A}\boldsymbol{x}$
$\boldsymbol{x} \leftarrow \boldsymbol{A}^{T}\boldsymbol{x}$
$\boldsymbol{x} \leftarrow \boldsymbol{A}^{H}\boldsymbol{x}$

Matrix $\boldsymbol{A}$ can be either upper or lower triangular, where:

- For the _TRMV subroutines, it is stored in upper- or lower-triangular storage mode, respectively.
- For the _TPMV subroutines, it is stored in upper- or lower-triangular-packed storage mode, respectively.

Table 69. Data Types

$\boldsymbol{A}$, $\boldsymbol{x}$	**Subprogram**
Short-precision real	STRMV and STPMV
Long-precision real	DTRMV and DTPMV
Short-precision complex	CTRMV and CTPMV
Long-precision complex	ZTRMV and ZTPMV

Syntax

Fortran	CALL STRMV | DTRMV | CTRMV | ZTRMV (*uplo*, *transa*, *diag*, *n*, *a*, *lda*, *x*, *incx*) CALL STPMV | DTPMV | CTPMV | ZTPMV (*uplo*, *transa*, *diag*, *n*, *ap*, *x*, *incx*)
C and C++	strmv | dtrmv | ctrmv | ztrmv (*uplo*, *transa*, *diag*, *n*, *a*, *lda*, *x*, *incx*); stpmv | dtpmv | ctpmv | ztpmv (*uplo*, *transa*, *diag*, *n*, *ap*, *x*, *incx*);
PL/I	CALL STRMV | DTRMV | CTRMV | ZTRMV (*uplo*, *transa*, *diag*, *n*, *a*, *lda*, *x*, *incx*); CALL STPMV | DTPMV | CTPMV | ZTPMV (*uplo*, *transa*, *diag*, *n*, *ap*, *x*, *incx*);

On Entry

uplo
indicates whether matrix $\boldsymbol{A}$ is an upper or lower triangular matrix, where:

If *uplo* = 'U', $\boldsymbol{A}$ is an upper triangular matrix.

If *uplo* = 'L', $\boldsymbol{A}$ is a lower triangular matrix.

Specified as: a single character. It must be 'U' or 'L'.

transa
indicates the form of matrix ***A*** to use in the computation, where:

If *transa* = 'N', ***A*** is used in the computation.

If *transa* = 'T', $\boldsymbol{A}^{T}$ is used in the computation.

If *transa* = 'C', $\boldsymbol{A}^{H}$ is used in the computation.

Specified as: a single character. It must be 'N', 'T', or 'C'.

diag
indicates the characteristics of the diagonal of matrix ***A***, where:

If *diag* = 'U', ***A*** is a unit triangular matrix.

If *diag* = 'N', ***A*** is not a unit triangular matrix.

Specified as: a single character. It must be 'U' or 'N'.

n
is the order of triangular matrix ***A***. Specified as: a fullword integer; $0 \le n \le lda$.

a
is the upper or lower triangular matrix ***A*** of order *n*, stored in upper- or lower-triangular storage mode, respectively.

Note: No data should be moved to form $\boldsymbol{A}^{T}$ or $\boldsymbol{A}^{H}$; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 69 on page 353.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and $lda \ge n$.

ap
is the upper or lower triangular matrix ***A*** of order *n*, stored in upper- or lower-triangular-packed storage mode, respectively. Specified as: a one-dimensional array of (at least) length $n(n+1)/2$, containing numbers of the data type indicated in Table 69 on page 353.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 69 on page 353.

incx
is the stride for vector ***x***. Specified as: a fullword integer; $incx > 0$ or $incx < 0$.

On Return

x
is the vector ***x*** of length *n*, containing the results of the computation. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 69 on page 353.

Notes

1. These subroutines accept lowercase letters for the *uplo*, *transa*, and *diag* arguments.
2. For STRMV, DTRMV, STPMV, and DTPMV if you specify 'C' for the *transa* argument, it is interpreted as though you specified 'T'.

3. Matrix ***A*** and vector ***x*** must have no common elements; otherwise, results are unpredictable.
4. ESSL assumes certain values in your array for parts of a triangular matrix. As a result, you do not have to set these values. For unit triangular matrices, the elements of the diagonal are assumed to be 1.0 for real matrices and (1.0, 0.0) for complex matrices. When using upper- or lower-triangular storage, the unreferenced elements in the lower and upper triangular part, respectively, are assumed to be zero.
5. For a description of triangular matrices and how they are stored in upper- and lower-triangular storage mode and in upper- and lower-triangular-packed storage mode, see "Triangular Matrix" on page 74.

Function: These subroutines can perform the following matrix-vector product computations, using the triangular matrix ***A***, its transpose, or its conjugate transpose, where ***A*** can be either upper or lower triangular:

$\boldsymbol{x} \leftarrow \boldsymbol{A}\boldsymbol{x}$
$\boldsymbol{x} \leftarrow \boldsymbol{A}^{T}\boldsymbol{x}$
$\boldsymbol{x} \leftarrow \boldsymbol{A}^{H}\boldsymbol{x}$ (for CTRMV, ZTRMV, CTPMV, and ZTPMV only)

where:

x is a vector of length *n*.
A is an upper or lower triangular matrix of order *n*. For _TRMV, it is stored in upper- or lower-triangular storage mode, respectively. For _TPMV, it is stored in upper- or lower-triangular-packed storage mode, respectively.

See references [32] and [38]. If *n* is 0, no computation is performed.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. *uplo* ≠ 'L' or 'U'
2. *transa* ≠ 'T', 'N', or 'C'
3. *diag* ≠ 'N' or 'U'
4. $n < 0$
5. $lda \leq 0$
6. $lda < n$
7. $incx = 0$

Example 1: This example shows the computation $\boldsymbol{x} \leftarrow \boldsymbol{A}\boldsymbol{x}$. Matrix ***A*** is a real 4 by 4 lower triangular matrix that is unit triangular, stored in lower-triangular storage mode. Vector ***x*** is a vector of length 4. Matrix ***A*** is:

```
┌                    ┐
│ 1.0   .    .    .  │
│ 1.0  1.0   .    .  │
│ 2.0  3.0  1.0   .  │
│ 3.0  4.0  3.0  1.0 │
└                    ┘
```

Note: Because matrix ***A*** is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of 1.0 for the diagonal elements.

Call Statement and Input

```
            UPLO TRANSA DIAG  N   A  LDA  X  INCX
             |     |     |    |   |   |   |   |
CALL STRMV( 'L' , 'N' , 'U' , 4 , A , 4 , X , 1  )
```

```
        ⌈                    ⌉
        |  .    .    .    .  |
A    =  | 1.0   .    .    .  |
        | 2.0  3.0   .    .  |
        | 3.0  4.0  3.0   .  |
        ⌊                    ⌋

X        =  (1.0, 2.0, 3.0, 4.0)
```

Output

```
X        =  (1.0, 3.0, 11.0, 24.0)
```

Example 2: This example shows the computation $\boldsymbol{x} \leftarrow \boldsymbol{A}^{\mathrm{T}}\boldsymbol{x}$. Matrix $\boldsymbol{A}$ is a real 4 by 4 upper triangular matrix that is unit triangular, stored in upper-triangular storage mode. Vector $\boldsymbol{x}$ is a vector of length 4. Matrix $\boldsymbol{A}$ is:

```
                        ⌈                    ⌉
                        | 1.0  2.0  3.0  2.0 |
                        |  .   1.0  2.0  5.0 |
                        |  .    .   1.0  3.0 |
                        |  .    .    .   1.0 |
                        ⌊                    ⌋
```

Note: Because matrix $\boldsymbol{A}$ is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of 1.0 for the diagonal elements.

Call Statement and Input

```
            UPLO TRANSA DIAG  N   A  LDA  X  INCX
             |     |     |    |   |   |   |   |
CALL STRMV( 'U' , 'T' , 'U' , 4 , A , 4 , X , 1  )
```

```
        ⌈                    ⌉
        |  .   2.0  3.0  2.0 |
A    =  |  .    .   2.0  5.0 |
        |  .    .    .   3.0 |
        |  .    .    .    .  |
        ⌊                    ⌋

X        =  (5.0, 4.0, 3.0, 2.0)
```

Output

```
X        =  (5.0, 14.0, 26.0, 41.0)
```

Example 3: This example shows the computation $\boldsymbol{x} \leftarrow \boldsymbol{A}^{\mathrm{H}}\boldsymbol{x}$. Matrix $\boldsymbol{A}$ is a complex 4 by 4 upper triangular matrix that is unit triangular, stored in upper-triangular storage mode. Vector $\boldsymbol{x}$ is a vector of length 4. Matrix $\boldsymbol{A}$ is:

```
⎡                                                    ⎤
| (1.0, 0.0)  (2.0, 2.0)  (3.0, 3.0)  (2.0, 2.0) |
|     .       (1.0, 0.0)  (2.0, 2.0)  (5.0, 5.0) |
|     .           .       (1.0, 0.0)  (3.0, 3.0) |
|     .           .           .       (1.0, 0.0) |
⎣                                                    ⎦
```

Note: Because matrix ***A*** is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of (1.0, 0.0) for the diagonal elements.

Call Statement and Input

```
           UPLO TRANSA DIAG  N   A  LDA  X  INCX
            |     |     |    |   |   |   |   |
CALL CTRMV( 'U' , 'C' , 'U' , 4 , A , 4 , X , 1  )
```

```
       ⎡                                    ⎤
       | .  (2.0, 2.0)  (3.0, 3.0)  (2.0, 2.0) |
A   =  | .      .       (2.0, 2.0)  (5.0, 5.0) |
       | .      .           .       (3.0, 3.0) |
       | .      .           .           .      |
       ⎣                                    ⎦
```

```
X       =  ((5.0, 5.0), (4.0, 4.0), (3.0, 3.0), (2.0, 2.0))
```

Output

```
X       =  ((5.0, 5.0), (24.0, 4.0), (49.0, 3.0), (80.0, 2.0))
```

Example 4: This example shows the computation $\boldsymbol{x} \leftarrow \boldsymbol{Ax}$. Matrix ***A*** is a real 4 by 4 lower triangular matrix that is unit triangular, stored in lower-triangular-packed storage mode. Vector ***x*** is a vector of length 4. Matrix ***A*** is:

```
⎡                    ⎤
| 1.0   .    .    .  |
| 1.0  1.0   .    .  |
| 2.0  3.0  1.0   .  |
| 3.0  4.0  3.0  1.0 |
⎣                    ⎦
```

Note: Because matrix ***A*** is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of 1.0 for the diagonal elements.

Call Statement and Input

```
           UPLO TRANSA DIAG  N   AP   X  INCX
            |     |     |    |   |    |   |
CALL STPMV( 'L' , 'N' , 'U' , 4 , AP , X , 1  )
```

```
AP      =  ( . , 1.0, 2.0, 3.0, . , 3.0, 4.0, . , 3.0, . )
X       =  (1.0, 2.0, 3.0, 4.0)
```

Output

```
X       =  (1.0, 3.0, 11.0, 24.0)
```

Example 5: This example shows the computation $\boldsymbol{x} \leftarrow \boldsymbol{A}^{T}\boldsymbol{x}$. Matrix $\boldsymbol{A}$ is a real 4 by 4 upper triangular matrix that is not unit triangular, stored in upper-triangular-packed storage mode. Vector $\boldsymbol{x}$ is a vector of length 4. Matrix $\boldsymbol{A}$ is:

```
⎡                    ⎤
| 1.0  2.0  3.0  2.0 |
|  .   2.0  2.0  5.0 |
|  .    .   3.0  3.0 |
|  .    .    .   1.0 |
⎣                    ⎦
```

Call Statement and Input

```
            UPLO TRANSA DIAG  N   AP   X  INCX
             |     |     |    |   |    |   |
CALL STPMV( 'U' , 'T' , 'N' , 4 , AP , X , 1  )

AP       =  (1.0, 2.0, 2.0, 3.0, 2.0, 3.0, 2.0, 5.0, 3.0, 1.0)
X        =  (5.0, 4.0, 3.0, 2.0)
```

Output

```
X        =  (5.0, 18.0, 32.0, 41.0)
```

Example 6: This example shows the computation $\boldsymbol{x} \leftarrow \boldsymbol{A}^{H}\boldsymbol{x}$. Matrix $\boldsymbol{A}$ is a complex 4 by 4 upper triangular matrix that is unit triangular, stored in upper-triangular-packed storage mode. Vector $\boldsymbol{x}$ is a vector of length 4. Matrix $\boldsymbol{A}$ is:

```
⎡                                                    ⎤
| (1.0, 0.0) (2.0, 2.0) (3.0, 3.0) (2.0, 2.0) |
|     .      (1.0, 0.0) (2.0, 2.0) (5.0, 5.0) |
|     .          .      (1.0, 0.0) (3.0, 3.0) |
|     .          .          .      (1.0, 0.0) |
⎣                                                    ⎦
```

Note: Because matrix $\boldsymbol{A}$ is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of (1.0, 0.0) for the diagonal elements.

Call Statement and Input

```
            UPLO TRANSA DIAG  N   AP   X  INCX
             |     |     |    |   |    |   |
CALL CTPMV( 'U' , 'C' , 'U' , 4 , AP , X , 1  )

AP       =  ( . , (2.0, 2.0), . , (3.0, 3.0), (2.0, 2.0), . ,
              (2.0, 2.0), (5.0, 5.0), (3.0, 3.0), . )
X        =  ((5.0, 5.0), (4.0, 4.0), (3.0, 3.0), (2.0, 2.0))
```

Output

```
X        =  ((5.0, 5.0), (24.0, 4.0), (49.0, 3.0), (80.0, 2.0))
```

STBMV, DTBMV, CTBMV, and ZTBMV—Matrix-Vector Product for a Triangular Band Matrix, Its Transpose, or Its Conjugate Transpose

STBMV and DTBMV compute one of the following matrix-vector products, using the vector x and triangular band matrix A or its transpose:

$x \leftarrow Ax$
$x \leftarrow A^{T}x$

CTBMV and ZTBMV compute one of the following matrix-vector products, using the vector x and triangular band matrix A, its transpose, or its conjugate transpose:

$x \leftarrow Ax$
$x \leftarrow A^{T}x$
$x \leftarrow A^{H}x$

Matrix A can be either upper or lower triangular and is stored in upper- or lower-triangular-band-packed storage mode, respectively.

Table 70. Data Types

A, x	Subprogram
Short-precision real	STBMV
Long-precision real	DTBMV
Short-precision complex	CTBMV
Long-precision complex	ZTBMV

Syntax

Fortran	CALL STBMV | DTBMV | CTBMV | ZTBMV (*uplo*, *transa*, *diag*, *n*, *k*, *a*, *lda*, *x*, *incx*)
C and C++	stbmv | dtbmv | ctbmv | ztbmv (*uplo*, *transa*, *diag*, *n*, *k*, *a*, *lda*, *x*, *incx*);
PL/I	CALL STBMV | DTBMV | CTBMV | ZTBMV (*uplo*, *transa*, *diag*, *n*, *k*, *a*, *lda*, *x*, *incx*);

On Entry

uplo
indicates whether matrix A is an upper or lower triangular band matrix, where:

If *uplo* = 'U', A is an upper triangular matrix.

If *uplo* = 'L', A is a lower triangular matrix.

Specified as: a single character. It must be 'U' or 'L'.

transa
indicates the form of matrix A to use in the computation, where:

If *transa* = 'N', A is used in the computation.

If *transa* = 'T', A^{T} is used in the computation.

If *transa* = 'C', A^{H} is used in the computation.

Specified as: a single character. It must be 'N', 'T', or 'C'.

diag
indicates the characteristics of the diagonal of matrix ***A***, where:

If *diag* = 'U', ***A*** is a unit triangular matrix.

If *diag* = 'N', ***A*** is not a unit triangular matrix.

Specified as: a single character. It must be 'U' or 'N'.

n
is the order of triangular band matrix ***A***. Specified as: a fullword integer; $n \geq 0$.

k
is the upper or lower band width *k* of the matrix ***A***. Specified as: a fullword integer; $k \geq 0$.

a
is the upper or lower triangular band matrix ***A*** of order *n*, stored in upper- or lower-triangular-band-packed storage mode, respectively.

Note: No data should be moved to form $\boldsymbol{A}^{T}$ or $\boldsymbol{A}^{H}$; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 70 on page 359.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and $lda \geq k+1$.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 70 on page 359.

incx
is the stride for vector ***x***. Specified as: a fullword integer; $incx > 0$ or $incx < 0$.

On Return

x
is the vector ***x*** of length *n*, containing the results of the computation. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 70 on page 359.

Notes

1. These subroutines accept lowercase letters for the *uplo*, *transa*, and *diag* arguments.
2. For STBMV and DTBMV, if you specify 'C' for the *transa* argument, it is interpreted as though you specified 'T'.
3. Matrix ***A*** and vector ***x*** must have no common elements; otherwise, results are unpredictable.
4. To achieve optimal performance in these subroutines, use $lda = k+1$.
5. For unit triangular matrices, the elements of the diagonal are assumed to be 1.0 for real matrices and (1.0, 0.0) for complex matrices. As a result, you do not have to set these values.
6. For both upper and lower triangular band matrices, if you specify $k \geq n$, ESSL assumes, **only for purposes of the computation**, that the upper or lower band width of matrix ***A*** is $n-1$; that is, it processes matrix ***A***, of order *n*, as though it is a (nonbanded) triangular matrix. However, ESSL uses the original value for *k* **for the purposes of finding the locations** of element a_{11} and all other elements in the array

specified for ***A***, as described in "Triangular Band Matrix" on page 86. For an illustration of this technique, see "Example 4" on page 364.

7. For a description of triangular band matrices and how they are stored in upper- and lower-triangular-band-packed storage mode, see "Triangular Band Matrix" on page 86.
8. If you are using a lower triangular band matrix, you may want to use this alternate approach instead of using lower-triangular-band-packed storage mode. Leave matrix ***A*** in full-matrix storage mode when you pass it to ESSL and specify the *lda* argument to be *lda*+1, which is the leading dimension of matrix ***A*** plus 1. ESSL then processes the matrix elements in the same way as though you had set them up in lower-triangular-band-packed storage mode.

Function: These subroutines can perform the following matrix-vector product computations, using the triangular band matrix ***A***, its transpose, or its conjugate transpose, where ***A*** can be either upper or lower triangular:

$\boldsymbol{x} \leftarrow \boldsymbol{A}\boldsymbol{x}$
$\boldsymbol{x} \leftarrow \boldsymbol{A}^{T}\boldsymbol{x}$
$\boldsymbol{x} \leftarrow \boldsymbol{A}^{H}\boldsymbol{x}$ (for CTBMV and ZTBMV only)

where:

x is a vector of length *n*.
A is an upper or lower triangular band matrix of order *n*, stored in upper- or lower-triangular-band-packed storage mode, respectively.

See references [34], [46], and [38]. If *n* is 0, no computation is performed.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. *uplo* ≠ 'L' or 'U'
2. *transa* ≠ 'T', 'N', or 'C'
3. *diag* ≠ 'N' or 'U'
4. $n < 0$
5. $k < 0$
6. $lda \leq 0$
7. $lda < k+1$
8. $incx = 0$

Example 1: This example shows the computation $\boldsymbol{x} \leftarrow \boldsymbol{Ax}$. Matrix $\boldsymbol{A}$ is a real 7 by 7 upper triangular band matrix with a half band width of 3 that is not unit triangular, stored in upper-triangular-band-packed storage mode. Vector $\boldsymbol{x}$ is a vector of length 7. Matrix $\boldsymbol{A}$ is:

```
⌈                                                  ⌉
| 1.0  1.0  1.0  1.0  0.0  0.0  0.0 |
| 0.0  2.0  2.0  2.0  2.0  0.0  0.0 |
| 0.0  0.0  3.0  3.0  3.0  3.0  0.0 |
| 0.0  0.0  0.0  4.0  4.0  4.0  4.0 |
| 0.0  0.0  0.0  0.0  5.0  5.0  5.0 |
| 0.0  0.0  0.0  0.0  0.0  6.0  6.0 |
| 0.0  0.0  0.0  0.0  0.0  0.0  7.0 |
⌊                                                  ⌋
```

Call Statement and Input

```
            UPLO TRANSA DIAG  N   K   A  LDA  X  INCX
             |     |     |    |   |   |   |   |   |
CALL STBMV( 'U' , 'N' , 'N' , 7 , 3 , A , 5 , X , 1  )

        ⌈                                        ⌉
        |  .    .    .   1.0  2.0  3.0  4.0 |
        |  .    .   1.0  2.0  3.0  4.0  5.0 |
A    =  |  .   1.0  2.0  3.0  4.0  5.0  6.0 |
        | 1.0  2.0  3.0  4.0  5.0  6.0  7.0 |
        |  .    .    .    .    .    .    .  |
        ⌊                                        ⌋

X        =  (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0)
```

Output

```
X        =  (10.0, 28.0, 54.0, 88.0, 90.0, 78.0, 49.0)
```

Example 2: This example shows the computation $\boldsymbol{x} \leftarrow \boldsymbol{A}^{\mathrm{T}}\boldsymbol{x}$. Matrix $\boldsymbol{A}$ is a real 7 by 7 lower triangular band matrix with a half band width of 3 that is not unit triangular, stored in lower-triangular-band-packed storage mode. Vector $\boldsymbol{x}$ is a vector of length 7. Matrix $\boldsymbol{A}$ is:

```
⌈                                                  ⌉
| 1.0  0.0  0.0  0.0  0.0  0.0  0.0 |
| 1.0  2.0  0.0  0.0  0.0  0.0  0.0 |
| 1.0  2.0  3.0  0.0  0.0  0.0  0.0 |
| 1.0  2.0  3.0  4.0  0.0  0.0  0.0 |
| 0.0  2.0  3.0  4.0  5.0  0.0  0.0 |
| 0.0  0.0  3.0  4.0  5.0  6.0  0.0 |
| 0.0  0.0  0.0  4.0  5.0  6.0  7.0 |
⌊                                                  ⌋
```

Call Statement and Input

```
            UPLO TRANSA DIAG  N   K   A  LDA  X  INCX
             |     |     |    |   |   |   |   |   |
CALL STBMV( 'L' , 'T' , 'N' , 7 , 3 , A , 5 , X , 1  )
```

$$A = \begin{bmatrix} 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & . \\ 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & . & . \\ 1.0 & 2.0 & 3.0 & 4.0 & . & . & . \\ . & . & . & . & . & . & . \end{bmatrix}$$

$$X = (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0)$$

Output

$$X = (10.0, 28.0, 54.0, 88.0, 90.0, 78.0, 49.0)$$

Example 3: This example shows the computation $x \leftarrow A^{H}x$. Matrix A is a complex 7 by 7 upper triangular band matrix with a half band width of 3 that is not unit triangular, stored in upper-triangular-band-packed storage mode. Vector x is a vector of length 7. Matrix A is:

$$\begin{bmatrix} (1.0, 1.0) & (1.0, 1.0) & (1.0, 1.0) & (1.0, 1.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) \\ (0.0, 0.0) & (2.0, 2.0) & (2.0, 2.0) & (2.0, 2.0) & (2.0, 2.0) & (0.0, 0.0) & (0.0, 0.0) \\ (0.0, 0.0) & (0.0, 0.0) & (3.0, 3.0) & (3.0, 3.0) & (3.0, 3.0) & (3.0, 3.0) & (0.0, 0.0) \\ (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (4.0, 4.0) & (4.0, 4.0) & (4.0, 4.0) & (4.0, 4.0) \\ (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (5.0, 5.0) & (5.0, 5.0) & (5.0, 5.0) \\ (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (6.0, 6.0) & (6.0, 6.0) \\ (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (7.0, 7.0) \end{bmatrix}$$

Call Statement and Input

```
            UPLO TRANSA DIAG  N   K   A  LDA  X  INCX
             |     |     |    |   |   |   |   |   |
CALL CTBMV( 'U' , 'C' , 'N' , 7 , 3 , A , 5 , X , 1  )
```

$$A = \begin{bmatrix} . & . & . & (1.0, 1.0) & (2.0, 2.0) & (3.0, 3.0) & (4.0, 4.0) \\ . & . & (1.0, 1.0) & (2.0, 2.0) & (3.0, 3.0) & (4.0, 4.0) & (5.0, 5.0) \\ . & (1.0, 1.0) & (2.0, 2.0) & (3.0, 3.0) & (4.0, 4.0) & (5.0, 5.0) & (6.0, 6.0) \\ (1.0, 1.0) & (2.0, 2.0) & (3.0, 3.0) & (4.0, 4.0) & (5.0, 5.0) & (6.0, 6.0) & (7.0, 7.0) \\ . & . & . & . & . & . & . \end{bmatrix}$$

$$X = ((1.0, 2.0), (2.0, 4.0), (3.0, 6.0), (4.0, 8.0), (5.0, 10.0), (6.0, 12.0), (7.0, 14.0))$$

Output

```
X       =  ((1.0, 2.0), (7.0, 9.0), (24.0, 23.0), (58.0, 46.0),
            (112.0, 79.0), (186.0, 122.0), (280.0, 175.0))
```

Example 4: This example shows the computation $\boldsymbol{x} \leftarrow \boldsymbol{A}^{T}\boldsymbol{x}$, where $k > n$. Matrix $\boldsymbol{A}$ is a real 4 by 4 upper triangular band matrix with a half band width of 5 that is not unit triangular, stored in upper-triangular-band-packed storage mode. Vector $\boldsymbol{x}$ is a vector of length 4. Matrix $\boldsymbol{A}$ is:

$$\begin{bmatrix} 1.0 & 1.0 & 1.0 & 1.0 \\ . & 2.0 & 2.0 & 2.0 \\ . & . & 3.0 & 3.0 \\ . & . & . & 4.0 \end{bmatrix}$$

Call Statement and Input

```
           UPLO TRANSA DIAG  N   K   A  LDA  X  INCX
            |     |     |    |   |   |   |   |   |
CALL STBMV( 'U' , 'T' , 'N' , 4 , 5 , A , 6 , X , 1  )
```

$$A = \begin{bmatrix} . & . & . & . \\ . & . & . & . \\ . & . & . & 1.0 \\ . & . & 1.0 & 2.0 \\ . & 1.0 & 2.0 & 3.0 \\ 1.0 & 2.0 & 3.0 & 4.0 \end{bmatrix}$$

```
X       =  (1.0, 2.0, 3.0, 4.0)
```

Output

```
X       =  (1.0, 5.0, 14.0, 30.0)
```

Sparse Matrix-Vector Subprograms

This section contains the sparse matrix-vector subprogram descriptions.

DSMMX—Matrix-Vector Product for a Sparse Matrix in Compressed-Matrix Storage Mode

This subprogram computes the matrix-vector product for sparse matrix ***A***, stored in compressed-matrix storage mode, using the matrix and vectors ***x*** and ***y***:

$$\boldsymbol{y} \leftarrow \boldsymbol{A}\boldsymbol{x}$$

where ***A***, ***x***, and ***y*** contain long-precision real numbers. You can use DSMTM to transpose matrix ***A*** before calling this subroutine. The resulting computation performed by this subroutine is then $\boldsymbol{y} \leftarrow \boldsymbol{A}^{\mathrm{T}}\boldsymbol{x}$.

Syntax

Fortran	CALL DSMMX (*m*, *nz*, *ac*, *ka*, *lda*, *x*, *y*)
C and C++	dsmmx (*m*, *nz*, *ac*, *ka*, *lda*, *x*, *y*);
PL/I	CALL DSMMX (*m*, *nz*, *ac*, *ka*, *lda*, *x*, *y*);

On Entry

m
is the number of rows in sparse matrix ***A*** and the number of elements in vector ***y***. Specified as: a fullword integer; $m \geq 0$.

nz
is the maximum number of nonzero elements in each row of sparse matrix ***A***. Specified as: a fullword integer; $nz \geq 0$.

ac
is the *m* by *n* sparse matrix ***A***, stored in compressed-matrix storage mode in an array, referred to as AC. Specified as: an *lda* by (at least) *nz* array, containing long-precision real numbers.

ka
is the array, referred to as KA, containing the column numbers of the matrix ***A*** elements stored in the corresponding positions in array AC. Specified as: an *lda* by (at least) *nz* array, containing fullword integers, where $1 \leq$ (elements of KA) $\leq n$.

lda
is the size of the leading dimension of the arrays specified for *ac* and *ka*. Specified as: a fullword integer; $lda > 0$ and $lda \geq m$.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length *n*, containing long-precision real numbers.

y
See On Return.

On Return

y
is the vector ***y*** of length *m*, containing the result of the computation. Returned as: a one-dimensional array of (at least) length *m*, containing long-precision real numbers.

Notes

1. Matrix ***A*** must have no common elements with vectors ***x*** and ***y***; otherwise, results are unpredictable.

2. For the KA array, where there are no corresponding nonzero elements in AC, you must still fill in a number between 1 and *n*. See the "Example" on page 368.
3. For a description of how sparse matrices are stored in compressed-matrix storage mode, see "Compressed-Matrix Storage Mode" on page 93.
4. If your sparse matrix is stored by rows, as defined in "Storage-by-Rows" on page 99, you should first use the DSRSM utility subroutine, described in "DSRSM—Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode" on page 986, to convert your sparse matrix to compressed-matrix storage mode.

Function: The matrix-vector product is computed for a sparse matrix, stored in compressed matrix mode:

$$\boldsymbol{y} \leftarrow \boldsymbol{A}\boldsymbol{x}$$

where:

A is an *m* by *n* sparse matrix, stored in compressed-matrix storage mode in arrays AC and KA.
x is a vector of length *n*.
y is a vector of length *m*.

It is expressed as follows:

$$\begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_m \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} & \ldots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{m1} & \ldots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix}$$

See reference [67]. If *m* is 0, no computation is performed; if *nz* is 0, output vector ***y*** is set to zero, because matrix ***A*** contains all zeros.

If your program uses a sparse matrix stored by rows and you want to use this subroutine, you should first convert your sparse matrix to compressed-matrix storage mode by using the DSRSM utility subroutine described in "DSRSM—Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode" on page 986.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. $m < 0$
2. $lda \leq 0$
3. $m > lda$
4. $nz < 0$

Example: This example shows the matrix-vector product computed for the following sparse matrix ***A***, which is stored in compressed-matrix storage mode in arrays AC and KA. Matrix ***A*** is:

```
⎡                              ⎤
| 4.0  0.0  7.0  0.0  0.0  0.0 |
| 3.0  4.0  0.0  2.0  0.0  0.0 |
| 0.0  2.0  4.0  0.0  4.0  0.0 |
| 0.0  0.0  7.0  4.0  0.0  1.0 |
| 1.0  0.0  0.0  3.0  4.0  0.0 |
| 1.0  1.0  0.0  0.0  3.0  4.0 |
⎣                              ⎦
```

Call Statement and Input

```
            M   NZ  AC   KA  LDA  X   Y
            |   |   |    |    |   |   |
CALL DSMMX( 6 , 4 , AC , KA , 6 , X , Y )

        ⎡                    ⎤
        | 4.0  7.0  0.0  0.0 |
        | 4.0  3.0  2.0  0.0 |
AC   =  | 4.0  2.0  4.0  0.0 |
        | 4.0  7.0  1.0  0.0 |
        | 4.0  1.0  3.0  0.0 |
        | 4.0  1.0  1.0  3.0 |
        ⎣                    ⎦

        ⎡            ⎤
        | 1  3  1  1 |
        | 2  1  4  1 |
KA   =  | 3  2  5  1 |
        | 4  3  6  1 |
        | 5  1  4  1 |
        | 6  1  2  5 |
        ⎣            ⎦

X        =  (1.0, 2.0, 3.0, 4.0, 5.0, 6.0)
```

Output

```
Y        =  (25.0, 19.0, 36.0, 43.0, 33.0, 42.0)
```

DSMTM—Transpose a Sparse Matrix in Compressed-Matrix Storage Mode

This subprogram transposes sparse matrix ***A***, stored in compressed-matrix storage mode, where ***A*** contains long-precision real numbers.

Syntax

Fortran	CALL DSMTM (*m*, *nz*, *ac*, *ka*, *lda*, *n*, *nt*, *at*, *kt*, *ldt*, *aux*, *naux*)
C and C++	dsmtm (*m*, *nz*, *ac*, *ka*, *lda*, *n*, *nt*, *at*, *kt*, *ldt*, *aux*, *naux*);
PL/I	CALL DSMTM (*m*, *nz*, *ac*, *ka*, *lda*, *n*, *nt*, *at*, *kt*, *ldt*, *aux*, *naux*);

On Entry

m
is the number of rows in sparse matrix ***A***. Specified as: a fullword integer; $m \geq 0$.

nz
is the maximum number of nonzero elements in each row of sparse matrix ***A***. Specified as: a fullword integer; $nz \geq 0$.

ac
is the *m* by *n* sparse matrix ***A***, stored in compressed-matrix storage mode in an array, referred to as AC. Specified as: an *lda* by (at least) *nz* array, containing long-precision real numbers.

ka
is the array, referred to as KA, containing the column numbers of the matrix ***A*** elements stored in the corresponding positions in array AC. Specified as: an *lda* by (at least) *nz* array, containing fullword integers, where $1 \leq$ (elements of KA) $\leq n$.

lda
is the size of the leading dimension of the arrays specified for *ac* and *ka*. Specified as: a fullword integer; $lda > 0$ and $lda \geq m$.

n
is the number of columns in sparse matrix ***A***. Specified as: a fullword integer; $0 \leq n \leq ldt$ and $n \geq$ (maximum column index in KA).

nt
is the number of columns in output arrays AT and KT that are available for use. Specified as: a fullword integer; $nt > 0$.

at
See On Return.

kt
See On Return.

ldt
is the size of the leading dimension of the arrays specified for *at* and *kt*. Specified as: a fullword integer; $ldt > 0$ and $ldt \geq n$.

aux
has the following meaning:

If $naux = 0$ and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is a storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing long-precision real numbers. They can have any value.

naux
is the size of the work area specified by *aux*—that is, the number of elements in *aux*. Specified as: a fullword integer, where:

If *naux* = 0 and error 2015 is unrecoverable, DSMTM dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux* ≥ *n*.

On Return

n
is the number of rows in the transposed matrix A^T. Returned as: a fullword integer; *n* = (maximum column index in KA).

nt
is the maximum number of nonzero elements, *nt*, in each row of the transposed matrix A^T. Returned as: a fullword integer; *nt* ≤ *m*.

at
is the *n* by (at least) *m* sparse matrix transpose A^T, stored in compressed-matrix storage mode in an array, referred to as AT. Returned as: an *ldt* by (at least) *nt* array, containing long-precision real numbers.

kt
is the array, referred to as KT, containing the column numbers of the transposed matrix A^T elements, stored in the corresponding positions in array AT. Returned as: an *ldt* by (at least) *nt* array, containing fullword integers, where 1 ≤ (elements of KT) ≤ *m*.

Notes

1. In your C program, arguments *n* and *nt* must be passed by reference.
2. The value specified for input argument *nt* should be greater than or equal to the number of nonzero elements you estimate to be in each row of the transposed sparse matrix A^T. The output value is less than or equal to the input value you specify.
3. For the KA array, where there are no corresponding nonzero elements in AC, you must still fill in a number between 1 and *n*. See the "Example" on page 371.
4. For a description of how sparse matrices are stored in compressed-matrix storage mode, see "Compressed-Matrix Storage Mode" on page 93.
5. If your sparse matrix is stored by rows, as defined in "Storage-by-Rows" on page 99, you should first use the DSRSM utility subroutine, described in "DSRSM—Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode" on page 986, to convert your sparse matrix to compressed-matrix storage mode.
6. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: A sparse matrix A, stored in arrays AC and KA in compressed-matrix storage mode, is transposed, forming A^T, and is stored in arrays AT and KT in compressed-matrix storage mode. See reference [67]. This subroutine is provided for when you want to do a matrix-vector product using a transposed matrix, A^T. First, you transpose a matrix, A, using this subroutine, then you call DSMMX with the transposed matrix A^T. This results in the following computation being performed: $y \leftarrow A^T x$.

If your program uses a sparse matrix stored by rows and you want to use this subroutine, you should first convert your sparse matrix to compressed-matrix storage mode by using the DSRSM utility subroutine described in "DSRSM—Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode" on page 986.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors: None

Input-Argument Errors

1. *m*, *n* < 0
2. *lda*, *ldt* < 1
3. *lda* < *m*
4. *ldt* < *n*
5. *nz* < 0
6. *n* is less than the maximum column index in KA.
7. *nt* or *ldt* are too small.
8. When the following two errors occur, arrays AT, KT, and AUX are overwritten:
   naux < *n*
   nt ≤ 0
9. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example: This example shows how to transpose the following 5 by 4 sparse matrix ***A***, which is stored in compressed-matrix storage mode in arrays AC and KA. Matrix ***A*** is:

$$\begin{bmatrix} 11.0 & 0.0 & 0.0 & 0.0 \\ 21.0 & 0.0 & 23.0 & 0.0 \\ 0.0 & 0.0 & 33.0 & 34.0 \\ 0.0 & 42.0 & 0.0 & 44.0 \\ 51.0 & 0.0 & 53.0 & 0.0 \end{bmatrix}$$

The resulting 4 by 5 matrix transpose $\boldsymbol{A}^{\mathrm{T}}$, stored in compressed-matrix storage mode in arrays AT and KT, is as follows. Matrix $\boldsymbol{A}^{\mathrm{T}}$ is:

$$\begin{bmatrix} 11.0 & 21.0 & 0.0 & 0.0 & 51.0 \\ 0.0 & 0.0 & 0.0 & 42.0 & 0.0 \\ 0.0 & 23.0 & 33.0 & 0.0 & 53.0 \\ 0.0 & 0.0 & 34.0 & 44.0 & 0.0 \end{bmatrix}$$

As shown here, the value of N is larger than the actual number of columns in the matrix ***A***. On output, the exact number of rows in the transposed matrix is returned in the output argument N.

On output, row 6 of AT and KT is is not accessed or modified by the subroutine. Column 4 and row 5 are accessed and modified. They are of no use in further computations and will not be used, because NT = 3 and M = 4.

Call Statement and Input

```
           M   NZ  AC   KA  LDA  N   NT  AT   KT   LDT  AUX  NAUX
           |   |   |    |    |   |   |   |    |    |    |    |
CALL DSMTM( 5 , 2 , AC , KA , 5 , 5 , 4 , AT , KT ,  6 , AUX , 5  )
```

```
        ┌              ┐
        │ 11.0   0.0   │
        │ 21.0  23.0   │
AC   =  │ 33.0  34.0   │
        │ 42.0  44.0   │
        │ 51.0  53.0   │
        └              ┘
```

```
        ┌      ┐
        │ 1  1 │
        │ 1  3 │
KA   =  │ 3  4 │
        │ 2  4 │
        │ 1  3 │
        └      ┘
```

Output

```
N        =  4
NT       =  3
```

```
        ┌                      ┐
        │ 11.0  21.0  51.0  0.0 │
        │ 42.0   0.0   0.0  0.0 │
AT   =  │ 33.0  23.0  53.0  0.0 │
        │ 34.0  44.0   0.0  0.0 │
        │  0.0   0.0   0.0  0.0 │
        │   .     .     .    .  │
        └                      ┘
```

```
        ┌              ┐
        │ 1  2  5  1   │
        │ 4  1  1  1   │
KT   =  │ 3  2  5  1   │
        │ 3  4  1  1   │
        │ 1  1  1  1   │
        │ .  .  .  .   │
        └              ┘
```

DSDMX—Matrix-Vector Product for a Sparse Matrix or Its Transpose in Compressed-Diagonal Storage Mode

This subprogram computes the matrix-vector product for square sparse matrix ***A***, stored in compressed-diagonal storage mode, using either the matrix or its transpose, and vectors ***x*** and ***y***:

$$\boldsymbol{y} \leftarrow \boldsymbol{A}\boldsymbol{x}$$
$$\boldsymbol{y} \leftarrow \boldsymbol{A}^{\mathrm{T}}\boldsymbol{x}$$

where ***A***, ***x***, and ***y*** contain long-precision real numbers.

Syntax

Fortran	CALL DSDMX (*iopt*, *n*, *nd*, *ad*, *lda*, *trans*, *la*, *x*, *y*)
C and C++	dsdmx (*iopt*, *n*, *nd*, *ad*, *lda*, *trans*, *la*, *x*, *y*);
PL/I	CALL DSDMX (*iopt*, *n*, *nd*, *ad*, *lda*, *trans*, *la*, *x*, *y*);

On Entry

iopt
indicates the storage variation used for sparse matrix ***A***, stored in compressed-diagonal storage mode, where:

If *iopt* = 0, matrix ***A*** is a general sparse matrix, where all the nonzero diagonals in matrix ***A*** are used to set up the storage arrays.

If *iopt* = 1, matrix ***A*** is a symmetric sparse matrix, where only the nonzero main diagonal and one of each of the unique nonzero diagonals are used to set up the storage arrays.

Specified as: a fullword integer; *iopt* = 0 or 1.

n
is the order of sparse matrix ***A*** and the number of elements in vectors ***x*** and ***y***. Specified as: a fullword integer; $n \geq 0$.

nd
is the number of diagonals stored in the columns of array AD, as well as the number of columns in AD and the number of elements in array LA. Specified as: a fullword integer; $nd \geq 0$.

ad
is the sparse matrix ***A*** of order *n*, stored in compressed diagonal storage in an array, referred to as AD. The *iopt* argument indicates the storage variation used for storing matrix ***A***. The *trans* argument indicates the following:

If *trans* = 'N', ***A*** is used in the computation.

If *trans* = 'T', $\boldsymbol{A}^{\mathrm{T}}$ is used in the computation.

Note: No data should be moved to form $\boldsymbol{A}^{\mathrm{T}}$; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: an *lda* by (at least) *nd* array, containing long-precision real numbers; $lda \geq n$.

lda
is the size of the leading dimension of the array specified for *ad*. Specified as: a fullword integer; $lda > 0$ and $lda \geq n$.

trans
indicates the form of matrix ***A*** to use in the computation, where:

If *trans* = 'N', ***A*** is used in the computation.

If *trans* = 'T', $\boldsymbol{A}^T$ is used in the computation.

Specified as: a single character; *trans* = 'N' or 'T'.

la
is the array, referred to as LA, containing the diagonal numbers *k* for the diagonals stored in each corresponding column in array AD. (For an explanation of how diagonal numbers are assigned, see "Compressed-Diagonal Storage Mode" on page 94.)

Specified as: a one-dimensional array of (at least) length *nd*, containing fullword integers; $1-n \le LA(i) \le n-1$.

x
is the vector ***x*** of length *n*. Specified as: a one-dimensional array, containing long-precision real numbers.

y
See On Return.

On Return

y
is the vector ***y*** of length *n*, containing the result of the computation. Returned as: a one-dimensional array, containing long-precision real numbers.

Notes

1. All subroutines accept lowercase letters for the *trans* argument.
2. Matrix ***A*** must have no common elements with vectors ***x*** and ***y***; otherwise, results are unpredictable.
3. For a description of how sparse matrices are stored in compressed-diagonal storage mode, see "Compressed-Diagonal Storage Mode" on page 94.

Function: The matrix-vector product of a square sparse matrix or its transpose, is computed for a matrix stored in compressed-diagonal storage mode:

$$\boldsymbol{y} \leftarrow \boldsymbol{Ax}$$
$$\boldsymbol{y} \leftarrow \boldsymbol{A}^T\boldsymbol{x}$$

where:

A is a sparse matrix of order *n*, stored in compressed-diagonal storage mode in AD and LA, using the storage variation for either general or symmetric sparse matrices, as indicated by the *iopt* argument.
x and ***y*** are vectors of length *n*.

It is expressed as follows for $\boldsymbol{y} \leftarrow \boldsymbol{Ax}$:

$$\begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} & \dots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{n1} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix}$$

It is expressed as follows for $\boldsymbol{y} \leftarrow \boldsymbol{A}^{\mathrm{T}}\boldsymbol{x}$:

$$\begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} & \dots & a_{n1} \\ . & & . \\ . & & . \\ . & & . \\ a_{1n} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix}$$

If n is 0, no computation is performed; if nd is 0, output vector $\boldsymbol{y}$ is set to zero, because matrix $\boldsymbol{A}$ contains all zeros.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. $iopt \neq 0$ or 1
2. $n < 0$
3. $lda \leq 0$
4. $n > lda$
5. $trans \neq$ 'N' or 'T'
6. $nd < 0$
7. LA(j) $\leq -n$ or LA(j) $\geq n$, for any $j = 1, n$

Example 1: This example shows the matrix-vector product using *trans* = 'N', which is computed for the following sparse matrix $\boldsymbol{A}$ of order 6. The matrix is stored in compressed-matrix storage mode in arrays AD and LA using the storage variation for general sparse matrices, storing all nonzero diagonals. Matrix $\boldsymbol{A}$ is:

```
[                                    ]
| 4.0  0.0  7.0  0.0  0.0  0.0 |
| 3.0  4.0  0.0  2.0  0.0  0.0 |
| 0.0  2.0  4.0  0.0  4.0  0.0 |
| 0.0  0.0  7.0  4.0  0.0  1.0 |
| 1.0  0.0  0.0  3.0  4.0  0.0 |
| 1.0  1.0  0.0  0.0  3.0  4.0 |
[                                    ]
```

Call Statement and Input

```
           IOPT  N   ND  AD  LDA TRANS  LA   X   Y
            |    |   |   |    |    |    |    |   |
CALL DSDMX( 0  , 6 , 5 , AD , 6 , 'N' , LA , X , Y )
```

```
        ⎡                          ⎤
        ⎢ 4.0  0.0  0.0  0.0  7.0 ⎥
        ⎢ 4.0  0.0  0.0  3.0  2.0 ⎥
AD   =  ⎢ 4.0  0.0  0.0  2.0  4.0 ⎥
        ⎢ 4.0  0.0  0.0  7.0  1.0 ⎥
        ⎢ 4.0  0.0  1.0  3.0  0.0 ⎥
        ⎢ 4.0  1.0  1.0  3.0  0.0 ⎥
        ⎣                          ⎦

LA       =  (0, -5, -4, -1, 2)
X        =  (1.0, 2.0, 3.0, 4.0, 5.0, 6.0)
```

Output

```
Y        =  (25.0, 19.0, 36.0, 43.0, 33.0, 42.0)
```

Example 2: This example shows the matrix-vector product using *trans* = 'N', which is computed for the following sparse matrix ***A*** of order 6. The matrix is stored in compressed-matrix storage mode in arrays AD and LA using the storage variation for symmetric sparse matrices, storing the nonzero main diagonal and one of each of the unique nonzero diagonals. Matrix ***A*** is:

```
⎡                                          ⎤
⎢ 11.0   0.0  13.0   0.0  15.0   0.0 ⎥
⎢  0.0  22.0   0.0  24.0   0.0  26.0 ⎥
⎢ 13.0   0.0  33.0   0.0  35.0   0.0 ⎥
⎢  0.0  24.0   0.0  44.0   0.0  46.0 ⎥
⎢ 15.0   0.0  35.0   0.0  55.0   0.0 ⎥
⎢  0.0  26.0   0.0  46.0   0.0  66.0 ⎥
⎣                                          ⎦
```

Call Statement and Input

```
           IOPT  N   ND  AD  LDA TRANS  LA   X   Y
            |    |   |   |    |    |    |    |   |
CALL DSDMX( 1  , 6 , 3 , AD , 6 , 'N' , LA , X , Y )

        ⎡                  ⎤
        ⎢ 11.0  13.0   0.0 ⎥
        ⎢ 22.0  24.0   0.0 ⎥
AD   =  ⎢ 33.0  35.0   0.0 ⎥
        ⎢ 44.0  46.0   0.0 ⎥
        ⎢ 55.0   0.0  15.0 ⎥
        ⎢ 66.0   0.0  26.0 ⎥
        ⎣                  ⎦

LA       =  (0, 2, -4)
X        =  (1.0, 2.0, 3.0, 4.0, 5.0, 6.0)
```

Output

```
Y        =  (125.0, 296.0, 287.0, 500.0, 395.0, 632.0)
```

Example 3: This example is the same as Example 1 except that it shows the matrix-vector product for the transpose of a matrix, using *trans* = 'T'. It is computed using the transpose of the following sparse matrix ***A*** of order 6, which is stored in compressed-matrix storage mode in arrays AD and LA, using the storage variation for general sparse matrices, storing all nonzero diagonals. It uses the same matrix ***A*** as in Example 1.

Call Statement and Input

```
           IOPT  N   ND  AD  LDA TRANS  LA   X   Y
            |    |   |   |    |    |    |    |   |
CALL DSDMX( 0  , 6 , 5 , AD , 6 , 'T' , LA , X , Y )
```

```
AD        =(same as input AD in Example 1)
LA        =(same as input LA in Example 1)
X         =(same as input X in Example 1)
```

Output

```
Y         =  (21.0, 20.0, 47.0, 35.0, 50.0, 28.0)
```

DSDMX

Chapter 9. Matrix Operations

The matrix operation subroutines are described in this chapter.

Overview of the Matrix Operation Subroutines

Some of the matrix operation subroutines were designed in accordance with the Level 3 BLAS de facto standard. If these subroutines do not comply with the standard as approved, IBM will consider updating them to do so. If IBM updates these subroutines, the updates could require modifications of the calling application program. For details on the Level 3 BLAS, see reference [32]. The matrix operation subroutines also include the commonly used matrix operations: addition, subtraction, multiplication, and transposition (Table 71).

Table 71 (Page 1 of 2). List of Matrix Operation Subroutines

Descriptive Name	Short-Precision Subroutine	Long-Precision Subroutine	Page
Matrix Addition for General Matrices or Their Transposes	SGEADD CGEADD	DGEADD ZGEADD	384
Matrix Subtraction for General Matrices or Their Transposes	SGESUB CGESUB	DGESUB ZGESUB	391
Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes	SGEMUL CGEMUL	DGEMUL ZGEMUL DGEMLP§	398
Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes Using Winograd's Variation of Strassen's Algorithm	SGEMMS CGEMMS	DGEMMS ZGEMMS	408
Combined Matrix Multiplication and Addition for General Matrices, Their Transposes, or Conjugate Transposes	SGEMM♦ CGEMM♦	DGEMM♦ ZGEMM♦	413
Matrix-Matrix Product Where One Matrix is Real or Complex Symmetric or Complex Hermitian	SSYMM♦ CSYMM♦ CHEMM♦	DSYMM♦ ZSYMM♦ ZHEMM♦	422
Triangular Matrix-Matrix Product	STRMM♦ CTRMM♦	DTRMM♦ ZTRMM♦	430
Rank-K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix	SSYRK♦ CSYRK♦ CHERK♦	DSYRK♦ ZSYRK♦ ZHERK♦	437
Rank-2K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix	SSYR2K♦ CSYR2K♦ CHER2K♦	DSYR2K♦ ZSYR2K♦ ZHER2K♦	444
General Matrix Transpose (In-Place)	SGETMI CGETMI	DGETMI ZGETMI	452

© Copyright IBM Corp. 1997

Table 71 (Page 2 of 2). List of Matrix Operation Subroutines			
Descriptive Name	**Short-Precision Subroutine**	**Long-Precision Subroutine**	**Page**
General Matrix Transpose (Out-of-Place)	SGETMO CGETMO	DGETMO ZGETMO	455
♦ Level 3 BLAS § This subroutine is provided only for migration from earlier releases of ESSL and is not intended for use in new programs. Documentation for this subroutine is no longer provided.			

Use Considerations

This section describes some key points about using the matrix operations subroutines.

Specifying Normal, Transposed, or Conjugate Transposed Input Matrices

On each invocation, the matrix operation subroutines can perform one of several possible computations, using different forms of the input matrices ***A*** and ***B***. For the real and complex versions of the subroutines, there are four and nine combinations, respectively, depending on the characters specified for the *transa* and *transb* arguments:

'N' Normal form
'T' Transposed form subroutines)
'C' Conjugate transposed form

The four and nine possible combinations are defined as follows:

Real Combinations	Complex Combinations
$\boldsymbol{AB}$	$\boldsymbol{AB}$
$\boldsymbol{A}^{T}\boldsymbol{B}$	$\boldsymbol{A}^{T}\boldsymbol{B}$
	$\boldsymbol{A}^{H}\boldsymbol{B}$
$\boldsymbol{AB}^{T}$	$\boldsymbol{AB}^{T}$
$\boldsymbol{A}^{T}\boldsymbol{B}^{T}$	$\boldsymbol{A}^{T}\boldsymbol{B}^{T}$
	$\boldsymbol{A}^{H}\boldsymbol{B}^{T}$
	$\boldsymbol{AB}^{H}$
	$\boldsymbol{A}^{T}\boldsymbol{B}^{H}$
	$\boldsymbol{A}^{H}\boldsymbol{B}^{H}$

Transposing or Conjugate Transposing:

This section describes some key points about using transposed and conjugate transposed matrices.

On Input

In every case, the input arrays for the matrix, its transpose, or its conjugate transpose should be stored in the original untransposed form. You then specify the desired form of the matrix to be used in the computation in the *transa* or *transb* arguments. For a description of matrix transpose and matrix conjugate transpose, see "Matrices" on page 62.

On Output

If you want to compute the transpose or the conjugate transpose of a matrix operation—that is, the output stored in matrix ***C***—you should use the matrix identities described in Special Usage for each subroutine description. Examples are provided in the subroutine descriptions to show the use of these matrix identities. This accomplishes the transpose or conjugate transpose as part of the multiply operation.

Performance and Accuracy Considerations

This section describes some key points about performance and accuracy in the matrix operations subroutines.

In General

1. The matrix operation subroutines use algorithms that are tuned specifically to the workstation processors they run on. The techniques involve using any one of several computational methods, based on certain operation counts and sizes of data.
2. The short-precision multiplication subroutines provide increased accuracy by partially accumulating results in long precision.
3. Strassen's method is not stable for certain row or column scalings of the input matrices ***A*** and ***B***. Therefore, for matrices ***A*** and ***B*** with divergent exponent values, Strassen's method may give inaccurate results. For these cases, you should use the _GEMUL or _GEMM subroutines.
4. There are ESSL-specific rules that apply to the results of computations on the workstation processors using the ANSI/IEEE standards. For details, see "What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?" on page 44.

For Large Matrices

If you are using large square matrices in your matrix multiplication operations, you get better performance by using SGEMMS, DGEMMS, CGEMMS, and ZGEMMS. These subroutines use Winograd's variation of Strassen's algorithm for both real and complex matrices.

For Combined Operations

If you want to perform a combined matrix multiplication and addition with scaling, SGEMM, DGEMM, CGEMM, and ZGEMM provide better performance than if you perform the parts of the computation separately in your program. See references [32] and [35].

Matrix Operation Subroutines

This section contains the matrix operation subroutine descriptions.

SGEADD, DGEADD, CGEADD, and ZGEADD—Matrix Addition for General Matrices or Their Transposes

These subroutines can perform any one of the following matrix additions, using matrices ***A*** and ***B*** or their transposes, and matrix ***C***:

$$C \leftarrow A+B$$
$$C \leftarrow A^{T}+B$$
$$C \leftarrow A+B^{T}$$
$$C \leftarrow A^{T}+B^{T}$$

Table 72. Data Types

A, ***B***, ***C***	**Subroutine**
Short-precision real	SGEADD
Long-precision real	DGEADD
Short-precision complex	CGEADD
Long-precision complex	ZGEADD

Syntax

Fortran	CALL SGEADD | DGEADD | CGEADD | ZGEADD (*a*, *lda*, *transa*, *b*, *ldb*, *transb*, *c*, *ldc*, *m*, *n*)
C and C++	sgeadd | dgeadd | cgeadd | zgeadd (*a*, *lda*, *transa*, *b*, *ldb*, *transb*, *c*, *ldc*, *m*, *n*);
PL/I	CALL SGEADD | DGEADD | CGEADD | ZGEADD (*a*, *lda*, *transa*, *b*, *ldb*, *transb*, *c*, *ldc*, *m*, *n*);

On Entry

a
is the matrix ***A***, where:

If *transa* = 'N', ***A*** is used in the computation, and ***A*** has *m* rows and *n* columns.

If *transa* = 'T', A^{T} is used in the computation, and ***A*** has *n* rows and *m* columns.

Note: No data should be moved to form A^{T}; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 72, where:

If *transa* = 'N', its size must be *lda* by (at least) *n*.

If *transa* = 'T', its size must be *lda* by (at least) *m*.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; *lda* > 0 and:

If *transa* = 'N', *lda* ≥ *m*.

If *transa* = 'T', *lda* ≥ *n*.

transa
indicates the form of matrix ***A*** to use in the computation, where:

If *transa* = 'N', ***A*** is used in the computation.

If *transa* = 'T', $\boldsymbol{A}^{T}$ is used in the computation.

Specified as: a single character; *transa* = 'N' or 'T'.

b
is the matrix ***B***, where:

If *transb* = 'N', ***B*** is used in the computation, and ***B*** has *m* rows and *n* columns.

If *transb* = 'T', $\boldsymbol{B}^{T}$ is used in the computation, and ***B*** has *n* rows and *m* columns.

Note: No data should be moved to form $\boldsymbol{B}^{T}$; that is, the matrix ***B*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 72 on page 384, where:

If *transb* = 'N', its size must be *ldb* by (at least) *n*.

If *transb* = 'T', its size must be *ldb* by (at least) *m*.

ldb
is the leading dimension of the array specified for *b*. Specified as: a fullword integer; *ldb* > 0 and:

If *transb* = 'N', *ldb* ≥ *m*.

If *transb* = 'T', *ldb* ≥ *n*.

transb
indicates the form of matrix ***B*** to use in the computation, where:

If *transb* = 'N', ***B*** is used in the computation.

If *transb* = 'T', $\boldsymbol{B}^{T}$ is used in the computation.

Specified as: a single character; *transb* = 'N' or 'T'.

c
See On Return.

ldc
is the leading dimension of the array specified for *c*. Specified as: a fullword integer; *ldc* > 0 and *ldc* ≥ *m*.

m
is the number of rows in matrix ***C***. Specified as: a fullword integer; 0 ≤ *m* ≤ *ldc*.

n
is the number of columns in matrix ***C***. Specified as: a fullword integer; 0 ≤ *n*.

On Return

c
is the *m* by *n* matrix ***C***, containing the results of the computation. Returned as: an *ldc* by (at least) *n* array, containing numbers of the data type indicated in Table 72 on page 384.

Notes

1. All subroutines accept lowercase letters for the *transa* and *transb* arguments.

2. Matrix ***C*** must have no common elements with matrices ***A*** or ***B***. However, ***C*** may (exactly) coincide with ***A*** if *transa* = 'N', and ***C*** may (exactly) coincide with ***B*** if *transb* = 'N'. Otherwise, results are unpredictable. See "Concepts" on page 55.

Function: The matrix sum is expressed as follows, where a_{ij}, b_{ij}, and c_{ij} are elements of matrices ***A, B,*** and ***C***, respectively:

$c_{ij} = a_{ij}+b_{ij}$ for $\boldsymbol{C} \leftarrow \boldsymbol{A}+\boldsymbol{B}$
$c_{ij} = a_{ij}+b_{ji}$ for $\boldsymbol{C} \leftarrow \boldsymbol{A}+\boldsymbol{B}^T$
$c_{ij} = a_{ji}+b_{ij}$ for $\boldsymbol{C} \leftarrow \boldsymbol{A}^T+\boldsymbol{B}$
$c_{ij} = a_{ji}+b_{ji}$ for $\boldsymbol{C} \leftarrow \boldsymbol{A}^T+\boldsymbol{B}^T$
for $i = 1, m$ and $j = 1, n$

If m or n is 0, no computation is performed.

Special Usage: You can compute the transpose $\boldsymbol{C}^T$ of each of the four computations listed under " Function" by using the following matrix identities:

$(\boldsymbol{A}+\boldsymbol{B})^T = \boldsymbol{A}^T+\boldsymbol{B}^T$
$(\boldsymbol{A}+\boldsymbol{B}^T)^T = \boldsymbol{A}^T+\boldsymbol{B}$
$(\boldsymbol{A}^T+\boldsymbol{B})^T = \boldsymbol{A}+\boldsymbol{B}^T$
$(\boldsymbol{A}^T+\boldsymbol{B}^T)^T = \boldsymbol{A}+\boldsymbol{B}$

Be careful that your output array receiving $\boldsymbol{C}^T$ has dimensions large enough to hold the transposed matrix. See "Example 4" on page 388.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. *lda*, *ldb*, *ldc* ≤ 0
2. *m*, *n* < 0
3. *m* > *ldc*
4. *transa*, *transb* ≠ 'N' or 'T'
5. *transa* = 'N' and *m* > *lda*
6. *transa* = 'T' and *n* > *lda*
7. *transb* = 'N' and *m* > *ldb*
8. *transb* = 'T' and *n* > *ldb*

Example 1: This example shows the computation $\boldsymbol{C} \leftarrow \boldsymbol{A}+\boldsymbol{B}$, where ***A*** and ***C*** are contained in larger arrays A and C, respectively, and ***B*** is the same size as array B, in which it is contained.

Call Statement and Input

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  M   N
             |   |    |     |   |    |     |   |   |   |
CALL SGEADD( A , 6 , 'N'  , B , 4 , 'N'  , C , 5 , 4 , 3 )
```

```
        ⎡                                  ⎤
        ⎢ 110000.0  120000.0  130000.0 ⎥
        ⎢ 210000.0  220000.0  230000.0 ⎥
A    =  ⎢ 310000.0  320000.0  330000.0 ⎥
        ⎢ 410000.0  420000.0  430000.0 ⎥
        ⎢       .         .         . ⎥
        ⎢       .         .         . ⎥
        ⎣                                  ⎦

        ⎡                    ⎤
        ⎢ 11.0  12.0  13.0 ⎥
B    =  ⎢ 21.0  22.0  23.0 ⎥
        ⎢ 31.0  32.0  33.0 ⎥
        ⎢ 41.0  42.0  43.0 ⎥
        ⎣                    ⎦
```

Output

```
        ⎡                                  ⎤
        ⎢ 110011.0  120012.0  130013.0 ⎥
        ⎢ 210021.0  220022.0  230023.0 ⎥
C    =  ⎢ 310031.0  320032.0  330033.0 ⎥
        ⎢ 410041.0  420042.0  430043.0 ⎥
        ⎢       .         .         . ⎥
        ⎣                                  ⎦
```

Example 2: This example shows the computation $\boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{T}}+\boldsymbol{B}$, where ***A, B,*** and ***C*** are the same size as arrays A, B, and C, in which they are contained.

Call Statement and Input

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  M   N
             |   |    |     |   |    |     |   |   |   |
CALL SGEADD( A , 3 , 'T'  , B , 4 , 'N'  , C , 4 , 4 , 3 )

        ⎡                                          ⎤
        ⎢ 110000.0  120000.0  130000.0  140000.0 ⎥
A    =  ⎢ 210000.0  220000.0  230000.0  240000.0 ⎥
        ⎢ 310000.0  320000.0  330000.0  340000.0 ⎥
        ⎣                                          ⎦

        ⎡                    ⎤
        ⎢ 11.0  12.0  13.0 ⎥
B    =  ⎢ 21.0  22.0  23.0 ⎥
        ⎢ 31.0  32.0  33.0 ⎥
        ⎢ 41.0  42.0  43.0 ⎥
        ⎣                    ⎦
```

Output

```
        ⎡                                  ⎤
        ⎢ 110011.0  210012.0  310013.0 ⎥
C    =  ⎢ 120021.0  220022.0  320023.0 ⎥
        ⎢ 130031.0  230032.0  330033.0 ⎥
        ⎢ 140041.0  240042.0  340043.0 ⎥
        ⎣                                  ⎦
```

Example 3: This example shows computation $\boldsymbol{C} \leftarrow \boldsymbol{A}+\boldsymbol{B}^{\mathrm{T}}$, where $\boldsymbol{A}$ is contained in a larger array A, and $\boldsymbol{B}$ and $\boldsymbol{C}$ are the same size as arrays B and C, in which they are contained.

Call Statement and Input

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  M   N
             |   |    |     |   |    |     |   |   |   |
CALL SGEADD( A , 5 , 'N'  , B , 3 , 'T'  , C , 4 , 4 , 3 )
```

```
        ⎡                                ⎤
        ⎢ 110000.0  120000.0  130000.0 ⎥
        ⎢ 210000.0  220000.0  230000.0 ⎥
A    =  ⎢ 310000.0  320000.0  330000.0 ⎥
        ⎢ 410000.0  420000.0  430000.0 ⎥
        ⎢       .         .         . ⎥
        ⎣                                ⎦
```

```
        ⎡                          ⎤
        ⎢ 11.0  12.0  13.0  14.0 ⎥
B    =  ⎢ 21.0  22.0  23.0  24.0 ⎥
        ⎢ 31.0  32.0  33.0  34.0 ⎥
        ⎣                          ⎦
```

Output

```
        ⎡                                ⎤
        ⎢ 110011.0  120021.0  130031.0 ⎥
C    =  ⎢ 210012.0  220022.0  230032.0 ⎥
        ⎢ 310013.0  320023.0  330033.0 ⎥
        ⎢ 410014.0  420024.0  430034.0 ⎥
        ⎣                                ⎦
```

Example 4: This example shows how to produce the transpose of the result of the computation performed in "Example 3," $\boldsymbol{C} \leftarrow \boldsymbol{A}+\boldsymbol{B}^{\mathrm{T}}$, which uses the calling sequence:

```
CALL SGEADD( A , 5 , 'N' , B , 3 , 'T' , C , 4 , 4 , 3 )
```

You instead code a calling sequence for $\boldsymbol{C}^{\mathrm{T}} \leftarrow \boldsymbol{A}^{\mathrm{T}}+\boldsymbol{B}$, as shown below, where the resulting matrix $\boldsymbol{C}^{\mathrm{T}}$ in the output array CT is the transpose of the matrix in the output array C in Example 3. Note that the array CT has dimensions large enough to receive the transposed matrix. For a description of all the matrix identities, see "Special Usage" on page 386.

Call Statement and Input

```
             A  LDA TRANSA  B  LDB TRANSB  C   LDC  M   N
             |   |    |     |   |    |     |    |   |   |
CALL SGEADD( A , 5 , 'T'  , B , 3 , 'N'  , CT , 4 , 3 , 4 )
```

```
        ⎡                                ⎤
        ⎢ 110000.0  120000.0  130000.0 ⎥
        ⎢ 210000.0  220000.0  230000.0 ⎥
A    =  ⎢ 310000.0  320000.0  330000.0 ⎥
        ⎢ 410000.0  420000.0  430000.0 ⎥
        ⎢       .         .         . ⎥
        ⎣                                ⎦
```

```
          ┌                          ┐
          │ 11.0  12.0  13.0  14.0   │
B    =    │ 21.0  22.0  23.0  24.0   │
          │ 31.0  32.0  33.0  34.0   │
          └                          ┘
```

Output

```
          ┌                                  ┐
          │ 110011.0  210012.0  310013.0  410014.0 │
CT   =    │ 120021.0  220022.0  320023.0  420024.0 │
          │ 130031.0  230032.0  330033.0  430034.0 │
          │       .         .         .         .  │
          └                                  ┘
```

Example 5: This example shows the computation $C \leftarrow A^T+B^T$, where ***A, B,*** and ***C*** are the same size as the arrays A, B, and C, in which they are contained.

Call Statement and Input

```
              A  LDA TRANSA  B  LDB TRANSB  C  LDC  M   N
              |   |    |     |   |    |     |   |   |   |
CALL SGEADD( A , 3 , 'T'  , B , 3 , 'T'  , C , 4 , 4 , 3 )
```

```
          ┌                          ┐
          │ 110000.0  120000.0  130000.0  140000.0 │
A    =    │ 210000.0  220000.0  230000.0  240000.0 │
          │ 310000.0  320000.0  330000.0  340000.0 │
          └                          ┘
```

```
          ┌                          ┐
          │ 11.0  12.0  13.0  14.0   │
B    =    │ 21.0  22.0  23.0  24.0   │
          │ 31.0  32.0  33.0  34.0   │
          └                          ┘
```

Output

```
          ┌                       ┐
          │ 110011.0  210021.0  310031.0 │
C    =    │ 120012.0  220022.0  320032.0 │
          │ 130013.0  230023.0  330033.0 │
          │ 140014.0  240024.0  340034.0 │
          └                       ┘
```

Example 6: This example shows the computation $C \leftarrow A+B$, where ***A***, ***B***, and ***C*** are contained in larger arrays A, B, and C, respectively, and the arrays contain complex data.

Call Statement and Input

```
              A  LDA TRANSA  B  LDB TRANSB  C  LDC  M   N
              |   |    |     |   |    |     |   |   |   |
CALL CGEADD( A , 6 , 'N'  , B , 5 , 'N'  , C , 5 , 4 , 3 )
```

$$A = \begin{bmatrix} (1.0, 5.0) & (9.0, 2.0) & (1.0, 9.0) \\ (2.0, 4.0) & (8.0, 3.0) & (1.0, 8.0) \\ (3.0, 3.0) & (7.0, 5.0) & (1.0, 7.0) \\ (6.0, 6.0) & (3.0, 6.0) & (1.0, 4.0) \\ \vdots & \vdots & \vdots \end{bmatrix}$$

$$B = \begin{bmatrix} (1.0, 8.0) & (2.0, 7.0) & (3.0, 2.0) \\ (4.0, 4.0) & (6.0, 8.0) & (6.0, 3.0) \\ (6.0, 2.0) & (4.0, 5.0) & (4.0, 5.0) \\ (7.0, 2.0) & (6.0, 4.0) & (1.0, 6.0) \\ \vdots & \vdots & \vdots \end{bmatrix}$$

Output

$$C = \begin{bmatrix} (2.0, 13.0) & (11.0, 9.0) & (4.0, 11.0) \\ (6.0, 8.0) & (14.0, 11.0) & (7.0, 11.0) \\ (9.0, 5.0) & (11.0, 10.0) & (5.0, 12.0) \\ (13.0, 8.0) & (9.0, 10.0) & (2.0, 10.0) \\ \vdots & \vdots & \vdots \end{bmatrix}$$

SGESUB, DGESUB, CGESUB, and ZGESUB—Matrix Subtraction for General Matrices or Their Transposes

These subroutines can perform any one of the following matrix subtractions, using matrices ***A*** and ***B*** or their transposes, and matrix ***C***:

$$C \leftarrow A - B$$
$$C \leftarrow A^{T} - B$$
$$C \leftarrow A - B^{T}$$
$$C \leftarrow A^{T} - B^{T}$$

Table 73. Data Types

A, ***B***, ***C***	**Subroutine**
Short-precision real	SGESUB
Long-precision real	DGESUB
Short-precision complex	CGESUB
Long-precision complex	ZGESUB

Syntax

Fortran	CALL SGESUB | DGESUB | CGESUB | ZGESUB (*a*, *lda*, *transa*, *b*, *ldb*, *transb*, *c*, *ldc*, *m*, *n*)
C and C++	sgesub | dgesub | cgesub | zgesub (*a*, *lda*, *transa*, *b*, *ldb*, *transb*, *c*, *ldc*, *m*, *n*);
PL/I	CALL SGESUB | DGESUB | CGESUB | ZGESUB (*a*, *lda*, *transa*, *b*, *ldb*, *transb*, *c*, *ldc*, *m*, *n*);

On Entry

a

is the matrix ***A***, where:

If *transa* = 'N', ***A*** is used in the computation, and ***A*** has *m* rows and *n* columns.

If *transa* = 'T', A^{T} is used in the computation, and ***A*** has *n* rows and *m* columns.

Note: No data should be moved to form A^{T}; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 73, where:

If *transa* = 'N', its size must be *lda* by (at least) *n*.

If *transa* = 'T', its size must be *lda* by (at least) *m*.

lda

is the leading dimension of the array specified for *a*. Specified as: a fullword integer; *lda* > 0 and:

If *transa* = 'N', *lda* ≥ *m*.

If *transa* = 'T', *lda* ≥ *n*.

transa

indicates the form of matrix ***A*** to use in the computation, where:

If *transa* = 'N', ***A*** is used in the computation.

If *transa* = 'T', $\boldsymbol{A}^T$ is used in the computation.

Specified as: a single character; *transa* = 'N' or 'T'.

b

is the matrix ***B***, where:

If *transb* = 'N', ***B*** is used in the computation, and ***B*** has *m* rows and *n* columns.

If *transb* = 'T', $\boldsymbol{B}^T$ is used in the computation, and ***B*** has *n* rows and *m* columns.

Note: No data should be moved to form $\boldsymbol{B}^T$; that is, the matrix ***B*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 72 on page 384, where:

If *transb* = 'N', its size must be *ldb* by (at least) *n*.

If *transb* = 'T', its size must be *ldb* by (at least) *m*.

ldb

is the leading dimension of the array specified for *b*. Specified as: a fullword integer; $ldb > 0$ and:

If *transb* = 'N', $ldb \geq m$.

If *transb* = 'T', $ldb \geq n$.

transb

indicates the form of matrix ***B*** to use in the computation, where:

If *transb* = 'N', ***B*** is used in the computation.

If *transb* = 'T', $\boldsymbol{B}^T$ is used in the computation.

Specified as: a single character; *transb* = 'N' or 'T'.

c

See On Return.

ldc

is the leading dimension of the array specified for *c*. Specified as: a fullword integer; $ldc > 0$ and $ldc \geq m$.

m

is the number of rows in matrix ***C***. Specified as: a fullword integer; $0 \leq m \leq ldc$.

n

is the number of columns in matrix ***C***. Specified as: a fullword integer; $0 \leq n$.

On Return

c

is the *m* by *n* matrix ***C***, containing the results of the computation. Returned as: an *ldc* by (at least) *n* array, containing numbers of the data type indicated in Table 73 on page 391.

Notes

1. All subroutines accept lowercase letters for the *transa* and *transb* arguments.
2. Matrix ***C*** must have no common elements with matrices ***A*** or ***B***. However, ***C*** may (exactly) coincide with ***A*** if *transa* = 'N', and ***C*** may (exactly) coincide with ***B*** if *transb* = 'N'. Otherwise, results are unpredictable. See "Concepts" on page 55.

Function: The matrix subtraction is expressed as follows, where a_{ij}, b_{ij}, and c_{ij} are elements of matrices ***A, B,*** and ***C***, respectively:

$$
\begin{array}{ll}
c_{ij} = a_{ij} - b_{ij} & \text{for } \boldsymbol{C} \leftarrow \boldsymbol{A} - \boldsymbol{B} \\
c_{ij} = a_{ij} - b_{ji} & \text{for } \boldsymbol{C} \leftarrow \boldsymbol{A} - \boldsymbol{B}^{\mathrm{T}} \\
c_{ij} = a_{ji} - b_{ij} & \text{for } \boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{T}} - \boldsymbol{B} \\
c_{ij} = a_{ji} - b_{ji} & \text{for } \boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{T}} - \boldsymbol{B}^{\mathrm{T}} \\
\multicolumn{2}{l}{\text{for } i = 1, m \text{ and } j = 1, n}
\end{array}
$$

If m or n is 0, no computation is performed.

Special Usage: You can compute the transpose $\boldsymbol{C}^{\mathrm{T}}$ of each of the four computations listed under "Function" by using the following matrix identities:

$$
\begin{array}{l}
(\boldsymbol{A} - \boldsymbol{B})^{\mathrm{T}} = \boldsymbol{A}^{\mathrm{T}} - \boldsymbol{B}^{\mathrm{T}} \\
(\boldsymbol{A} - \boldsymbol{B}^{\mathrm{T}})^{\mathrm{T}} = \boldsymbol{A}^{\mathrm{T}} - \boldsymbol{B} \\
(\boldsymbol{A}^{\mathrm{T}} - \boldsymbol{B})^{\mathrm{T}} = \boldsymbol{A} - \boldsymbol{B}^{\mathrm{T}} \\
(\boldsymbol{A}^{\mathrm{T}} - \boldsymbol{B}^{\mathrm{T}})^{\mathrm{T}} = \boldsymbol{A} - \boldsymbol{B}
\end{array}
$$

Be careful that your output array receiving $\boldsymbol{C}^{\mathrm{T}}$ has dimensions large enough to hold the transposed matrix. See "Example 5" on page 395.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. *lda*, *ldb*, *ldc* ≤ 0
2. *m*, *n* < 0
3. *m* > *ldc*
4. *transa*, *transb* ≠ 'N' or 'T'
5. *transa* = 'N' and *m* > *lda*
6. *transa* = 'T' and *n* > *lda*
7. *transb* = 'N' and *m* > *ldb*
8. *transb* = 'T' and *n* > *ldb*

Example 1: This example shows the computation $\boldsymbol{C} \leftarrow \boldsymbol{A} - \boldsymbol{B}$, where ***A*** and ***C*** are contained in larger arrays A and C, respectively, and ***B*** is the same size as array B, in which it is contained.

Call Statement and Input

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  M   N
             |   |    |     |   |    |     |   |   |   |
CALL SGESUB( A , 6 , 'N'  , B , 4 , 'N'  , C , 5 , 4 , 3 )

        ⌈                                  ⌉
        | 110000.0  120000.0  130000.0 |
        | 210000.0  220000.0  230000.0 |
A    =  | 310000.0  320000.0  330000.0 |
        | 410000.0  420000.0  430000.0 |
        |       .         .         .  |
        |       .         .         .  |
        ⌊                                  ⌋
```

```
        ⎡                    ⎤
        ⎢ -11.0  -12.0  -13.0 ⎥
B    =  ⎢ -21.0  -22.0  -23.0 ⎥
        ⎢ -31.0  -32.0  -33.0 ⎥
        ⎢ -41.0  -42.0  -43.0 ⎥
        ⎣                    ⎦
```

Output

```
        ⎡                              ⎤
        ⎢ 110011.0  120012.0  130013.0 ⎥
        ⎢ 210021.0  220022.0  230023.0 ⎥
C    =  ⎢ 310031.0  320032.0  330033.0 ⎥
        ⎢ 410041.0  420042.0  430043.0 ⎥
        ⎢       .         .         . ⎥
        ⎣                              ⎦
```

Example 2: This example shows the computation $C \leftarrow A^T - B$, where ***A, B,*** and ***C*** are the same size as arrays A, B, and C, in which they are contained.

Call Statement and Input

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  M   N
             |   |    |     |   |    |     |   |   |   |
CALL SGESUB( A , 3 , 'T'  , B , 4 , 'N'  , C , 4 , 4 , 3 )
```

```
        ⎡                                      ⎤
        ⎢ 110000.0  120000.0  130000.0  140000.0 ⎥
A    =  ⎢ 210000.0  220000.0  230000.0  240000.0 ⎥
        ⎢ 310000.0  320000.0  330000.0  340000.0 ⎥
        ⎣                                      ⎦
```

```
        ⎡                    ⎤
        ⎢ -11.0  -12.0  -13.0 ⎥
B    =  ⎢ -21.0  -22.0  -23.0 ⎥
        ⎢ -31.0  -32.0  -33.0 ⎥
        ⎢ -41.0  -42.0  -43.0 ⎥
        ⎣                    ⎦
```

Output

```
        ⎡                              ⎤
        ⎢ 110011.0  210012.0  310013.0 ⎥
C    =  ⎢ 120021.0  220022.0  320023.0 ⎥
        ⎢ 130031.0  230032.0  330033.0 ⎥
        ⎢ 140041.0  240042.0  340043.0 ⎥
        ⎣                              ⎦
```

Example 3: This example shows computation $C \leftarrow A - B^T$, where ***A*** is contained in a larger array A, and ***B*** and ***C*** are the same size as arrays B and C, in which they are contained.

Call Statement and Input

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  M   N
             |   |    |     |   |    |     |   |   |   |
CALL SGESUB( A , 5 , 'N'  , B , 3 , 'T'  , C , 4 , 4 , 3 )
```

```
        ⎡                                  ⎤
        ⎢ 110000.0  120000.0  130000.0 ⎥
        ⎢ 210000.0  220000.0  230000.0 ⎥
A    =  ⎢ 310000.0  320000.0  330000.0 ⎥
        ⎢ 410000.0  420000.0  430000.0 ⎥
        ⎢       .         .         . ⎥
        ⎣                                  ⎦

        ⎡                              ⎤
        ⎢ -11.0  -12.0  -13.0  -14.0 ⎥
B    =  ⎢ -21.0  -22.0  -23.0  -24.0 ⎥
        ⎢ -31.0  -32.0  -33.0  -34.0 ⎥
        ⎣                              ⎦
```

Output

```
        ⎡                                  ⎤
        ⎢ 110011.0  120021.0  130031.0 ⎥
C    =  ⎢ 210012.0  220022.0  230032.0 ⎥
        ⎢ 310013.0  320023.0  330033.0 ⎥
        ⎢ 410014.0  420024.0  430034.0 ⎥
        ⎣                                  ⎦
```

Example 4: This example shows the computation $C \leftarrow A^T - B^T$, where ***A, B,*** and ***C*** are the same size as the arrays `A`, `B`, and `C`, in which they are contained.

Call Statement and Input

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  M   N
             |   |    |     |   |    |     |   |   |   |
CALL SGESUB( A , 3 , 'T'  , B , 3 , 'T'  , C , 4 , 4 , 3 )

        ⎡                                          ⎤
        ⎢ 110000.0  120000.0  130000.0  140000.0 ⎥
A    =  ⎢ 210000.0  220000.0  230000.0  240000.0 ⎥
        ⎢ 310000.0  320000.0  330000.0  340000.0 ⎥
        ⎣                                          ⎦

        ⎡                              ⎤
        ⎢ -11.0  -12.0  -13.0  -14.0 ⎥
B    =  ⎢ -21.0  -22.0  -23.0  -24.0 ⎥
        ⎢ -31.0  -32.0  -33.0  -34.0 ⎥
        ⎣                              ⎦
```

Output

```
        ⎡                                  ⎤
        ⎢ 110011.0  210021.0  310031.0 ⎥
C    =  ⎢ 120012.0  220022.0  320032.0 ⎥
        ⎢ 130013.0  230023.0  330033.0 ⎥
        ⎢ 140014.0  240024.0  340034.0 ⎥
        ⎣                                  ⎦
```

Example 5: This example shows how to produce the transpose of the result of the computation performed in "Example 4," $C \leftarrow A^T - B^T$, which uses the calling sequence:

```
CALL SGESUB( A , 3 , 'T' , B , 3 , 'T' , C , 4 , 4 , 3 )
```

You instead code a calling sequence for $C^T \leftarrow A - B$, as shown below, where the resulting matrix C^T in the output array `CT` is the transpose of the matrix in the output

array C in Example 4. Note that the array CT has dimensions large enough to receive the transposed matrix. For a description of all the matrix identities, see "Special Usage" on page 393.

Call Statement and Input

```
            A  LDA TRANSA  B  LDB TRANSB  C   LDC  M   N
            |   |    |     |   |    |     |    |   |   |
CALL SGESUB( A , 3 , 'N'  , B , 3 , 'N'  , CT , 3 , 3 , 4 )

        ⎡                                          ⎤
        | 110000.0  120000.0  130000.0  140000.0 |
A    =  | 210000.0  220000.0  230000.0  240000.0 |
        | 310000.0  320000.0  330000.0  340000.0 |
        ⎣                                          ⎦

        ⎡                                ⎤
        | -11.0  -12.0  -13.0  -14.0 |
B    =  | -21.0  -22.0  -23.0  -24.0 |
        | -31.0  -32.0  -33.0  -34.0 |
        ⎣                                ⎦
```

Output

```
        ⎡                                          ⎤
        | 110011.0  120012.0  130013.0  140014.0 |
CT   =  | 210021.0  220022.0  230023.0  240024.0 |
        | 310031.0  320032.0  330033.0  340034.0 |
        ⎣                                          ⎦
```

Example 6: This example shows the computation $\boldsymbol{C} \leftarrow \boldsymbol{A}-\boldsymbol{B}$, where $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ are contained in larger arrays A, B, and C, respectively, and the arrays contain complex data.

Call Statement and Input

```
            A  LDA TRANSA  B  LDB TRANSB  C  LDC  M   N
            |   |    |     |   |    |     |   |   |   |
CALL CGESUB( A , 6 , 'N'  , B , 5 , 'N'  , C , 5 , 4 , 3 )

        ⎡                                    ⎤
        | (1.0, 5.0)  (9.0, 2.0)  (1.0, 9.0) |
        | (2.0, 4.0)  (8.0, 3.0)  (1.0, 8.0) |
A    =  | (3.0, 3.0)  (7.0, 5.0)  (1.0, 7.0) |
        | (6.0, 6.0)  (3.0, 6.0)  (1.0, 4.0) |
        |     .           .           .      |
        |     .           .           .      |
        ⎣                                    ⎦

        ⎡                                    ⎤
        | (1.0, 8.0)  (2.0, 7.0)  (3.0, 2.0) |
        | (4.0, 4.0)  (6.0, 8.0)  (6.0, 3.0) |
B    =  | (6.0, 2.0)  (4.0, 5.0)  (4.0, 5.0) |
        | (7.0, 2.0)  (6.0, 4.0)  (1.0, 6.0) |
        |     .           .           .      |
        ⎣                                    ⎦
```

Output

```
        ⎡                                                  ⎤
        ⎢  (0.0, -3.0)   (7.0, -5.0)  (-2.0,  7.0)  ⎥
        ⎢ (-2.0,  0.0)   (2.0, -5.0)  (-5.0,  5.0)  ⎥
C    =  ⎢ (-3.0,  1.0)   (3.0,  0.0)  (-3.0,  2.0)  ⎥
        ⎢ (-1.0,  4.0)  (-3.0,  2.0)   (0.0, -2.0)  ⎥
        ⎢      .             .             .        ⎥
        ⎣                                                  ⎦
```

SGEMUL, DGEMUL, CGEMUL, and ZGEMUL—Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes

SGEMUL and DGEMUL can perform any one of the following matrix multiplications, using matrices ***A*** and ***B*** or their transposes, and matrix ***C***:

$$C \leftarrow AB \qquad C \leftarrow AB^{T}$$
$$C \leftarrow A^{T}B \qquad C \leftarrow A^{T}B^{T}$$

CGEMUL and ZGEMUL can perform any one of the following matrix multiplications, using matrices ***A*** and ***B***, their transposes or their conjugate transposes, and matrix ***C***:

$$C \leftarrow AB \qquad C \leftarrow AB^{T} \qquad C \leftarrow AB^{H}$$
$$C \leftarrow A^{T}B \qquad C \leftarrow A^{T}B^{T} \qquad C \leftarrow A^{T}B^{H}$$
$$C \leftarrow A^{H}B \qquad C \leftarrow A^{H}B^{T} \qquad C \leftarrow A^{H}B^{H}$$

Table 74. Data Types

A, ***B***, ***C***	**Subroutine**
Short-precision real	SGEMUL
Long-precision real	DGEMUL
Short-precision complex	CGEMUL
Long-precision complex	ZGEMUL

Syntax

Fortran	CALL SGEMUL | DGEMUL | CGEMUL | ZGEMUL (*a*, *lda*, *transa*, *b*, *ldb*, *transb*, *c*, *ldc*, *l*, *m*, *n*)
C and C++	sgemul | dgemul | cgemul | zgemul (*a*, *lda*, *transa*, *b*, *ldb*, *transb*, *c*, *ldc*, *l*, *m*, *n*);
PL/I	CALL SGEMUL | DGEMUL | CGEMUL | ZGEMUL (*a*, *lda*, *transa*, *b*, *ldb*, *transb*, *c*, *ldc*, *l*, *m*, *n*);
APL2	SGEMUL | DGEMUL | CGEMUL | ZGEMUL *a lda transa b ldb transb* `'c'` *ldc l m n*

On Entry

a
is the matrix ***A***, where:

If *transa* = `'N'`, ***A*** is used in the computation, and ***A*** has *l* rows and *m* columns.

If *transa* = `'T'`, A^{T} is used in the computation, and ***A*** has *m* rows and *l* columns.

If *transa* = `'C'`, A^{H} is used in the computation, and ***A*** has *m* rows and *l* columns.

Note: No data should be moved to form A^{T} or A^{H}; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 74, where:

If *transa* = `'N'`, its size must be *lda* by (at least) *m*.

If *transa* = 'T' or 'C', its size must be *lda* by (at least) *l*.

lda

is the leading dimension of the array specified for *a*. Specified as: a fullword integer; *lda* > 0 and:

If *transa* = 'N', *lda* ≥ *l*.

If *transa* = 'T' or 'C', *lda* ≥ *m*.

transa

indicates the form of matrix ***A*** to use in the computation, where:

If *transa* = 'N', ***A*** is used in the computation.

If *transa* = 'T', $\boldsymbol{A}^{T}$ is used in the computation.

If *transa* = 'C', $\boldsymbol{A}^{H}$ is used in the computation.

Specified as: a single character; *transa* = 'N' or 'T' for SGEMUL and DGEMUL; *transa* = 'N', 'T', or 'C' for CGEMUL and ZGEMUL.

b

is the matrix ***B***, where:

If *transb* = 'N', ***B*** is used in the computation, and ***B*** has *m* rows and *n* columns.

If *transb* = 'T', $\boldsymbol{B}^{T}$ is used in the computation, and ***B*** has *n* rows and *m* columns.

If *transb* = 'C', $\boldsymbol{B}^{H}$ is used in the computation, and ***B*** has *n* rows and *m* columns.

Note: No data should be moved to form $\boldsymbol{B}^{T}$ or $\boldsymbol{B}^{H}$; that is, the matrix ***B*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 74 on page 398, where:

If *transb* = 'N', its size must be *ldb* by (at least) *n*.

If *transb* = 'T' or 'C', its size must be *ldb* by (at least) *m*.

ldb

is the leading dimension of the array specified for *b*. Specified as: a fullword integer; *ldb* > 0 and:

If *transb* = 'N', *ldb* ≥ *m*.

If *transb* = 'T' or 'C', *ldb* ≥ *n*.

transb

indicates the form of matrix ***B*** to use in the computation, where:

If *transb* = 'N', ***B*** is used in the computation.

If *transb* = 'T', $\boldsymbol{B}^{T}$ is used in the computation.

If *transb* = 'C', $\boldsymbol{B}^{H}$ is used in the computation.

Specified as: a single character; *transb* = 'N' or 'T' for SGEMUL and DGEMUL; *transb* = 'N', 'T', or 'C' for CGEMUL and ZGEMUL.

c

See On Return.

ldc

is the leading dimension of the array specified for *c*. Specified as: a fullword integer; *ldc* > 0 and *ldc* ≥ *l*.

l

is the number of rows in matrix ***C***. Specified as: a fullword integer; 0 ≤ *l* ≤ *ldc*.

m
has the following meaning, where:

If *transa* = 'N', it is the number of columns in matrix ***A***.

If *transa* = 'T' or 'C', it is the number of rows in matrix ***A***.

In addition:

If *transb* = 'N', it is the number of rows in matrix ***B***.

If *transb* = 'T' or 'C', it is the number of columns in matrix ***B***.

Specified as: a fullword integer; $m \geq 0$.

n
is the number of columns in matrix ***C***. Specified as: a fullword integer; $n \geq 0$.

On Return

c
is the *l* by *n* matrix ***C***, containing the results of the computation. Returned as: an *ldc* by (at least) *n* numbers of the data type indicated in Table 74 on page 398.

Notes

1. All subroutines accept lowercase letters for the *transa* and *transb* arguments.
2. Matrix ***C*** must have no common elements with matrices ***A*** or ***B***; otherwise, results are unpredictable. See "Concepts" on page 55.

Function: The matrix multiplication is expressed as follows, where a_{ik}, b_{kj}, and c_{ij} are elements of matrices ***A, B,*** and ***C***, respectively:

$$c_{ij} = \sum_{k=1}^{m} a_{ik}\, b_{kj} \quad \text{for } \boldsymbol{C} \leftarrow \boldsymbol{A}\ \boldsymbol{B}$$

$$c_{ij} = \sum_{k=1}^{m} a_{ki}\, b_{kj} \quad \text{for } \boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{T}}\ \boldsymbol{B}$$

$$c_{ij} = \sum_{k=1}^{m} \overline{a}_{ki}\, b_{kj} \quad \text{for } \boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{H}}\ \boldsymbol{B}$$

$$c_{ij} = \sum_{k=1}^{m} a_{ik}\, b_{jk} \quad \text{for } \boldsymbol{C} \leftarrow \boldsymbol{A}\ \boldsymbol{B}^{\mathrm{T}}$$

$$c_{ij} = \sum_{k=1}^{m} a_{ki}\, b_{jk} \quad \text{for } \boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{T}}\ \boldsymbol{B}^{\mathrm{T}}$$

$$c_{ij} = \sum_{k=1}^{m} \overline{a}_{ki}\, b_{jk} \quad \text{for } \boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{H}}\ \boldsymbol{B}^{\mathrm{T}}$$

$$c_{ij} = \sum_{k=1}^{m} a_{ik}\, \overline{b}_{jk} \quad \text{for } \boldsymbol{C} \leftarrow \boldsymbol{A}\ \boldsymbol{B}^{\mathrm{H}}$$

$$c_{ij} = \sum_{k=1}^{m} a_{ki}\, \overline{b}_{jk} \quad \text{for } \boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{T}}\ \boldsymbol{B}^{\mathrm{H}}$$

$$c_{ij} = \sum_{k=1}^{m} \overline{a}_{ki}\, \overline{b}_{jk} \quad \text{for } \boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{H}}\ \boldsymbol{B}^{\mathrm{H}}$$

$$\text{for } \ i = 1,\ l \ \text{ and } \ j = 1,n$$

See reference [38]. If *l* or *n* is 0, no computation is performed. If *l* and *n* are greater than 0, and *m* is 0, an *l* by *n* matrix of zeros is returned.

Special Usage

Equivalence Rules: By using the following equivalence rules, you can compute the transpose $\boldsymbol{C}^{\mathrm{T}}$ or the conjugate transpose $\boldsymbol{C}^{\mathrm{H}}$ of some of the computations performed by these subroutines:

Transpose	Conjugate Transpose
$(\boldsymbol{AB})^{\mathrm{T}} = \boldsymbol{B}^{\mathrm{T}}\boldsymbol{A}^{\mathrm{T}}$	$(\boldsymbol{AB})^{\mathrm{H}} = \boldsymbol{B}^{\mathrm{H}}\boldsymbol{A}^{\mathrm{H}}$
$(\boldsymbol{A}^{\mathrm{T}}\boldsymbol{B})^{\mathrm{T}} = \boldsymbol{B}^{\mathrm{T}}\boldsymbol{A}$	$(\boldsymbol{A}^{\mathrm{H}}\boldsymbol{B})^{\mathrm{H}} = \boldsymbol{B}^{\mathrm{H}}\boldsymbol{A}$
$(\boldsymbol{AB}^{\mathrm{T}})^{\mathrm{T}} = \boldsymbol{B}\boldsymbol{A}^{\mathrm{T}}$	$(\boldsymbol{AB}^{\mathrm{H}})^{\mathrm{H}} = \boldsymbol{B}\boldsymbol{A}^{\mathrm{H}}$
$(\boldsymbol{A}^{\mathrm{T}}\boldsymbol{B}^{\mathrm{T}})^{\mathrm{T}} = \boldsymbol{BA}$	$(\boldsymbol{A}^{\mathrm{H}}\boldsymbol{B}^{\mathrm{H}})^{\mathrm{H}} = \boldsymbol{BA}$

When coding the calling sequences for these cases, be careful to code your matrix arguments and dimension arguments in the order indicated by the rule. Also, be careful that your output array, receiving $\boldsymbol{C}^{\mathrm{T}}$ or $\boldsymbol{C}^{\mathrm{H}}$, has dimensions large enough to hold the resulting transposed or conjugate transposed matrix. See "Example 2" on page 403 and "Example 4" on page 404.

Error Conditions

Resource Errors: Unable to allocate internal work area (CGEMUL and ZGEMUL only).

Computational Errors: None

Input-Argument Errors

1. *lda*, *ldb*, *ldc* ≤ 0
2. *l*, *m*, *n* < 0
3. *l* > *ldc*
4. *transa*, *transb* ≠ 'N' or 'T' for SGEMUL and DGEMUL
5. *transa*, *transb* ≠ 'N', 'T', or 'C' for CGEMUL and ZGEMUL
6. *transa* = 'N' and *l* > *lda*
7. *transa* = 'T' or 'C' and *m* > *lda*
8. *transb* = 'N' and *m* > *ldb*
9. *transb* = 'T' or 'C' and *n* > *ldb*

Example 1: This example shows the computation ***C*** ← ***AB***, where ***A, B,*** and ***C*** are contained in larger arrays A, B, and C, respectively.

Call Statement and Input

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  L   M   N
             |   |    |     |   |    |     |   |   |   |   |
CALL SGEMUL( A , 8 , 'N'  , B , 6 , 'N'  , C , 7 , 6 , 5 , 4 )
```

```
        ⎡                                  ⎤
        ⎢  1.0   2.0  -1.0  -1.0   4.0 ⎥
        ⎢  2.0   0.0   1.0   1.0  -1.0 ⎥
        ⎢  1.0  -1.0  -1.0   1.0   2.0 ⎥
A    =  ⎢ -3.0   2.0   2.0   2.0   0.0 ⎥
        ⎢  4.0   0.0  -2.0   1.0  -1.0 ⎥
        ⎢ -1.0  -1.0   1.0  -3.0   2.0 ⎥
        ⎢   .     .     .     .     .  ⎥
        ⎣   .     .     .     .     .  ⎦
```

```
        ⎡                          ⎤
        ⎢  1.0  -1.0   0.0   2.0 ⎥
        ⎢  2.0   2.0  -1.0  -2.0 ⎥
B    =  ⎢  1.0   0.0  -1.0   1.0 ⎥
        ⎢ -3.0  -1.0   1.0  -1.0 ⎥
        ⎢  4.0   2.0  -1.0   1.0 ⎥
        ⎣   .     .     .     .  ⎦
```

Output

```
        ⎡                                ⎤
        ⎢ 23.0  12.0  -6.0    2.0 ⎥
        ⎢ -4.0  -5.0   1.0    3.0 ⎥
        ⎢  3.0   0.0   1.0    4.0 ⎥
C    =  ⎢ -3.0   5.0  -2.0  -10.0 ⎥
        ⎢ -5.0  -7.0   4.0    4.0 ⎥
        ⎢ 15.0   6.0  -5.0    6.0 ⎥
        ⎢   .     .     .      .  ⎥
        ⎣                                ⎦
```

Example 2: This example shows how to produce the transpose of the result of the computation performed in "Example 1" on page 402, $\boldsymbol{C} \leftarrow \boldsymbol{AB}$, which uses the calling sequence:

```
CALL  SGEMUL (A,8,'N',B,6,'N',C,7,6,5,4)
```

You instead code a calling sequence for $\boldsymbol{C}^{\mathrm{T}} \leftarrow \boldsymbol{B}^{\mathrm{T}}\boldsymbol{A}^{\mathrm{T}}$, as shown below, where the resulting matrix $\boldsymbol{C}^{\mathrm{T}}$ in the output array `CT` is the transpose of the matrix in the output array `C` in Example 1. Note that the array `CT` has dimensions large enough to receive the transposed matrix. For a description of all the matrix identities, see "Special Usage" on page 401.

Call Statement and Input

```
             A  LDA TRANSA  B  LDB TRANSB  C   LDC  L   M   N
             |   |    |     |   |    |     |    |   |   |   |
CALL SGEMUL( B , 6 , 'T'  , A , 8 , 'T'  , CT , 5 , 4 , 5 , 6 )
```

```
        ⎡                                ⎤
        ⎢  1.0  -1.0   0.0   2.0 ⎥
        ⎢  2.0   2.0  -1.0  -2.0 ⎥
B    =  ⎢  1.0   0.0  -1.0   1.0 ⎥
        ⎢ -3.0  -1.0   1.0  -1.0 ⎥
        ⎢  4.0   2.0  -1.0   1.0 ⎥
        ⎢   .     .     .     .  ⎥
        ⎣                                ⎦
```

```
        ⎡                                    ⎤
        ⎢  1.0   2.0  -1.0  -1.0   4.0 ⎥
        ⎢  2.0   0.0   1.0   1.0  -1.0 ⎥
        ⎢  1.0  -1.0  -1.0   1.0   2.0 ⎥
A    =  ⎢ -3.0   2.0   2.0   2.0   0.0 ⎥
        ⎢  4.0   0.0  -2.0   1.0  -1.0 ⎥
        ⎢ -1.0  -1.0   1.0  -3.0   2.0 ⎥
        ⎢   .     .     .     .     .  ⎥
        ⎢   .     .     .     .     .  ⎥
        ⎣                                    ⎦
```

Output

```
        ⎡                                                ⎤
        ⎢ 23.0  -4.0  3.0   -3.0  -5.0  15.0 ⎥
        ⎢ 12.0  -5.0  0.0    5.0  -7.0   6.0 ⎥
CT   =  ⎢ -6.0   1.0  1.0   -2.0   4.0  -5.0 ⎥
        ⎢  2.0   3.0  4.0  -10.0   4.0   6.0 ⎥
        ⎢   .     .    .      .     .     .  ⎥
        ⎣                                                ⎦
```

Example 3: This example shows the computation $\boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{T}}\boldsymbol{B}$, where $\boldsymbol{A}$ and $\boldsymbol{C}$ are contained in larger arrays `A` and `C`, respectively, and $\boldsymbol{B}$ is the same size as the

Call Statement and Input

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  L   M   N
             |   |    |     |   |    |     |   |   |   |   |
CALL SGEMUL( A , 4 , 'T'  , B , 3 , 'N'  , C , 5 , 3 , 3 , 6 )
```

```
        ⎡                    ⎤
        ⎢ 1.0  -3.0   2.0 ⎥
A    =  ⎢ 2.0   4.0   0.0 ⎥
        ⎢ 1.0  -1.0  -1.0 ⎥
        ⎢  .     .     .  ⎥
        ⎣                    ⎦
```

```
        ⎡                                        ⎤
        ⎢ 1.0  -3.0   2.0   2.0  -1.0   2.0 ⎥
B    =  ⎢ 2.0   4.0   0.0   0.0   1.0  -2.0 ⎥
        ⎢ 1.0  -1.0  -1.0  -1.0  -1.0   1.0 ⎥
        ⎣                                        ⎦
```

Output

```
        ⎡                                        ⎤
        ⎢ 6.0   4.0   1.0   1.0   0.0   -1.0 ⎥
        ⎢ 4.0  26.0  -5.0  -5.0   8.0  -15.0 ⎥
C    =  ⎢ 1.0  -5.0   5.0   5.0  -1.0    3.0 ⎥
        ⎢  .     .     .     .     .      .  ⎥
        ⎢  .     .     .     .     .      .  ⎥
        ⎣                                        ⎦
```

Example 4: This example shows how to produce the transpose of the result of the computation performed in "Example 3," $\boldsymbol{C} \leftarrow \boldsymbol{A}^{\mathrm{T}}\boldsymbol{B}$, which uses the calling sequence:

```
CALL  SGEMUL (A,4,'T',B,3,'N',C,5,3,3,6)
```

You instead code the calling sequence for $\boldsymbol{C}^{\mathrm{T}} \leftarrow \boldsymbol{B}^{\mathrm{T}}\boldsymbol{A}$, as shown below, where the resulting matrix $\boldsymbol{C}^{\mathrm{T}}$ in the output array `CT` is the transpose of the matrix in the output array `C` in Example 3. Note that the array `CT` has dimensions large enough to receive the transposed matrix. For a description of all the matrix identities, see "Special Usage" on page 401.

Call Statement and Input

```
             A  LDA TRANSA  B  LDB TRANSB  C   LDC  L   M   N
             |   |    |     |   |    |     |    |   |   |   |
CALL SGEMUL( B , 3 , 'T'  , A , 4 , 'N'  , CT , 8 , 6 , 3 , 3 )
```

```
        ⎡                                          ⎤
        ⎢ 1.0  -3.0   2.0   2.0  -1.0   2.0 ⎥
B    =  ⎢ 2.0   4.0   0.0   0.0   1.0  -2.0 ⎥
        ⎢ 1.0  -1.0  -1.0  -1.0  -1.0   1.0 ⎥
        ⎣                                          ⎦

        ⎡                    ⎤
        ⎢ 1.0  -3.0   2.0 ⎥
A    =  ⎢ 2.0   4.0   0.0 ⎥
        ⎢ 1.0  -1.0  -1.0 ⎥
        ⎢  .     .     .  ⎥
        ⎣                    ⎦
```

Output

```
        ⎡                      ⎤
        ⎢  6.0    4.0   1.0 ⎥
        ⎢  4.0   26.0  -5.0 ⎥
        ⎢  1.0   -5.0   5.0 ⎥
CT   =  ⎢  1.0   -5.0   5.0 ⎥
        ⎢  0.0    8.0  -1.0 ⎥
        ⎢ -1.0  -15.0   3.0 ⎥
        ⎢   .      .     .  ⎥
        ⎢   .      .     .  ⎥
        ⎣                      ⎦
```

Example 5: This example shows the computation $\boldsymbol{C} \leftarrow \boldsymbol{AB}^{\mathrm{T}}$, where $\boldsymbol{A}$ and $\boldsymbol{C}$ are contained in larger arrays `A` and `C`, respectively, and $\boldsymbol{B}$ is the same size as the array `B` in which it is contained.

Call Statement and Input

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  L   M   N
             |   |    |     |   |    |     |   |   |   |   |
CALL SGEMUL( A , 4 , 'N'  , B , 3 , 'T'  , C , 5 , 3 , 2 , 3 )

        ⎡            ⎤
        ⎢ 1.0  -3.0 ⎥
A    =  ⎢ 2.0   4.0 ⎥
        ⎢ 1.0  -1.0 ⎥
        ⎢  .     .  ⎥
        ⎣            ⎦

        ⎡            ⎤
        ⎢ 1.0  -3.0 ⎥
B    =  ⎢ 2.0   4.0 ⎥
        ⎢ 1.0  -1.0 ⎥
        ⎣            ⎦
```

Output

```
        ⎡                      ⎤
        ⎢  10.0  -10.0   4.0 ⎥
        ⎢ -10.0   20.0  -2.0 ⎥
C    =  ⎢   4.0   -2.0   2.0 ⎥
        ⎢    .      .     .  ⎥
        ⎢    .      .     .  ⎥
        ⎣                      ⎦
```

Example 6: This example shows the computation $C \leftarrow A^{T}B^{T}$, where ***A, B,*** and ***C*** are the same size as the arrays A, B, and C in which they are contained. (Based on the dimensions of the matrices, ***A*** is actually a column vector, and ***C*** is actually a row vector.)

Call Statement and Input

```
            A  LDA TRANSA  B  LDB TRANSB  C  LDC  L   M   N
            |   |    |     |   |    |     |   |   |   |   |
CALL SGEMUL( A , 3 , 'T'  , B , 3 , 'T'  , C , 1 , 1 , 3 , 3 )

        ┌     ┐
        │ 1.0 │
A    =  │ 2.0 │
        │ 1.0 │
        └     ┘

        ┌                ┐
        │ 1.0  -3.0   2.0 │
B    =  │ 2.0   4.0   0.0 │
        │ 1.0  -1.0  -1.0 │
        └                ┘
```

Output

```
        ┌                ┐
B    =  │ -3.0  10.0  -2.0 │
        └                ┘
```

Example 7: This example shows the computation $C \leftarrow A^{T}B$ using complex data, where ***A***, ***B***, and ***C*** are contained in larger arrays A, B, and C, respectively.

Call Statement and Input

```
            A  LDA TRANSA  B  LDB TRANSB  C  LDC  L   M   N
            |   |    |     |   |    |     |   |   |   |   |
CALL CGEMUL( A , 6 , 'T'  , B , 7 , 'N'  , C , 3 , 2 , 3 , 3 )

        ┌                      ┐
        │ (1.0, 2.0)  (3.0, 4.0) │
        │ (4.0, 6.0)  (7.0, 1.0) │
A    =  │ (6.0, 3.0)  (2.0, 5.0) │
        │     .           .      │
        │     .           .      │
        │     .           .      │
        └                      ┘

        ┌                                ┐
        │ (1.0, 9.0)  (2.0, 6.0)  (5.0, 6.0) │
        │ (2.0, 5.0)  (6.0, 2.0)  (6.0, 4.0) │
        │ (2.0, 6.0)  (5.0, 4.0)  (2.0, 6.0) │
B    =  │     .           .           .      │
        │     .           .           .      │
        │     .           .           .      │
        │     .           .           .      │
        └                                ┘
```

Output

```
        ⎡                                          ⎤
        ⎢ (-45.0, 85.0)  (20.0, 93.0)  (-13.0, 110.0) ⎥
C    =  ⎢ (-50.0, 90.0)  (12.0, 79.0)    (3.0,  94.0) ⎥
        ⎢       .            .             .        ⎥
        ⎣                                          ⎦
```

Example 8: This example shows the computation $\boldsymbol{C} \leftarrow \boldsymbol{AB}^{H}$ using complex data, where ***A*** and ***C*** are contained in larger arrays A and C, respectively, and ***B*** is the same size as the array B in which it is contained.

Call Statement and Input

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  L   M   N
             |   |    |     |   |    |     |   |   |   |   |
CALL CGEMUL( A , 4 , 'N'  , B , 3 , 'C'  , C , 4 , 3 , 2 , 3 )

        ⎡                    ⎤
        ⎢ (1.0, 2.0)  (-3.0, 2.0) ⎥
A    =  ⎢ (2.0, 6.0)   (4.0, 5.0) ⎥
        ⎢ (1.0, 2.0)  (-1.0, 8.0) ⎥
        ⎢     .           .      ⎥
        ⎣                    ⎦

        ⎡                    ⎤
        ⎢ (1.0, 3.0)  (-3.0, 2.0) ⎥
B    =  ⎢ (2.0, 5.0)   (4.0, 6.0) ⎥
        ⎢ (1.0, 1.0)  (-1.0, 9.0) ⎥
        ⎣                    ⎦
```

Output

```
        ⎡                                          ⎤
        ⎢ (20.0,  -1.0)  (12.0, 25.0)  (24.0,  26.0) ⎥
C    =  ⎢ (18.0, -23.0)  (80.0, -2.0)  (49.0, -37.0) ⎥
        ⎢ (26.0, -23.0)  (56.0, 37.0)  (76.0,   2.0) ⎥
        ⎢      .            .             .        ⎥
        ⎣                                          ⎦
```

SGEMMS, DGEMMS, CGEMMS, and ZGEMMS—Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes Using Winograd's Variation of Strassen's Algorithm

These subroutines use Winograd's variation of the Strassen's algorithm to perform the matrix multiplication for both real and complex matrices. SGEMMS and DGEMMS can perform any one of the following matrix multiplications, using matrices ***A*** and ***B*** or their transposes, and matrix ***C***:

$$C \leftarrow AB \qquad C \leftarrow AB^{T}$$
$$C \leftarrow A^{T}B \qquad C \leftarrow A^{T}B^{T}$$

CGEMMS and ZGEMMS can perform any one of the following matrix multiplications, using matrices ***A*** and ***B***, their transposes or their conjugate transposes, and matrix ***C***:

$$C \leftarrow AB \qquad C \leftarrow AB^{T} \qquad C \leftarrow AB^{H}$$
$$C \leftarrow A^{T}B \qquad C \leftarrow A^{T}B^{T} \qquad C \leftarrow A^{T}B^{H}$$
$$C \leftarrow A^{H}B \qquad C \leftarrow A^{H}B^{T} \qquad C \leftarrow A^{H}B^{H}$$

Table 75. Data Types

A, ***B***, ***C***	*aux*	**Subroutine**
Short-precision real	Short-precision real	SGEMMS
Long-precision real	Long-precision real	DGEMMS
Short-precision complex	Short-precision real	CGEMMS
Long-precision complex	Long-precision real	ZGEMMS

Syntax

Fortran	CALL SGEMMS | DGEMMS | CGEMMS | ZGEMMS (*a*, *lda*, *transa*, *b*, *ldb*, *transb*, *c*, *ldc*, *l*, *m*, *n*, *aux*, *naux*)
C and C++	sgemms | dgemms | cgemms | zgemms (*a*, *lda*, *transa*, *b*, *ldb*, *transb*, *c*, *ldc*, *l*, *m*, *n*, *aux*, *naux*);
PL/I	CALL SGEMMS | DGEMMS | CGEMMS | ZGEMMS (*a*, *lda*, *transa*, *b*, *ldb*, *transb*, *c*, *ldc*, *l*, *m*, *n*, *aux*, *naux*);

On Entry

a

is the matrix ***A***, where:

If *transa* = 'N', ***A*** is used in the computation, and ***A*** has *l* rows and *m* columns.

If *transa* = 'T', A^{T} is used in the computation, and ***A*** has *m* rows and *l* columns.

If *transa* = 'C', A^{H} is used in the computation, and ***A*** has *m* rows and *l* columns.

Note: No data should be moved to form A^{T} or A^{H}; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 75, where:

If *transa* = 'N', its size must be *lda* by (at least) *m*.

If *transa* = 'T' or 'C', its size must be *lda* by (at least) *l*.

lda

is the leading dimension of the array specified for *a*. Specified as: a fullword integer; *lda* > 0 and:

If *transa* = 'N', *lda* ≥ *l*.

If *transa* = 'T' or 'C', *lda* ≥ *m*.

transa

indicates the form of matrix ***A*** to use in the computation, where:

If *transa* = 'N', ***A*** is used in the computation.

If *transa* = 'T', $\boldsymbol{A}^{T}$ is used in the computation.

If *transa* = 'C', $\boldsymbol{A}^{H}$ is used in the computation.

Specified as: a single character; *transa* = 'N' or 'T' for SGEMMS and DGEMMS; *transa* = 'N', 'T', or 'C' for CGEMMS and ZGEMMS.

b

is the matrix ***B***, where:

If *transb* = 'N', ***B*** is used in the computation, and ***B*** has *m* rows and *n* columns.

If *transb* = 'T', $\boldsymbol{B}^{T}$ is used in the computation, and ***B*** has *n* rows and *m* columns.

If *transb* = 'C', $\boldsymbol{B}^{H}$ is used in the computation, and ***B*** has *n* rows and *m* columns.

Note: No data should be moved to form $\boldsymbol{B}^{T}$ or $\boldsymbol{B}^{H}$; that is, the matrix ***B*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 75 on page 408, where:

If *transb* = 'N', its size must be *ldb* by (at least) *n*.

If *transb* = 'T' or 'C', its size must be *ldb* by (at least) *m*.

ldb

is the leading dimension of the array specified for *b*. Specified as: a fullword integer; *ldb* > 0 and:

If *transb* = 'N', *ldb* ≥ *m*.

If *transb* = 'T' or 'C', *ldb* ≥ *n*.

transb

indicates the form of matrix ***B*** to use in the computation, where:

If *transb* = 'N', ***B*** is used in the computation.

If *transb* = 'T', $\boldsymbol{B}^{T}$ is used in the computation.

If *transb* = 'C', $\boldsymbol{B}^{H}$ is used in the computation.

Specified as: a single character; *transb* = 'N' or 'T' for SGEMMS and DGEMMS; *transb* = 'N', 'T', or 'C' for CGEMMS and ZGEMMS.

c

See On Return.

ldc

is the leading dimension of the array specified for *c*. Specified as: a fullword integer; *ldc* > 0 and *ldc* ≥ *l*.

l

is the number of rows in matrix ***C***. Specified as: a fullword integer; 0 ≤ *l* ≤ *ldc*.

m
has the following meaning, where:

If *transa* = 'N', it is the number of columns in matrix ***A***.

If *transa* = 'T' or 'C', it is the number of rows in matrix ***A***.

In addition:

If *transb* = 'N', it is the number of rows in matrix ***B***.

If *transb* = 'T' or 'C', it is the number of columns in matrix ***B***.

Specified as: a fullword integer; $m \geq 0$.

n
is the number of columns in matrix ***C***. Specified as: a fullword integer; $n \geq 0$.

aux
has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, is the storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage containing numbers of the data type indicated in Table 75 on page 408.

naux
is the size of the work area specified by *aux*—that is, the number of elements in *aux*.

Specified as: a fullword integer, where:

If *naux* = 0 and error 2015 is unrecoverable, SGEMMS, DGEMMS, CGEMMS, and ZGEMMS dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise,

When this subroutine uses Strassen's algorithm:

- For SGEMMS and DGEMMS:

  Use $naux = \max[(n)(l), 0.7m(l+n)]$.

- For CGEMMS and ZGEMMS:

  Use $naux = \max[(n)(l), 0.7m(l+n)]+nb1+nb2$, where:
  If $l \geq n$, then $nb1 \geq (l)(n+20)$ and $nb2 \geq \max[(n)(l), (m)(n+20)]$.
  If $l < n$, then $nb1 \geq (m)(n+20)$ and $nb2 \geq \max[(n)(l), (l)(m+20)]$.

When this subroutine uses the direct method (_GEMUL), use $naux \geq 0$.

Notes:

1. In most cases, these formulas provide an overestimate.
2. For an explanation of when this subroutine uses the direct method versus Strassen's algorithm, see "Notes" on page 411.

On Return

c
is the *l* by *n* matrix ***C***, containing the results of the computation. Returned as: an *ldc* by (at least) *n* array, containing numbers of the data type indicated in Table 75 on page 408.

Notes

1. There are two instances when these subroutines use the direct method (_GEMUL), rather than using Strassen's algorithm:
   - When either or both of the input matrices are small
   - For CGEMMS and ZGEMMS, when input matrices ***A*** and ***B*** overlap

   In these instances when the direct method is used, the subroutine does not use auxiliary storage, and you can specify *naux* = 0.
2. For CGEMMS and ZGEMMS, one of the input matrices, ***A*** or ***B***, is rearranged during the computation and restored to its original form on return. Keep this in mind when diagnosing an abnormal termination.
3. All subroutines accept lowercase letters for the *transa* and *transb* arguments.
4. Matrix ***C*** must have no common elements with matrices ***A*** or ***B***; otherwise, results are unpredictable. See “Concepts” on page 55.
5. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see “Using Auxiliary Storage in ESSL” on page 31.

Function: The matrix multiplications performed by these subroutines are functionally equivalent to those performed by SGEMUL, DGEMUL, CGEMUL, and ZGEMUL. For details on the computations performed, see “Function” on page 400.

SGEMMS, DGEMMS, CGEMMS, and ZGEMMS use Winograd's variation of the Strassen's algorithm with minor changes for tuning purposes. (See pages 45 and 46 in reference [11].) The subroutines compute matrix multiplication for both real and complex matrices of large sizes. Complex matrix multiplication uses a special technique, using three real matrix multiplications and five real matrix additions. Each of these three resulting matrix multiplications then uses Strassen's algorithm.

Strassen's Algorithm: The steps of Strassen's algorithm can be repeated up to four times by these subroutines, with each step reducing the dimensions of the matrix by a factor of two. The number of steps used by this subroutine depends on the size of the input matrices. Each step reduces the number of operations by about 10% from the normal matrix multiplication. On the other hand, if the matrix is small, a normal matrix multiplication is performed without using the Strassen's algorithm, and no improvement is gained. For details about small matrices, see “Notes.”

Complex Matrix Multiplication: The complex multiplication is performed by forming the real and imaginary parts of the input matrices. These subroutines uses three real matrix multiplications and five real matrix additions, instead of the normal four real matrix multiplications and two real matrix additions. Using only three real matrix multiplications allows the subroutine to achieve up to a 25% reduction in matrix operations, which can result in a significant savings in computing time for large matrices.

Accuracy Considerations: Strassen's method is not stable for certain row or column scalings of the input matrices ***A*** and ***B***. Therefore, for matrices ***A*** and ***B*** with divergent exponent values Strassen's method may give inaccurate results. For these cases, you should use the _GEMUL or _GEMM subroutines.

Special Usage: The equivalence rules, defined for matrix multiplication of ***A*** and ***B*** in "Special Usage" on page 401, also apply to these subroutines. You should use the equivalence rules when you want to transpose or conjugate transpose the result of the multiplication computation. When coding the calling sequences for these cases, be careful to code your matrix arguments and dimension arguments in the order indicated by the rule. Also, be careful that your output array, receiving $\boldsymbol{C}^{T}$ or $\boldsymbol{C}^{H}$, has dimensions large enough to hold the resulting transposed or conjugate transposed matrix. See "Example 2" on page 403 and "Example 4" on page 404.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors: None

Input-Argument Errors

1. *lda*, *ldb*, *ldc* ≤ 0
2. *l*, *m*, *n* < 0
3. *l* > *ldc*
4. *transa*, *transb* ≠ 'N' or 'T' for SGEMMS and DGEMMS
5. *transa*, *transb* ≠ 'N', 'T', or 'C' for CGEMMS and ZGEMMS
6. *transa* = 'N' and *l* > *lda*
7. *transa* = 'T' or 'C' and *m* > *lda*
8. *transb* = 'N' and *m* > *ldb*
9. *transb* = 'T' or 'C' and *n* > *ldb*
10. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example 1: This example shows the computation $\boldsymbol{C} \leftarrow \boldsymbol{AB}$, where ***A***, ***B***, and ***C*** are contained in larger arrays A, B, and C, respectively. It shows how to code the calling sequence for SGEMMS, but does not use the Strassen algorithm for doing the computation. The calling sequence is shown below. The input and output, other than auxiliary storage, is the same as in "Example 1" on page 402 for SGEMUL.

Call Statement and Input

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  L   M   N   AUX  NAUX
             |   |    |     |   |    |     |   |   |   |   |    |    |
CALL SGEMMS( A , 8 , 'N'  , B , 6 , 'N'  , C , 7 , 6 , 5 , 4 , AUX , 0  )
```

Example 2: This example shows the computation $\boldsymbol{C} \leftarrow \boldsymbol{AB}^{H}$, where ***A*** and ***C*** are contained in larger arrays A and C, respectively, and ***B*** is the same size as the array B in which it is contained. The arrays contain complex data. This example shows how to code the calling sequence for CGEMMS, but does not use the Strassen algorithm for doing the computation. The calling sequence is shown below. The input and output, other than auxiliary storage, is the same as in "Example 8" on page 407 for CGEMUL.

Call Statement and Input

```
             A  LDA TRANSA  B  LDB TRANSB  C  LDC  L   M   N   AUX  NAUX
             |   |    |     |   |    |     |   |   |   |   |    |    |
CALL CGEMMS( A , 4 , 'N'  , B , 3 , 'C'  , C , 4 , 3 , 2 , 3 , AUX , 0 )
```

SGEMM, DGEMM, CGEMM, and ZGEMM—Combined Matrix Multiplication and Addition for General Matrices, Their Transposes, or Conjugate Transposes

SGEMM and DGEMM can perform any one of the following combined matrix computations, using scalars α and β, matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ or their transposes, and matrix $\boldsymbol{C}$:

$$\boldsymbol{C} \leftarrow \alpha\boldsymbol{AB}+\beta\boldsymbol{C} \qquad \boldsymbol{C} \leftarrow \alpha\boldsymbol{AB}^{T}+\beta\boldsymbol{C}$$
$$\boldsymbol{C} \leftarrow \alpha\boldsymbol{A}^{T}\boldsymbol{B}+\beta\boldsymbol{C} \qquad \boldsymbol{C} \leftarrow \alpha\boldsymbol{A}^{T}\boldsymbol{B}^{T}+\beta\boldsymbol{C}$$

CGEMM and ZGEMM can perform any one of the following combined matrix computations, using scalars α and β, matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, their transposes or their conjugate transposes, and matrix $\boldsymbol{C}$:

$$\boldsymbol{C} \leftarrow \alpha\boldsymbol{AB}+\beta\boldsymbol{C} \qquad \boldsymbol{C} \leftarrow \alpha\boldsymbol{AB}^{T}+\beta\boldsymbol{C} \qquad \boldsymbol{C} \leftarrow \alpha\boldsymbol{AB}^{H}+\beta\boldsymbol{C}$$
$$\boldsymbol{C} \leftarrow \alpha\boldsymbol{A}^{T}\boldsymbol{B}+\beta\boldsymbol{C} \qquad \boldsymbol{C} \leftarrow \alpha\boldsymbol{A}^{T}\boldsymbol{B}^{T}+\beta\boldsymbol{C} \qquad \boldsymbol{C} \leftarrow \alpha\boldsymbol{A}^{T}\boldsymbol{B}^{H}+\beta\boldsymbol{C}$$
$$\boldsymbol{C} \leftarrow \alpha\boldsymbol{A}^{H}\boldsymbol{B}+\beta\boldsymbol{C} \qquad \boldsymbol{C} \leftarrow \alpha\boldsymbol{A}^{H}\boldsymbol{B}^{T}+\beta\boldsymbol{C} \qquad \boldsymbol{C} \leftarrow \alpha\boldsymbol{A}^{H}\boldsymbol{B}^{H}+\beta\boldsymbol{C}$$

Table 76. Data Types

$\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, α, β	**Subroutine**
Short-precision real	SGEMM
Long-precision real	DGEMM
Short-precision complex	CGEMM
Long-precision complex	ZGEMM

Syntax

Fortran	CALL SGEMM | DGEMM | CGEMM | ZGEMM (*transa*, *transb*, *l*, *n*, *m*, *alpha*, *a*, *lda*, *b*, *ldb*, *beta*, *c*, *ldc*)
C and C++	sgemm | dgemm | cgemm | zgemm (*transa*, *transb*, *l*, *n*, *m*, *alpha*, *a*, *lda*, *b*, *ldb*, *beta*, *c*, *ldc*);
PL/I	CALL SGEMM | DGEMM | CGEMM | ZGEMM (*transa*, *transb*, *l*, *n*, *m*, *alpha*, *a*, *lda*, *b*, *ldb*, *beta*, *c*, *ldc*);

On Entry

transa
indicates the form of matrix $\boldsymbol{A}$ to use in the computation, where:

If *transa* = 'N', $\boldsymbol{A}$ is used in the computation.

If *transa* = 'T', $\boldsymbol{A}^{T}$ is used in the computation.

If *transa* = 'C', $\boldsymbol{A}^{H}$ is used in the computation.

Specified as: a single character; *transa* = 'N', 'T', or 'C'.

transb
indicates the form of matrix $\boldsymbol{B}$ to use in the computation, where:

If *transb* = 'N', $\boldsymbol{B}$ is used in the computation.

If *transb* = 'T', $\boldsymbol{B}^{T}$ is used in the computation.

If *transb* = 'C', $\boldsymbol{B}^H$ is used in the computation.

Specified as: a single character; *transb* = 'N', 'T', or 'C'.

l

is the number of rows in matrix ***C***. Specified as: a fullword integer; $0 \leq l \leq ldc$.

n

is the number of columns in matrix ***C***. Specified as: a fullword integer; $n \geq 0$.

m

has the following meaning, where:

If *transa* = 'N', it is the number of columns in matrix ***A***.

If *transa* = 'T' or 'C', it is the number of rows in matrix ***A***.

In addition:

If *transb* = 'N', it is the number of rows in matrix ***B***.

If *transb* = 'T' or 'C', it is the number of columns in matrix ***B***.

Specified as: a fullword integer; $m \geq 0$.

alpha

is the scalar α. Specified as: a number of the data type indicated in Table 76 on page 413.

a

is the matrix ***A***, where:

If *transa* = 'N', ***A*** is used in the computation, and ***A*** has *l* rows and *m* columns.

If *transa* = 'T', $\boldsymbol{A}^T$ is used in the computation, and ***A*** has *m* rows and *l* columns.

If *transa* = 'C', $\boldsymbol{A}^H$ is used in the computation, and ***A*** has *m* rows and *l* columns.

Note: No data should be moved to form $\boldsymbol{A}^T$ or $\boldsymbol{A}^H$; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 76 on page 413, where:

If *transa* = 'N', its size must be *lda* by (at least) *m*.

If *transa* = 'T' or 'C', its size must be *lda* by (at least) *l*.

lda

is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and:

If *transa* = 'N', $lda \geq l$.

If *transa* = 'T' or 'C', $lda \geq m$.

b

is the matrix ***B***, where:

If *transb* = 'N', ***B*** is used in the computation, and ***B*** has *m* rows and *n* columns.

If *transb* = 'T', $\boldsymbol{B}^T$ is used in the computation, and ***B*** has *n* rows and *m* columns.

If *transb* = 'C', $\boldsymbol{B}^H$ is used in the computation, and ***B*** has *n* rows and *m* columns.

Note: No data should be moved to form $\boldsymbol{B}^T$ or $\boldsymbol{B}^H$; that is, the matrix ***B*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 76 on page 413, where:

If *transb* = 'N', its size must be *ldb* by (at least) *n*.

If *transb* = 'T' or 'C', its size must be *ldb* by (at least) *m*.

ldb
is the leading dimension of the array specified for *b*. Specified as: a fullword integer; *ldb* > 0 and:

If *transb* = 'N', *ldb* ≥ *m*.

If *transb* = 'T' or 'C', *ldb* ≥ *n*.

beta
is the scalar β. Specified as: a number of the data type indicated in Table 76 on page 413.

c
is the *l* by *n* matrix ***C***. Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 76 on page 413.

ldc
is the leading dimension of the array specified for *c*. Specified as: a fullword integer; *ldc* > 0 and *ldc* ≥ *l*.

On Return

c
is the *l* by *n* matrix ***C***, containing the results of the computation. Returned as: an *ldc* by (at least) *n* array, containing numbers of the data type indicated in Table 76 on page 413.

Notes

1. All subroutines accept lowercase letters for the *transa* and *transb* arguments.
2. For SGEMM and DGEMM, if you specify 'C' for the *transa* or *transb* argument, it is interpreted as though you specified 'T'.
3. Matrix ***C*** must have no common elements with matrices ***A*** or ***B***; otherwise, results are unpredictable. See "Concepts" on page 55.

Function: The combined matrix addition and multiplication is expressed as follows, where a_{ik}, b_{kj}, and c_{ij} are elements of matrices ***A, B,*** and ***C***, respectively:

$$c_{ij} = \left(\alpha \sum_{k=1}^{m} a_{ik}\, b_{kj} \right) + \beta c_{ij} \quad \text{for } \boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\ \boldsymbol{B} + \beta \boldsymbol{C}$$

$$c_{ij} = \left(\alpha \sum_{k=1}^{m} a_{ki}\, b_{kj} \right) + \beta c_{ij} \quad \text{for } \boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^{\mathrm{T}}\ \boldsymbol{B} + \beta \boldsymbol{C}$$

$$c_{ij} = \left(\alpha \sum_{k=1}^{m} \overline{a}_{ki}\, b_{kj} \right) + \beta c_{ij} \quad \text{for } \boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^{\mathrm{H}}\ \boldsymbol{B} + \beta \boldsymbol{C}$$

$$c_{ij} = \left(\alpha \sum_{k=1}^{m} a_{ik}\, b_{jk} \right) + \beta c_{ij} \quad \text{for } \boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\ \boldsymbol{B}^{\mathrm{T}} + \beta \boldsymbol{C}$$

$$c_{ij} = \left(\alpha \sum_{k=1}^{m} a_{ki}\, b_{jk} \right) + \beta c_{ij} \quad \text{for } \boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^{\mathrm{T}}\ \boldsymbol{B}^{\mathrm{T}} + \beta \boldsymbol{C}$$

$$c_{ij} = \left(\alpha \sum_{k=1}^{m} \overline{a}_{ki}\, b_{jk} \right) + \beta c_{ij} \quad \text{for } \boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^{\mathrm{H}}\ \boldsymbol{B}^{\mathrm{T}} + \beta \boldsymbol{C}$$

$$c_{ij} = \left(\alpha \sum_{k=1}^{m} a_{ik}\, \overline{b}_{jk} \right) + \beta c_{ij} \quad \text{for } \boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\ \boldsymbol{B}^{\mathrm{H}} + \beta \boldsymbol{C}$$

$$c_{ij} = \left(\alpha \sum_{k=1}^{m} a_{ki}\, \overline{b}_{jk} \right) + \beta c_{ij} \quad \text{for } \boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^{\mathrm{T}}\ \boldsymbol{B}^{\mathrm{H}} + \beta \boldsymbol{C}$$

$$c_{ij} = \left(\alpha \sum_{k=1}^{m} \overline{a}_{ki}\, \overline{b}_{jk} \right) + \beta c_{ij} \quad \text{for } \boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^{\mathrm{H}}\ \boldsymbol{B}^{\mathrm{H}} + \beta \boldsymbol{C}$$

$$\text{for } \ i = 1,\ l \ \text{ and } \ j = 1, n$$

See references [32] and [38]. In the following three cases, no computation is performed:

- l is 0.
- n is 0.
- β is 1 and α is 0.

Assuming the above conditions do not exist, if $\beta \neq 1$ and m is 0, then $\beta \boldsymbol{C}$ is returned.

Special Usage

Equivalence Rules: The equivalence rules, defined for matrix multiplication of ***A*** and ***B*** in "Special Usage" on page 401, also apply to the matrix multiplication part of the computation performed by this subroutine. You should use the equivalent rules when you want to transpose or conjugate transpose the multiplication part of the computation. When coding the calling sequences for these cases, be careful to code your matrix arguments and dimension arguments in the order indicated by the rule. Also, be careful that your input and output array `C` has dimensions large enough to hold the resulting matrix. See "Example 4" on page 420.

Error Conditions

Resource Errors: Unable to allocate internal work area (CGEMM and ZGEMM only).

Computational Errors: None

Input-Argument Errors

1. *lda*, *ldb*, *ldc* ≤ 0
2. *l*, *m*, *n* < 0
3. *l* > *ldc*
4. *transa*, *transb* ≠ 'N', 'T', or 'C'
5. *transa* = 'N' and *l* > *lda*
6. *transa* = 'T' or 'C' and *m* > *lda*
7. *transb* = 'N' and *m* > *ldb*
8. *transb* = 'T' or 'C' and *n* > *ldb*

Example 1: This example shows the computation $\boldsymbol{C} \leftarrow \alpha\boldsymbol{AB}+\beta\boldsymbol{C}$, where $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ are contained in larger arrays A, B, and C, respectively.

Call Statement and Input

```
          TRANSA TRANSB  L   N   M  ALPHA  A  LDA  B  LDB  BETA  C  LDC
            |      |     |   |   |    |    |   |   |   |    |    |   |
CALL SGEMM( 'N'  , 'N'  , 6 , 4 , 5 , 1.0 , A , 8 , B , 6 , 2.0 , C , 7 )
```

```
        ⎡                                  ⎤
        ⎢  1.0   2.0  -1.0  -1.0   4.0  ⎥
        ⎢  2.0   0.0   1.0   1.0  -1.0  ⎥
        ⎢  1.0  -1.0  -1.0   1.0   2.0  ⎥
A    =  ⎢ -3.0   2.0   2.0   2.0   0.0  ⎥
        ⎢  4.0   0.0  -2.0   1.0  -1.0  ⎥
        ⎢ -1.0  -1.0   1.0  -3.0   2.0  ⎥
        ⎢   .     .     .     .     .   ⎥
        ⎣   .     .     .     .     .   ⎦
```

```
        ⎡                          ⎤
        ⎢  1.0  -1.0   0.0   2.0  ⎥
        ⎢  2.0   2.0  -1.0  -2.0  ⎥
B    =  ⎢  1.0   0.0  -1.0   1.0  ⎥
        ⎢ -3.0  -1.0   1.0  -1.0  ⎥
        ⎢  4.0   2.0  -1.0   1.0  ⎥
        ⎢   .     .     .     .   ⎥
        ⎣                          ⎦
```

```
        ⎡                        ⎤
        ⎢ 0.5  0.5  0.5  0.5  ⎥
        ⎢ 0.5  0.5  0.5  0.5  ⎥
        ⎢ 0.5  0.5  0.5  0.5  ⎥
C    =  ⎢ 0.5  0.5  0.5  0.5  ⎥
        ⎢ 0.5  0.5  0.5  0.5  ⎥
        ⎢ 0.5  0.5  0.5  0.5  ⎥
        ⎢  .    .    .    .   ⎥
        ⎣                        ⎦
```

Output

```
        ⎡                          ⎤
        ⎢ 24.0  13.0  -5.0   3.0 ⎥
        ⎢ -3.0  -4.0   2.0   4.0 ⎥
        ⎢  4.0   1.0   2.0   5.0 ⎥
C    =  ⎢ -2.0   6.0  -1.0  -9.0 ⎥
        ⎢ -4.0  -6.0   5.0   5.0 ⎥
        ⎢ 16.0   7.0  -4.0   7.0 ⎥
        ⎢   .     .     .     .  ⎥
        ⎣                          ⎦
```

Example 2: This example shows the computation $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{B}^{T}+\beta \boldsymbol{C}$, where $\boldsymbol{A}$ and $\boldsymbol{C}$ are contained in larger arrays `A` and `C`, respectively, and $\boldsymbol{B}$ is the same size as array `B` in which it is contained.

Call Statement and Input

```
          TRANSA TRANSB  L   N   M  ALPHA  A  LDA  B  LDB  BETA  C  LDC
            |      |     |   |   |    |    |   |   |   |    |    |   |
CALL SGEMM( 'N'  , 'T'  , 3 , 3 , 2 , 1.0 , A , 4 , B , 3 , 2.0 , C , 5 )
```

```
        ⎡           ⎤
        ⎢ 1.0  -3.0 ⎥
A    =  ⎢ 2.0   4.0 ⎥
        ⎢ 1.0  -1.0 ⎥
        ⎢  .     .  ⎥
        ⎣           ⎦
```

```
        ⎡           ⎤
        ⎢ 1.0  -3.0 ⎥
B    =  ⎢ 2.0   4.0 ⎥
        ⎢ 1.0  -1.0 ⎥
        ⎣           ⎦
```

```
        ⎡               ⎤
        ⎢ 0.5  0.5  0.5 ⎥
        ⎢ 0.5  0.5  0.5 ⎥
C    =  ⎢ 0.5  0.5  0.5 ⎥
        ⎢  .    .    .  ⎥
        ⎢  .    .    .  ⎥
        ⎣               ⎦
```

Output

```
        ⎡                    ⎤
        ⎢ 11.0  -9.0   5.0 ⎥
        ⎢ -9.0  21.0  -1.0 ⎥
C    =  ⎢  5.0  -1.0   3.0 ⎥
        ⎢   .     .     .  ⎥
        ⎢   .     .     .  ⎥
        ⎣                    ⎦
```

Example 3: This example shows the computation $\boldsymbol{C} \leftarrow \alpha\boldsymbol{AB}+\beta\boldsymbol{C}$ using complex data, where $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ are contained in larger arrays, A, B, and C, respectively.

Call Statement and Input

```
          TRANSA TRANSB  L   N   M   ALPHA   A  LDA  B  LDB  BETA   C  LDC
            |      |     |   |   |     |     |   |   |   |    |     |   |
CALL CGEMM( 'N'  , 'N'  , 6 , 2 , 3 , ALPHA , A , 8 , B , 4 , BETA , C , 8 )

ALPHA    =  (1.0, 0.0)
BETA     =  (2.0, 0.0)
```

```
        ⎡                                    ⎤
        ⎢ (1.0, 5.0)  (9.0, 2.0)  (1.0, 9.0) ⎥
        ⎢ (2.0, 4.0)  (8.0, 3.0)  (1.0, 8.0) ⎥
        ⎢ (3.0, 3.0)  (7.0, 5.0)  (1.0, 7.0) ⎥
A    =  ⎢ (4.0, 2.0)  (4.0, 7.0)  (1.0, 5.0) ⎥
        ⎢ (5.0, 1.0)  (5.0, 1.0)  (1.0, 6.0) ⎥
        ⎢ (6.0, 6.0)  (3.0, 6.0)  (1.0, 4.0) ⎥
        ⎢     .           .           .      ⎥
        ⎢     .           .           .      ⎥
        ⎣                                    ⎦
```

```
        ⎡                          ⎤
        ⎢ (1.0, 8.0)  (2.0, 7.0) ⎥
B    =  ⎢ (4.0, 4.0)  (6.0, 8.0) ⎥
        ⎢ (6.0, 2.0)  (4.0, 5.0) ⎥
        ⎢     .           .      ⎥
        ⎣                          ⎦
```

```
        ⎡                          ⎤
        ⎢ (0.5, 0.0)  (0.5, 0.0) ⎥
        ⎢ (0.5, 0.0)  (0.5, 0.0) ⎥
        ⎢ (0.5, 0.0)  (0.5, 0.0) ⎥
C    =  ⎢ (0.5, 0.0)  (0.5, 0.0) ⎥
        ⎢ (0.5, 0.0)  (0.5, 0.0) ⎥
        ⎢ (0.5, 0.0)  (0.5, 0.0) ⎥
        ⎢     .           .      ⎥
        ⎢     .           .      ⎥
        ⎣                          ⎦
```

Output

```
        ⎡                                    ⎤
        ⎢ (-22.0, 113.0)  (-35.0, 142.0) ⎥
        ⎢ (-19.0, 114.0)  (-35.0, 141.0) ⎥
        ⎢ (-20.0, 119.0)  (-43.0, 146.0) ⎥
C    =  ⎢ (-27.0, 110.0)  (-58.0, 131.0) ⎥
        ⎢   (8.0, 103.0)    (0.0, 112.0) ⎥
        ⎢ (-55.0, 116.0)  (-75.0, 135.0) ⎥
        ⎢       .               .        ⎥
        ⎢       .               .        ⎥
        ⎣                                    ⎦
```

Example 4: This example shows how to obtain the conjugate transpose of $\boldsymbol{AB}^{\mathrm{H}}$.

$$\left(\boldsymbol{AB}^{\mathrm{H}}\right)^{\mathrm{H}} = \overline{\overline{\boldsymbol{B}}\boldsymbol{A}^{\mathrm{T}}} = \boldsymbol{BA}^{\mathrm{H}}$$

This shows the conjugate transpose of the computation performed in "Example 8" on page 407 for CGEMUL, which uses the following calling sequence:

```
CALL CGEMUL( A , 4 , 'N' , B , 3 , 'C' , C , 4 , 3 , 2 , 3 )
```

You instead code the calling sequence for $\boldsymbol{C} \leftarrow \beta\boldsymbol{C}+\alpha\boldsymbol{BA}^{\mathrm{H}}$, where $\beta = 0$, $\alpha = 1$, and the array $\boldsymbol{C}$ has the correct dimensions to receive the transposed matrix. Because β is zero, $\beta\boldsymbol{C} = 0$. For a description of all the matrix identities, see "Special Usage" on page 401.

Call Statement and Input

```
           TRANSA TRANSB  L   N   M   ALPHA   A  LDA  B  LDB  BETA   C  LDC
             |      |     |   |   |     |     |   |   |   |    |     |   |
CALL CGEMM( 'N'  , 'C'  , 3 , 3 , 2 , ALPHA , B , 3 , A , 3 , BETA , C , 4 )

ALPHA    =  (1.0, 0.0)
BETA     =  (0.0, 0.0)

         ┌                        ┐
         │ (1.0, 3.0)  (-3.0, 2.0) │
B    =   │ (2.0, 5.0)   (4.0, 6.0) │
         │ (1.0, 1.0)  (-1.0, 9.0) │
         └                        ┘

         ┌                        ┐
         │ (1.0, 2.0)  (-3.0, 2.0) │
A    =   │ (2.0, 6.0)   (4.0, 5.0) │
         │ (1.0, 2.0)  (-1.0, 8.0) │
         │     .           .       │
         └                        ┘

C        =(not relevant)
```

Output

```
         ┌                                          ┐
         │ (20.0,   1.0)  (18.0, 23.0)  (26.0,  23.0) │
C    =   │ (12.0, -25.0)  (80.0,  2.0)  (56.0, -37.0) │
         │ (24.0, -26.0)  (49.0, 37.0)  (76.0,  -2.0) │
         │      .              .            .         │
         └                                          ┘
```

Example 5: This example shows the computation $C \leftarrow \alpha A^{T} B^{H} + \beta C$ using complex data, where ***A, B***, and ***C*** are the same size as the arrays A, B, and C, in which they are contained. Because β is zero, $\beta C = 0$. (Based on the dimensions of the matrices, ***A*** is actually a column vector, and ***C*** is actually a row vector.)

Call Statement and Input

```
           TRANSA TRANSB  L   N   M   ALPHA   A  LDA  B  LDB  BETA   C  LDC
             |      |     |   |   |     |     |   |   |   |    |     |   |
CALL CGEMM( 'T'  , 'C'  , 1 , 3 , 3 , ALPHA , A , 3 , B , 3 , BETA , C , 1 )

ALPHA    =  (1.0, 1.0)
BETA     =  (0.0, 0.0)

         ┌             ┐
         │ (1.0,  2.0) │
A    =   │ (2.0,  5.0) │
         │ (1.0,  6.0) │
         └             ┘

         ┌                                    ┐
         │ (1.0, 6.0)  (-3.0, 4.0)   (2.0, 6.0) │
B    =   │ (2.0, 3.0)   (4.0, 6.0)   (0.0, 3.0) │
         │ (1.0, 3.0)  (-1.0, 6.0)  (-1.0, 9.0) │
         └                                    ┘

C         =(not relevant)
```

Output

```
         ┌                                      ┐
A    =   │ (86.0, 44.0) (58.0, 70.0) (121.0, 55.0) │
         └                                      ┘
```

SSYMM, DSYMM, CSYMM, ZSYMM, CHEMM, and ZHEMM—Matrix-Matrix Product Where One Matrix is Real or Complex Symmetric or Complex Hermitian

These subroutines compute one of the following matrix-matrix products, using the scalars α and β and matrices ***A***, ***B***, and ***C***:

1. $\boldsymbol{C} \leftarrow \alpha\boldsymbol{AB}+\beta\boldsymbol{C}$
2. $\boldsymbol{C} \leftarrow \alpha\boldsymbol{BA}+\beta\boldsymbol{C}$

where matrix ***A*** is stored in either upper or lower storage mode, and:

- For SSYMM and DSYMM, matrix ***A*** is real symmetric.
- For CSYMM and ZSYMM, matrix ***A*** is complex symmetric.
- For CHEMM and ZHEMM, matrix ***A*** is complex Hermitian.

Table 77. Data Types

α, ***A***, ***B***, β, ***C***	**Subprogram**
Short-precision real	SSYMM
Long-precision real	DSYMM
Short-precision complex	CSYMM and CHEMM
Long-precision complex	ZSYMM and ZHEMM

Syntax

Fortran	CALL SSYMM | DSYMM | CSYMM | ZSYMM | CHEMM | ZHEMM (*side*, *uplo*, *m*, *n*, *alpha*, *a*, *lda*, *b*, *ldb*, *beta*, *c*, *ldc*)
C and C++	ssymm | dsymm | csymm | zsymm | chemm | zhemm (*side*, *uplo*, *m*, *n*, *alpha*, *a*, *lda*, *b*, *ldb*, *beta*, *c*, *ldc*);
PL/I	CALL SSYMM | DSYMM | CSYMM | ZSYMM | CHEMM | ZHEMM (*side*, *uplo*, *m*, *n*, *alpha*, *a*, *lda*, *b*, *ldb*, *beta*, *c*, *ldc*);

On Entry

side
indicates whether matrix ***A*** is located to the left or right of rectangular matrix ***B*** in the equation used for this computation, where:

If *side* = 'L', ***A*** is to the left of ***B***, resulting in equation 1.

If *side* = 'R', ***A*** is to the right of ***B***, resulting in equation 2.

Specified as: a single character. It must be 'L' or 'R'.

uplo
indicates the storage mode used for matrix ***A***, where:

If *uplo* = 'U', ***A*** is stored in upper storage mode.

If *uplo* = 'L', ***A*** is stored in lower storage mode.

Specified as: a single character. It must be 'U' or 'L'.

m
is the number of rows in rectangular matrices ***B*** and ***C***, and:

If *side* = 'L', *m* is the order of triangular matrix ***A***.

Specified as: a fullword integer; $0 \le m \le ldb$, $m \le ldc$, and:

If *side* = 'L', $m \le lda$.

n
is the number of columns in rectangular matrices ***B*** and ***C***, and:

If *side* = 'R', *n* is the order of triangular matrix ***A***.

Specified as: a fullword integer; $n \ge 0$ and:

If *side* = 'R', $n \le lda$.

alpha
is the scalar α. Specified as: a number of the data type indicated in Table 77 on page 422.

a
is the real symmetric, complex symmetric, or complex Hermitian matrix ***A***, where:

If *side* = 'L', ***A*** is order *m*.

If *side* = 'R', ***A*** is order *n*.

and where it is stored as follows:

If *uplo* = 'U', ***A*** is stored in upper storage mode.

If *uplo* = 'L', ***A*** is stored in lower storage mode.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 77 on page 422, where:

If *side* = 'L', its size must be *lda* by (at least) *m*.

If *side* = 'R', it size must be *lda* by (at least) *n*.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and:

If *side* = 'L', $lda \ge m$.

If *side* = 'R', $lda \ge n$.

b
is the *m* by *n* rectangular matrix ***B***. Specified as: an *ldb* by (at least) *n* array, containing numbers of the data type indicated in Table 77 on page 422.

ldb
is the leading dimension of the array specified for *b*. Specified as: a fullword integer; $ldb > 0$ and $ldb \ge m$.

beta
is the scalar β. Specified as: a number of the data type indicated in Table 77 on page 422.

c
is the *m* by *n* rectangular matrix ***C***. Specified as: an *ldc* by (at least) *n* array, containing numbers of the data type indicated in Table 77 on page 422.

ldc
is the leading dimension of the array specified for *c*. Specified as: a fullword integer; $ldc > 0$ and $ldc \ge m$.

On Return

c
is the *m* by *n* matrix ***C***, containing the results of the computation.

Returned as: an *ldc* by (at least) *n* array, containing numbers of the data type indicated in Table 77 on page 422.

Notes

1. These subroutines accept lowercase letters for the *side* and *uplo* arguments.
2. Matrices ***A***, ***B***, and ***C*** must have no common elements; otherwise, results are unpredictable.
3. If matrix ***A*** is upper triangular (*uplo* = 'U'), these subroutines use only the data in the upper triangular portion of the array. If matrix ***A*** is lower triangular, (*uplo* = 'L'), these subroutines use only the data in the lower triangular portion of the array. In each case, the other portion of the array is altered during the computation, but restored before exit.
4. The imaginary parts of the diagonal elements of a complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values.
5. For a description of how symmetric matrices are stored in upper and lower storage mode, see "Symmetric Matrix" on page 65. For a description of how complex Hermitian matrices are stored in upper and lower storage mode, see "Complex Hermitian Matrix" on page 70.

Function: These subroutines can perform the following matrix-matrix product computations using matrix ***A***, which is real symmetric for SSYMM and DSYMM, complex symmetric for CSYMM and ZSYMM, and complex Hermitian for CHEMM and ZHEMM:

1. $\boldsymbol{C} \leftarrow \alpha\boldsymbol{AB}+\beta\boldsymbol{C}$
2. $\boldsymbol{C} \leftarrow \alpha\boldsymbol{BA}+\beta\boldsymbol{C}$

where:

α and β are scalars.
A is a matrix of the type indicated above, stored in upper or lower storage mode. It is order *m* for equation 1 and order *n* for equation 2.
B and ***C*** are *m* by *n* rectangular matrices.

See references [32] and [38]. In the following two cases, no computation is performed:

- *n* or *m* is 0.
- β is one and α is zero.

Error Conditions

Resource Errors: Unable to allocate internal work area.

Computational Errors: None

Input-Argument Errors

1. $m < 0$
2. $m > ldb$

3. $m > ldc$
4. $n < 0$
5. $lda, ldb, ldc \leq 0$
6. $side \neq$ 'L' or 'R'
7. $uplo \neq$ 'L' or 'U'
8. $side =$ 'L' and $m > lda$
9. $side =$ 'R' and $n > lda$

Example 1: This example shows the computation $\boldsymbol{C} \leftarrow \alpha\boldsymbol{AB}+\beta\boldsymbol{C}$, where $\boldsymbol{A}$ is a real symmetric matrix of order 5, stored in upper storage mode, and $\boldsymbol{B}$ and $\boldsymbol{C}$ are 5 by 4 rectangular matrices.

Call Statement and Input

```
            SIDE  UPLO  M   N   ALPHA   A  LDA  B  LDB  BETA  C  LDC
             |     |    |   |     |     |   |   |   |    |    |   |
CALL SSYMM( 'L' , 'U' , 5 , 4 ,  2.0  , A , 8 , B , 6 , 1.0 , C , 5 )
```

```
        [                                  ]
        | 1.0  2.0  -1.0  -1.0   4.0 |
        |  .   0.0   1.0   1.0  -1.0 |
        |  .    .   -1.0   1.0   2.0 |
A    =  |  .    .     .    2.0   0.0 |
        |  .    .     .     .   -1.0 |
        |  .    .     .     .     .  |
        |  .    .     .     .     .  |
        |  .    .     .     .     .  |
        [                                  ]
```

```
        [                          ]
        |  1.0  -1.0   0.0   2.0 |
        |  2.0   2.0  -1.0  -2.0 |
B    =  |  1.0   0.0  -1.0   1.0 |
        | -3.0  -1.0   1.0  -1.0 |
        |  4.0   2.0  -1.0   1.0 |
        |   .     .     .     .  |
        [                          ]
```

```
        [                          ]
        | 23.0  12.0  -6.0   2.0 |
        | -4.0  -5.0   1.0   3.0 |
C    =  |  5.0   6.0  -1.0  -4.0 |
        | -4.0   1.0   0.0  -5.0 |
        |  8.0  -4.0  -2.0  13.0 |
        [                          ]
```

Output

```
        [                                  ]
        |  69.0   36.0  -18.0    6.0 |
        | -12.0  -15.0    3.0    9.0 |
C    =  |  15.0   18.0   -3.0  -12.0 |
        | -12.0    3.0    0.0  -15.0 |
        |   8.0  -20.0   -2.0   35.0 |
        [                                  ]
```

Example 2: This example shows the computation $C \leftarrow \alpha AB+\beta C$, where A is a real symmetric matrix of order 3, stored in lower storage mode, and B and C are 3 by 6 rectangular matrices.

Call Statement and Input

```
            SIDE  UPLO  M   N   ALPHA   A  LDA  B  LDB  BETA  C  LDC
             |     |    |   |     |     |   |   |   |    |    |   |
CALL SSYMM( 'L' , 'L' , 3 , 6 ,  2.0  , A , 4 , B , 3 , 2.0 , C , 5 )
```

```
        [                 ]
        | 1.0    .     .  |
A    =  | 2.0   4.0    .  |
        | 1.0  -1.0  -1.0 |
        |  .     .     .  |
        [                 ]
```

```
        [                                   ]
        | 1.0  -3.0   2.0   2.0  -1.0   2.0 |
B    =  | 2.0   4.0   0.0   0.0   1.0  -2.0 |
        | 1.0  -1.0  -1.0  -1.0  -1.0   1.0 |
        [                                   ]
```

```
        [                                   ]
        |  6.0   4.0  1.0  1.0   0.0  -1.0 |
        |  9.0  11.0  5.0  5.0   3.0  -5.0 |
C    =  | -2.0  -6.0  3.0  3.0  -1.0  32.0 |
        |   .     .    .    .     .     .  |
        |   .     .    .    .     .     .  |
        [                                   ]
```

Output

```
        [                                     ]
        | 24.0   16.0   4.0   4.0   0.0   -4.0 |
        | 36.0   44.0  20.0  20.0  12.0  -20.0 |
C    =  | -8.0  -24.0  12.0  12.0  -4.0   12.0 |
        |   .      .     .     .     .      .  |
        |   .      .     .     .     .      .  |
        [                                     ]
```

Example 3: This example shows the computation $C \leftarrow \alpha BA+\beta C$, where A is a real symmetric matrix of order 3, stored in upper storage mode, and B and C are 2 by 3 rectangular matrices.

Call Statement and Input

```
            SIDE  UPLO  M   N   ALPHA   A  LDA  B  LDB  BETA  C  LDC
             |     |    |   |     |     |   |   |   |    |    |   |
CALL SSYMM( 'R' , 'U' , 2 , 3 ,  2.0  , A , 4 , B , 3 , 1.0 , C , 5 )
```

```
        [                 ]
        | 1.0  -3.0   1.0 |
A    =  |  .    4.0  -1.0 |
        |  .     .    2.0 |
        |  .     .     .  |
        [                 ]
```

```
        ⎡                  ⎤
        ⎢ 1.0  -3.0   3.0 ⎥
B    =  ⎢ 2.0   4.0  -1.0 ⎥
        ⎢  .     .     .  ⎥
        ⎣                  ⎦

        ⎡                    ⎤
        ⎢  13.0  -18.0  10.0 ⎥
        ⎢ -11.0   11.0  -4.0 ⎥
C    =  ⎢    .      .     .  ⎥
        ⎢    .      .     .  ⎥
        ⎢    .      .     .  ⎥
        ⎣                    ⎦
```

Output

```
        ⎡                    ⎤
        ⎢  39.0  -54.0   30.0 ⎥
        ⎢ -33.0   33.0  -12.0 ⎥
C    =  ⎢    .      .      .  ⎥
        ⎢    .      .      .  ⎥
        ⎢    .      .      .  ⎥
        ⎣                    ⎦
```

Example 4: This example shows the computation $\boldsymbol{C} \leftarrow \alpha\boldsymbol{BA}+\beta\boldsymbol{C}$, where $\boldsymbol{A}$ is a real symmetric matrix of order 3, stored in lower storage mode, and $\boldsymbol{B}$ and $\boldsymbol{C}$ are 3 by 3 square matrices.

Call Statement and Input

```
            SIDE  UPLO  M   N   ALPHA   A  LDA  B  LDB  BETA  C  LDC
             |     |    |   |     |     |   |   |   |    |    |   |
CALL SSYMM( 'R' , 'L' , 3 , 3 , -1.0  , A , 3 , B , 3 , 1.0 , C , 3 )

        ⎡                  ⎤
        ⎢ 1.0    .    .  ⎥
A    =  ⎢ 2.0  10.0   .  ⎥
        ⎢ 1.0  11.0  4.0 ⎥
        ⎣                  ⎦

        ⎡                  ⎤
        ⎢ 1.0  -3.0   2.0 ⎥
B    =  ⎢ 2.0   4.0   0.0 ⎥
        ⎢ 1.0  -1.0  -1.0 ⎥
        ⎣                  ⎦

        ⎡                  ⎤
        ⎢  1.0   5.0  -9.0 ⎥
C    =  ⎢ -3.0  10.0  -2.0 ⎥
        ⎢ -2.0   8.0   0.0 ⎥
        ⎣                  ⎦
```

Output

```
        ⎡                    ⎤
        ⎢   4.0   11.0   15.0 ⎥
C    =  ⎢ -13.0  -34.0  -48.0 ⎥
        ⎢   0.0   27.0   14.0 ⎥
        ⎣                    ⎦
```

Example 5: This example shows the computation $C \leftarrow \alpha BA+\beta C$, where A is a complex symmetric matrix of order 3, stored in upper storage mode, and B and C are 2 by 3 rectangular matrices.

Call Statement and Input

```
            SIDE  UPLO  M   N   ALPHA   A  LDA  B  LDB  BETA   C  LDC
             |     |    |   |     |     |   |   |   |    |     |   |
CALL CSYMM( 'R' , 'U' , 2 , 3 , ALPHA , A , 4 , B , 3 , BETA , C , 5 )

ALPHA    =  (2.0, 3.0)

BETA     =  (1.0, 6.0)

        ⌈                                    ⌉
        | (1.0, 5.0)  (-3.0, 2.0)   (1.0, 6.0) |
A    =  |     .        (4.0, 5.0)  (-1.0, 4.0) |
        |     .            .        (2.0, 5.0) |
        |     .            .            .      |
        ⌊                                    ⌋

        ⌈                                    ⌉
        | (1.0, 1.0)  (-3.0, 2.0)   (3.0, 3.0) |
B    =  | (2.0, 6.0)   (4.0, 5.0)  (-1.0, 4.0) |
        |     .            .            .      |
        ⌊                                    ⌋

        ⌈                                    ⌉
        |  (13.0, 6.0)  (-18.0, 6.0)  (10.0, 7.0) |
        | (-11.0, 8.0)   (11.0, 1.0)  (-4.0, 2.0) |
C    =  |       .            .            .      |
        |       .            .            .      |
        |       .            .            .      |
        ⌊                                    ⌋
```

Output

```
        ⌈                                                    ⌉
        |  (-96.0,   72.0)  (-141.0, -226.0)  (-112.0,   38.0) |
        | (-230.0, -269.0)  (-133.0,  -23.0)  (-272.0, -198.0) |
C    =  |        .                 .                 .         |
        |        .                 .                 .         |
        |        .                 .                 .         |
        ⌊                                                    ⌋
```

Example 6: This example shows the computation $C \leftarrow \alpha BA+\beta C$, where A is a complex Hermitian matrix of order 3, stored in lower storage mode, and B and C are 3 by 3 square matrices.

Note: The imaginary parts of the diagonal elements of a complex Hermitian matrix are assumed to be zero, so you do not have to set these values.

Call Statement and Input

```
            SIDE  UPLO  M   N   ALPHA   A  LDA  B  LDB  BETA   C  LDC
             |     |    |   |     |     |   |   |   |    |     |   |
CALL CHEMM( 'R' , 'L' , 2 , 3 , ALPHA , A , 4 , B , 3 , BETA , C , 5 )

ALPHA    =  (2.0, 3.0)

BETA     =  (1.0, 6.0)
```

```
        ⎡                                    ⎤
        ⎢  (1.0,  . )      .          .      ⎥
A    =  ⎢  (3.0, 2.0)  (4.0,  . )      .      ⎥
        ⎢ (-1.0, 6.0)  (1.0, 4.0)  (2.0,  . ) ⎥
        ⎢      .          .          .      ⎥
        ⎣                                    ⎦

        ⎡                                    ⎤
        ⎢ (1.0, 1.0)  (-3.0, 2.0)   (3.0, 3.0) ⎥
B    =  ⎢ (2.0, 6.0)   (4.0, 5.0)  (-1.0, 4.0) ⎥
        ⎢     .           .           .      ⎥
        ⎣                                    ⎦

        ⎡                                    ⎤
        ⎢  (13.0, 6.0)  (-18.0, 6.0)  (10.0, 7.0) ⎥
        ⎢ (-11.0, 8.0)   (11.0, 1.0)  (-4.0, 2.0) ⎥
C    =  ⎢       .            .           .      ⎥
        ⎢       .            .           .      ⎥
        ⎢       .            .           .      ⎥
        ⎣                                    ⎦
```

Output

```
        ⎡                                                ⎤
        ⎢ (-137.0,  17.0)  (-158.0, -102.0)  (-39.0, 141.0) ⎥
        ⎢ (-154.0, -77.0)   (-63.0,  186.0)  (159.0, 104.0) ⎥
C    =  ⎢        .               .                .        ⎥
        ⎢        .               .                .        ⎥
        ⎢        .               .                .        ⎥
        ⎣                                                ⎦
```

STRMM, DTRMM, CTRMM, and ZTRMM—Triangular Matrix-Matrix Product

STRMM and DTRMM compute one of the following matrix-matrix products, using the scalar α, rectangular matrix $\boldsymbol{B}$, and triangular matrix $\boldsymbol{A}$ or its transpose:

1. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{AB}$
2. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{A}^{\mathrm{T}}\boldsymbol{B}$
3. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{BA}$
4. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{BA}^{\mathrm{T}}$

CTRMM and ZTRMM compute one of the following matrix-matrix products, using the scalar α, rectangular matrix $\boldsymbol{B}$, and triangular matrix $\boldsymbol{A}$, its transpose, or its conjugate transpose:

1. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{AB}$
2. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{A}^{\mathrm{T}}\boldsymbol{B}$
3. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{BA}$
4. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{BA}^{\mathrm{T}}$
5. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{A}^{\mathrm{H}}\boldsymbol{B}$
6. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{BA}^{\mathrm{H}}$

Table 78. Data Types

$\boldsymbol{A}$, $\boldsymbol{B}$, α	**Subroutine**
Short-precision real	STRMM
Long-precision real	DTRMM
Short-precision complex	CTRMM
Long-precision complex	ZTRMM

Syntax

Fortran	CALL STRMM | DTRMM | CTRMM | ZTRMM (*side*, *uplo*, *transa*, *diag*, *m*, *n*, *alpha*, *a*, *lda*, *b*, *ldb*)
C and C++	strmm | dtrmm | ctrmm | ztrmm (*side*, *uplo*, *transa*, *diag*, *m*, *n*, *alpha*, *a*, *lda*, *b*, *ldb*);
PL/I	CALL STRMM | DTRMM | CTRMM | ZTRMM (*side*, *uplo*, *transa*, *diag*, *m*, *n*, *alpha*, *a*, *lda*, *b*, *ldb*);

On Entry

side
indicates whether the triangular matrix $\boldsymbol{A}$ is located to the left or right of rectangular matrix $\boldsymbol{B}$ in the equation used for this computation, where:

If *side* = 'L', $\boldsymbol{A}$ is to the left of $\boldsymbol{B}$ in the equation, resulting in either equation 1, 2, or 5.

If *side* = 'R', $\boldsymbol{A}$ is to the right of $\boldsymbol{B}$ in the equation, resulting in either equation 3, 4, or 6.

Specified as: a single character. It must be 'L' or 'R'.

uplo
indicates whether matrix $\boldsymbol{A}$ is an upper or lower triangular matrix, where:

If *uplo* = 'U', $\boldsymbol{A}$ is an upper triangular matrix.

If *uplo* = 'L', $\boldsymbol{A}$ is a lower triangular matrix.

Specified as: a single character. It must be 'U' or 'L'.

transa
indicates the form of matrix ***A*** to use in the computation, where:

If *transa* = 'N', ***A*** is used in the computation, resulting in either equation 1 or 3.

If *transa* = 'T', $\boldsymbol{A}^{T}$ is used in the computation, resulting in either equation 2 or 4.

If *transa* = 'C', $\boldsymbol{A}^{H}$ is used in the computation, resulting in either equation 5 or 6.

Specified as: a single character. It must be 'N', 'T', or 'C'.

diag
indicates the characteristics of the diagonal of matrix ***A***, where:

If *diag* = 'U', ***A*** is a unit triangular matrix.

If *diag* = 'N', ***A*** is not a unit triangular matrix.

Specified as: a single character. It must be 'U' or 'N'.

m
is the number of rows in rectangular matrix ***B***, and:

If *side* = 'L', *m* is the order of triangular matrix ***A***.

Specified as: a fullword integer, where:

If *side* = 'L', $0 \le m \le lda$ and $m \le ldb$.

If *side* = 'R', $0 \le m \le ldb$.

n
is the number of columns in rectangular matrix ***B***, and:

If *side* = 'R', *n* is the order of triangular matrix ***A***.

Specified as: a fullword integer; $n \ge 0$ and:

If *side* = 'R', $n \le lda$.

alpha
is the scalar α. Specified as: a number of the data type indicated in Table 78 on page 430.

a
is the triangular matrix ***A***, of which only the upper or lower triangular portion is used, where:

If *side* = 'L', ***A*** is order *m*.

If *side* = 'R', ***A*** is order *n*.

Note: No data should be moved to form $\boldsymbol{A}^{T}$ or $\boldsymbol{A}^{H}$; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 78 on page 430, where:

If *side* = 'L', its size must be *lda* by (at least) *m*.

If *side* = 'R', it size must be *lda* by (at least) *n*.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and:

If *side* = 'L', $lda \ge m$.

If *side* = 'R', $lda \ge n$.

b
is the *m* by *n* rectangular matrix ***B***. Specified as: an *ldb* by (at least) *n* array, containing numbers of the data type indicated in Table 78 on page 430.

ldb
is the leading dimension of the array specified for *b*. Specified as: a fullword integer; $ldb > 0$ and $ldb \geq m$.

On Return

b
is the *m* by *n* matrix ***B***, containing the results of the computation. Returned as: an *ldb* by (at least) *n* array, containing numbers of the data type indicated in Table 78 on page 430.

Notes

1. These subroutines accept lowercase letters for the *side*, *uplo*, *transa*, and *diag* arguments.
2. For STRMM and DTRMM, if you specify `'C'` for the *transa* argument, it is interpreted as though you specified `'T'`.
3. Matrices ***A*** and ***B*** must have no common elements; otherwise, results are unpredictable.
4. ESSL assumes certain values in your array for parts of a triangular matrix. As a result, you do not have to set these values. For unit triangular matrices, the elements of the diagonal are assumed to be 1.0 for real matrices and (1.0, 0.0) for complex matrices. When using upper- or lower-triangular storage, the unreferenced elements in the lower and upper triangular part, respectively, are assumed to be zero.
5. For a description of triangular matrices and how they are stored, see "Triangular Matrix" on page 74.

Function: These subroutines can perform the following matrix-matrix product computations, using the triangular matrix ***A***, its transpose, or its conjugate transpose, where ***A*** can be either upper- or lower-triangular:

1. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{AB}$
2. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{A}^{T}\boldsymbol{B}$
3. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{A}^{H}\boldsymbol{B}$ (for CTRMM and ZTRMM only)

   where:

   α is a scalar.
   A is a triangular matrix of order *m*.
   B is an *m* by *n* rectangular matrix.
4. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{BA}$
5. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{BA}^{T}$
6. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{BA}^{H}$ (for CTRMM and ZTRMM only)

   where:

   α is a scalar.
   A is a triangular matrix of order *n*.
   B is an *m* by *n* rectangular matrix.

See references [32] and [38]. If *n* or *m* is 0, no computation is performed.

Error Conditions

Resource Errors: Unable to allocate internal work area.

Computational Errors: None

Input-Argument Errors

1. $m < 0$
2. $n < 0$
3. $lda, ldb \leq 0$
4. *side* ≠ 'L' or 'R'
5. *uplo* ≠ 'L' or 'U'
6. *transa* ≠ 'T', 'N', or 'C'
7. *diag* ≠ 'N' or 'U'
8. *side* = 'L' and $m > lda$
9. $m > ldb$
10. *side* = 'R' and $n > lda$

Example 1: This example shows the computation $\boldsymbol{B} \leftarrow \alpha\boldsymbol{AB}$, where $\boldsymbol{A}$ is a 5 by 5 upper triangular matrix that is not unit triangular, and $\boldsymbol{B}$ is a 5 by 3 rectangular matrix.

Call Statement and Input

```
            SIDE  UPLO TRANSA  DIAG  M   N   ALPHA   A  LDA  B  LDB
             |     |     |      |    |   |     |     |   |   |   |
CALL STRMM( 'L' , 'U' , 'N'  , 'N' , 5 , 3 ,  1.0  , A , 7 , B , 6 )
```

$$A = \begin{bmatrix} 3.0 & -1.0 & 2.0 & 2.0 & 1.0 \\ . & -2.0 & 4.0 & -1.0 & 3.0 \\ . & . & -3.0 & 0.0 & 2.0 \\ . & . & . & 4.0 & -2.0 \\ . & . & . & . & 1.0 \\ . & . & . & . & . \\ . & . & . & . & . \end{bmatrix}$$

$$B = \begin{bmatrix} 2.0 & 3.0 & 1.0 \\ 5.0 & 5.0 & 4.0 \\ 0.0 & 1.0 & 2.0 \\ 3.0 & 1.0 & -3.0 \\ -1.0 & 2.0 & 1.0 \\ . & . & . \end{bmatrix}$$

Output

```
          ┌                    ┐
          │   6.0  10.0   -2.0 │
          │ -16.0  -1.0    6.0 │
B    =    │  -2.0   1.0   -4.0 │
          │  14.0   0.0  -14.0 │
          │  -1.0   2.0    1.0 │
          │    .     .      .  │
          └                    ┘
```

Example 2: This example shows the computation $B \leftarrow \alpha A^{T}B$, where A is a 5 by 5 upper triangular matrix that is not unit triangular, and B is a 5 by 4 rectangular matrix.

Call Statement and Input

```
            SIDE  UPLO TRANSA  DIAG  M   N    ALPHA  A  LDA  B  LDB
             |     |     |      |    |   |     |     |   |   |   |
CALL STRMM( 'L' , 'U' , 'T'  , 'N' , 5 , 4 ,  1.0  , A , 7 , B , 6 )

          ┌                          ┐
          │ -1.0  -4.0  -2.0   2.0   3.0 │
          │   .   -2.0   2.0   2.0   2.0 │
          │   .     .   -3.0  -1.0   4.0 │
A    =    │   .     .     .    1.0   0.0 │
          │   .     .     .     .   -2.0 │
          │   .     .     .     .     .  │
          │   .     .     .     .     .  │
          └                          ┘

          ┌                    ┐
          │  1.0   2.0   3.0   4.0 │
          │  3.0   3.0  -1.0   2.0 │
B    =    │ -2.0  -1.0   0.0   1.0 │
          │  4.0   4.0  -3.0  -3.0 │
          │  2.0   2.0   2.0   2.0 │
          │   .     .     .     .  │
          └                    ┘
```

Output

```
          ┌                    ┐
          │ -1.0  -2.0   -3.0   -4.0 │
          │  2.0  -2.0  -14.0  -12.0 │
B    =    │ 10.0   5.0   -8.0   -7.0 │
          │ 14.0  15.0    1.0    8.0 │
          │ -3.0   4.0    3.0   16.0 │
          │   .     .      .      .  │
          └                    ┘
```

Example 3: This example shows the computation $B \leftarrow \alpha BA$, where A is a 5 by 5 lower triangular matrix that is not unit triangular, and B is a 3 by 5 rectangular matrix.

Call Statement and Input

```
            SIDE  UPLO TRANSA  DIAG  M   N    ALPHA  A  LDA  B  LDB
             |     |     |      |    |   |     |     |   |   |   |
CALL STRMM( 'R' , 'L' , 'N'  , 'N' , 3 , 5 ,  1.0  , A , 7 , B , 4 )
```

```
        ⎡                                  ⎤
        ⎢ 2.0   .     .     .     .      ⎥
        ⎢ 2.0  3.0    .     .     .      ⎥
        ⎢ 2.0  1.0   1.0    .     .      ⎥
A    =  ⎢ 0.0  3.0   0.0  -2.0    .      ⎥
        ⎢ 2.0  4.0  -1.0   2.0  -1.0     ⎥
        ⎢  .    .     .     .     .      ⎥
        ⎣  .    .     .     .     .      ⎦

        ⎡                                  ⎤
        ⎢  3.0   4.0  -1.0  -1.0  -1.0    ⎥
B    =  ⎢  2.0   1.0  -1.0   0.0   3.0    ⎥
        ⎢ -2.0  -1.0  -3.0   0.0   2.0    ⎥
        ⎣   .     .     .     .     .      ⎦
```

Output

```
        ⎡                                  ⎤
        ⎢ 10.0   4.0   0.0  0.0   1.0     ⎥
B    =  ⎢ 10.0  14.0  -4.0  6.0  -3.0     ⎥
        ⎢ -8.0   2.0  -5.0  4.0  -2.0     ⎥
        ⎣   .     .     .    .     .      ⎦
```

Example 4: This example shows the computation $\boldsymbol{B} \leftarrow \alpha\boldsymbol{BA}$, where $\boldsymbol{A}$ is a 6 by 6 upper triangular matrix that is unit triangular, and $\boldsymbol{B}$ is a 1 by 6 rectangular matrix.

Note: Because matrix $\boldsymbol{A}$ is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of 1.0 for the diagonal elements.

Call Statement and Input

```
            SIDE  UPLO TRANSA  DIAG  M   N    ALPHA  A  LDA  B  LDB
             |     |     |      |    |   |     |     |   |   |   |
CALL STRMM( 'R' , 'U' , 'N'  , 'U' , 1 , 6 ,  1.0  , A , 7 , B , 2 )

        ⎡                                  ⎤
        ⎢ .  2.0  -3.0  1.0   2.0   4.0    ⎥
        ⎢ .   .    0.0  1.0   1.0  -2.0    ⎥
        ⎢ .   .     .   4.0  -1.0   1.0    ⎥
A    =  ⎢ .   .     .    .    0.0  -1.0    ⎥
        ⎢ .   .     .    .     .    2.0    ⎥
        ⎢ .   .     .    .     .     .     ⎥
        ⎣ .   .     .    .     .     .     ⎦

        ⎡                                  ⎤
B    =  ⎢ 1.0  2.0  1.0  3.0  -1.0  -2.0    ⎥
        ⎣  .    .    .    .     .     .     ⎦
```

Output

```
        ⎡                                  ⎤
B    =  ⎢ 1.0  4.0  -2.0  10.0  2.0  -6.0    ⎥
        ⎣  .    .     .     .    .     .     ⎦
```

Example 5: This example shows the computation $\boldsymbol{B} \leftarrow \alpha \boldsymbol{A}^{H}\boldsymbol{B}$, where $\boldsymbol{A}$ is a 5 by 5 upper triangular matrix that is not unit triangular, and $\boldsymbol{B}$ is a 5 by 1 rectangular matrix.

Call Statement and Input

```
            SIDE  UPLO TRANSA  DIAG  M   N    ALPHA   A  LDA  B  LDB
             |     |     |      |    |   |      |     |   |   |   |
CALL CTRMM( 'L' , 'U' , 'C'  , 'N' , 5 , 1 ,  ALPHA , A , 6 , B , 6 )

ALPHA    =  (1.0, 0.0)

        ⌈                                                                ⌉
        | (-4.0, 1.0)  (4.0, -3.0)  (-1.0,  3.0)  (0.0,  0.0)  (-1.0,  0.0) |
        |      .      (-2.0,  0.0)  (-3.0, -1.0) (-2.0, -1.0)   (4.0,  3.0) |
A    =  |      .           .        (-5.0,  3.0) (-3.0, -3.0)  (-5.0, -5.0) |
        |      .           .             .        (4.0, -4.0)   (2.0,  0.0) |
        |      .           .             .            .         (2.0, -1.0) |
        |      .           .             .            .             .       |
        ⌊                                                                ⌋

        ⌈              ⌉
        |  (3.0, 4.0)  |
        | (-4.0, 2.0)  |
B    =  | (-5.0, 0.0)  |
        |  (1.0, 3.0)  |
        |  (3.0, 1.0)  |
        |      .       |
        ⌊              ⌋
```

Output

```
        ⌈                ⌉
        | (-8.0, -19.0)  |
        |  (8.0,  21.0)  |
B    =  | (44.0,  -8.0)  |
        | (13.0,  -7.0)  |
        | (19.0,   2.0)  |
        |      .         |
        ⌊                ⌋
```

SSYRK, DSYRK, CSYRK, ZSYRK, CHERK, and ZHERK—Rank-K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix

These subroutines compute one of the following rank-k updates, where matrix ***C*** is stored in either upper or lower storage mode. SSYRK, DSYRK, CSYRK, and ZSYRK use the scalars α and β, real or complex matrix ***A*** or its transpose, and real or complex symmetric matrix ***C*** to compute:

1. $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{A}^{T}+\beta \boldsymbol{C}$
2. $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^{T}\boldsymbol{A}+\beta \boldsymbol{C}$

CHERK and ZHERK use the scalars α and β, complex matrix ***A*** or its complex conjugate transpose, and complex Hermitian matrix ***C*** to compute:

3. $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{A}^{H}+\beta \boldsymbol{C}$
4. $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^{H}\boldsymbol{A}+\beta \boldsymbol{C}$

Table 79. Data Types

A, ***C***	α, β	**Subprogram**
Short-precision real	Short-precision real	SSYRK
Long-precision real	Long-precision real	DSYRK
Short-precision complex	Short-precision complex	CSYRK
Long-precision complex	Long-precision complex	ZSYRK
Short-precision complex	Short-precision real	CHERK
Long-precision complex	Long-precision real	ZHERK

Syntax

Fortran	CALL SSYRK | DSYRK | CSYRK | ZSYRK | CHERK | ZHERK (*uplo*, *trans*, *n*, *k*, *alpha*, *a*, *lda*, *beta*, *c*, *ldc*)
C and C++	ssyrk | dsyrk | csyrk | zsyrk | cherk | zherk (*uplo*, *trans*, *n*, *k*, *alpha*, *a*, *lda*, *beta*, *c*, *ldc*);
PL/I	CALL SSYRK | DSYRK | CSYRK | ZSYRK | CHERK | ZHERK (*uplo*, *trans*, *n*, *k*, *alpha*, *a*, *lda*, *beta*, *c*, *ldc*);

On Entry

uplo
indicates the storage mode used for matrix ***C***, where:

If *uplo* = 'U', ***C*** is stored in upper storage mode.

If *uplo* = 'L', ***C*** is stored in lower storage mode.

Specified as: a single character. It must be 'U' or 'L'.

trans
indicates the form of matrix ***A*** to use in the computation, where:

If *trans* = 'N', ***A*** is used, resulting in equation 1 or 3.

If *trans* = 'T', $\boldsymbol{A}^{T}$ is used, resulting in equation 2.

If *trans* = 'C', A^H is used, resulting in equation 4.

Specified as: a single character, where:

For SSYRK and DSYRK, it must be 'N', 'T', or 'C'.

For CSYRK and ZSYRK, it must be 'N' or 'T'.

For CHERK and ZHERK, it must be 'N' or 'C'.

n

is the order of matrix ***C***. Specified as: a fullword integer; $0 \le n \le ldc$ and:

If *trans* = 'N', then $n \le lda$.

k

has the following meaning, where:

If *trans* = 'N', it is the number of columns in matrix ***A***.

If *trans* = 'T' or 'C', it is the number of rows in matrix ***A***.

Specified as: a fullword integer; $k \ge 0$ and:

If *trans* = 'T' or 'C', then $k \le lda$.

alpha

is the scalar α. Specified as: a number of the data type indicated in Table 79 on page 437.

a

is the rectangular matrix ***A***, where:

If *trans* = 'N', ***A*** is *n* by *k*.

If *trans* = 'T' or 'C', ***A*** is *k* by *n*.

Note: No data should be moved to form A^T or A^H; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 79 on page 437, where:

If *trans* = 'N', its size must be *lda* by (at least) *k*.

If *trans* = 'T' or 'C', its size must be *lda* by (at least) *n*.

lda

is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and:

If *trans* = 'N', $lda \ge n$.

If *trans* = 'T' or 'C', $lda \ge k$.

beta

is the scalar β. Specified as: a number of the data type indicated in Table 79 on page 437.

c

is matrix ***C*** of order *n*, which is real symmetric, complex symmetric, or complex Hermitian, where:

If *uplo* = 'U', ***C*** is stored in upper storage mode.

If *uplo* = 'L', ***C*** is stored in lower storage mode.

Specified as: an *ldc* by (at least) *n* array, containing numbers of the data type indicated in Table 79 on page 437.

ldc
is the leading dimension of the array specified for *c*. Specified as: a fullword integer; *ldc* > 0 and *ldc* ≥ *n*.

On Return

c
is matrix ***C*** of order *n*, which is real symmetric, complex symmetric, or complex Hermitian, containing the results of the computation, where:

If *uplo* = 'U', ***C*** is stored in upper storage mode.

If *uplo* = 'L', ***C*** is stored in lower storage mode.

Returned as: an *ldc* by (at least) *n* array, containing numbers of the data type indicated in Table 79 on page 437.

Notes

1. These subroutines accept lowercase letters for the *uplo* and *trans* arguments.
2. For SSYRK and DSYRK, if you specify 'C' for the *trans* argument, it is interpreted as though you specified 'T'.
3. Matrices ***A*** and ***C*** must have no common elements; otherwise, results are unpredictable.
4. The imaginary parts of the diagonal elements of a complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, they are set to zero, except when β is one and α or *k* is zero, in which case no computation is performed.
5. For a description of how symmetric matrices are stored in upper and lower storage mode, see "Symmetric Matrix" on page 65. For a description of how complex Hermitian matrices are stored in upper and lower storage mode, see "Complex Hermitian Matrix" on page 70.

Function: These subroutines can perform the following rank-k updates. For SSYRK and DSYRK, matrix ***C*** is real symmetric. For CSYRK and ZSYRK, matrix ***C*** is complex symmetric. They perform:

1. $\boldsymbol{C} \leftarrow \alpha\boldsymbol{A}\boldsymbol{A}^{T}+\beta\boldsymbol{C}$
2. $\boldsymbol{C} \leftarrow \alpha\boldsymbol{A}^{T}\boldsymbol{A}+\beta\boldsymbol{C}$

For CHERK and ZHERK, matrix ***C*** is complex Hermitian. They perform:

3. $\boldsymbol{C} \leftarrow \alpha\boldsymbol{A}\boldsymbol{A}^{H}+\beta\boldsymbol{C}$
4. $\boldsymbol{C} \leftarrow \alpha\boldsymbol{A}^{H}\boldsymbol{A}+\beta\boldsymbol{C}$

where:

α and β are scalars.
A is a rectangular matrix, which is *n* by *k* for equations 1 and 3, and is *k* by *n* for equations 2 and 4.
C is a matrix of order *n* of the type indicated above, stored in upper or lower storage mode.

See references [32] and [38]. In the following two cases, no computation is performed:

- *n* is 0.

- β is one, and α is zero or *k* is zero.

Assuming the above conditions do not exist, if β is not one, and α is zero or *k* is zero, then β***C*** is returned.

Error Conditions

Resource Errors: Unable to allocate internal work area.

Computational Errors: None

Input-Argument Errors

1. *lda*, *ldc* ≤ 0
2. *ldc* < *n*
3. *k*, *n* < 0
4. *uplo* ≠ 'U' or 'L'
5. *trans* ≠ 'N', 'T', or 'C' for SSYRK and DSYRK
6. *trans* ≠ 'N' or 'T' for CSYRK and ZSYRK
7. *trans* ≠ 'N' or 'C' for CHERK and ZHERK
8. *trans* = 'N' and *lda* < *n*
9. *trans* = 'T' or 'C' and *lda* < *k*

Example 1: This example shows the computation $\boldsymbol{C} \leftarrow \alpha\boldsymbol{A}\boldsymbol{A}^{T}+\beta\boldsymbol{C}$, where ***A*** is an 8 by 2 real rectangular matrix, and ***C*** is a real symmetric matrix of order 8, stored in upper storage mode.

Call Statement and Input

```
            UPLO TRANS   N   K   ALPHA   A  LDA  BETA  C   LDC
             |     |     |   |     |     |   |    |    |    |
CALL SSYRK( 'U' , 'N' ,  8 , 2 ,  1.0  , A , 9 , 1.0 , C , 10 )
```

```
        ┌          ┐
        │ 0.0   8.0 │
        │ 1.0   9.0 │
        │ 2.0  10.0 │
        │ 3.0  11.0 │
A   =   │ 4.0  12.0 │
        │ 5.0  13.0 │
        │ 6.0  14.0 │
        │ 7.0  15.0 │
        │  .     .  │
        └          ┘
```

```
        ┌                                          ┐
        │ 0.0  1.0  3.0  6.0  10.0  15.0  21.0  28.0 │
        │  .   2.0  4.0  7.0  11.0  16.0  22.0  29.0 │
        │  .    .   5.0  8.0  12.0  17.0  23.0  30.0 │
        │  .    .    .   9.0  13.0  18.0  24.0  31.0 │
C   =   │  .    .    .    .   14.0  19.0  25.0  32.0 │
        │  .    .    .    .     .   20.0  26.0  33.0 │
        │  .    .    .    .     .     .   27.0  34.0 │
        │  .    .    .    .     .     .     .   35.0 │
        │  .    .    .    .     .     .     .     .  │
        │  .    .    .    .     .     .     .     .  │
        └                                          ┘
```

Output

```
        ⎡                                                                ⎤
        ⎢ 64.0  73.0   83.0   94.0  106.0  119.0  133.0  148.0 ⎥
        ⎢   .   84.0   96.0  109.0  123.0  138.0  154.0  171.0 ⎥
        ⎢   .     .   109.0  124.0  140.0  157.0  175.0  194.0 ⎥
        ⎢   .     .      .   139.0  157.0  176.0  196.0  217.0 ⎥
C    =  ⎢   .     .      .      .   174.0  195.0  217.0  240.0 ⎥
        ⎢   .     .      .      .      .   214.0  238.0  263.0 ⎥
        ⎢   .     .      .      .      .      .   259.0  286.0 ⎥
        ⎢   .     .      .      .      .      .      .   309.0 ⎥
        ⎢   .     .      .      .      .      .      .      .  ⎥
        ⎢   .     .      .      .      .      .      .      .  ⎥
        ⎣                                                                ⎦
```

Example 2: This example shows the computation $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^{T}\boldsymbol{A}+\beta \boldsymbol{C}$, where $\boldsymbol{A}$ is a 3 by 8 real rectangular matrix, and $\boldsymbol{C}$ is a real symmetric matrix of order 8, stored in lower storage mode.

Call Statement and Input

```
            UPLO TRANS   N   K   ALPHA   A  LDA  BETA  C  LDC
             |     |     |   |     |     |   |    |    |   |
CALL SSYRK( 'L' , 'T' ,  8 , 3 ,  1.0  , A , 4 , 1.0 , C , 8 )

        ⎡                                                    ⎤
        ⎢ 0.0  3.0  6.0   9.0  12.0  15.0  18.0  21.0 ⎥
A    =  ⎢ 1.0  4.0  7.0  10.0  13.0  16.0  19.0  22.0 ⎥
        ⎢ 2.0  5.0  8.0  11.0  14.0  17.0  20.0  23.0 ⎥
        ⎢  .    .    .     .     .     .     .     .  ⎥
        ⎣                                                    ⎦

        ⎡                                                      ⎤
        ⎢ 0.0    .     .     .     .     .     .     .  ⎥
        ⎢ 1.0   8.0    .     .     .     .     .     .  ⎥
        ⎢ 2.0   9.0  15.0    .     .     .     .     .  ⎥
C    =  ⎢ 3.0  10.0  16.0  21.0    .     .     .     .  ⎥
        ⎢ 4.0  11.0  17.0  22.0  26.0    .     .     .  ⎥
        ⎢ 5.0  12.0  18.0  23.0  27.0  30.0    .     .  ⎥
        ⎢ 6.0  13.0  19.0  24.0  28.0  31.0  33.0    .  ⎥
        ⎢ 7.0  14.0  20.0  25.0  29.0  32.0  34.0  35.0 ⎥
        ⎣                                                      ⎦
```

Output

```
        ⎡                                                                ⎤
        ⎢  5.0     .      .      .      .       .       .       .  ⎥
        ⎢ 15.0   58.0     .      .      .       .       .       .  ⎥
        ⎢ 25.0   95.0  164.0     .      .       .       .       .  ⎥
C    =  ⎢ 35.0  132.0  228.0  323.0     .       .       .       .  ⎥
        ⎢ 45.0  169.0  292.0  414.0  535.0      .       .       .  ⎥
        ⎢ 55.0  206.0  356.0  505.0  653.0   800.0      .       .  ⎥
        ⎢ 65.0  243.0  420.0  596.0  771.0   945.0  1118.0      .  ⎥
        ⎢ 75.0  280.0  484.0  687.0  889.0  1090.0  1290.0  1489.0 ⎥
        ⎣                                                                ⎦
```

Example 3: This example shows the computation $C \leftarrow \alpha AA^{T}+\beta C$, where A is a 3 by 5 complex rectangular matrix, and C is a complex symmetric matrix of order 3, stored in upper storage mode.

Call Statement and Input

```
            UPLO TRANS   N   K   ALPHA   A  LDA  BETA   C  LDC
             |     |     |   |     |     |   |    |     |   |
CALL CSYRK( 'U' , 'N' ,  3 , 5 , ALPHA , A , 3 , BETA , C , 4 )

ALPHA    =  (1.0, 1.0)
BETA     =  (1.0, 1.0)

        ┌                                                       ┐
        | (2.0, 0.0) (3.0, 2.0) (4.0, 1.0) (1.0, 7.0) (0.0, 0.0) |
A    =  | (3.0, 3.0) (8.0, 0.0) (2.0, 5.0) (2.0, 4.0) (1.0, 2.0) |
        | (1.0, 3.0) (2.0, 1.0) (6.0, 0.0) (3.0, 2.0) (2.0, 2.0) |
        └                                                       ┘

        ┌                          ┐
        | (2.0, 1.0) (1.0, 9.0) (4.0, 5.0) |
C    =  |     .      (3.0, 1.0) (6.0, 7.0) |
        |     .          .      (8.0, 1.0) |
        |     .          .          .      |
        └                          ┘
```

Output

```
        ┌                                    ┐
        | (-57.0, 13.0) (-63.0, 79.0) (-24.0,  70.0) |
C    =  |       .       (-28.0, 90.0) (-55.0, 103.0) |
        |       .             .        (13.0,  75.0) |
        |       .             .            .         |
        └                                    ┘
```

Example 4: This example shows the computation $C \leftarrow \alpha A^{H}A+\beta C$, where A is a 5 by 3 complex rectangular matrix, and C is a complex Hermitian matrix of order 3, stored in lower storage mode.

Note: The imaginary parts of the diagonal elements of a complex Hermitian matrix are assumed to be zero, so you do not have to set these values. On output, they are set to zero.

Call Statement and Input

```
            UPLO TRANS   N   K   ALPHA   A  LDA  BETA   C  LDC
             |     |     |   |     |     |   |    |     |   |
CALL CHERK( 'L' , 'C' ,  3 , 5 ,  1.0  , A , 5 , 1.0  , C , 4 )

        ┌                          ┐
        | (2.0, 0.0) (3.0, 2.0) (4.0, 1.0) |
        | (3.0, 3.0) (8.0, 0.0) (2.0, 5.0) |
A    =  | (1.0, 3.0) (2.0, 1.0) (6.0, 0.0) |
        | (3.0, 3.0) (8.0, 0.0) (2.0, 5.0) |
        | (1.0, 9.0) (3.0, 0.0) (6.0, 7.0) |
        └                          ┘
```

```
        ⎡                                    ⎤
        ⎢ (6.0,  . )       .           .     ⎥
C    =  ⎢ (3.0, 4.0)  (10.0,  . )      .     ⎥
        ⎢ (9.0, 1.0)  (12.0, 2.0)  (3.0, . ) ⎥
        ⎢     .            .           .     ⎥
        ⎣                                    ⎦
```

Output

```
        ⎡                                          ⎤
        ⎢ (138.0,  0.0)        .               .      ⎥
C    =  ⎢  (65.0, 80.0)  (165.0,   0.0)        .      ⎥
        ⎢ (134.0, 46.0)   (88.0, -88.0)  (199.0, 0.0) ⎥
        ⎢       .              .               .      ⎥
        ⎣                                          ⎦
```

SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, and ZHER2K—Rank-2K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix

These subroutines compute one of the following rank-2k updates, where matrix ***C*** is stored in upper or lower storage mode. SSYR2K, DSYR2K, CSYR2K, and ZSYR2K use the scalars α and β, real or complex matrices ***A*** and ***B*** or their transposes, and real or complex symmetric matrix ***C*** to compute:

1. $\boldsymbol{C} \leftarrow \alpha\boldsymbol{A}\boldsymbol{B}^{\mathrm{T}}+\alpha\boldsymbol{B}\boldsymbol{A}^{\mathrm{T}}+\beta\boldsymbol{C}$
2. $\boldsymbol{C} \leftarrow \alpha\boldsymbol{A}^{\mathrm{T}}\boldsymbol{B}+\alpha\boldsymbol{B}^{\mathrm{T}}\boldsymbol{A}+\beta\boldsymbol{C}$

CHER2K and ZHER2K use the scalars α and β, complex matrices ***A*** and ***B*** or their complex conjugate transposes, and complex Hermitian matrix ***C*** to compute:

3. $\boldsymbol{C} \leftarrow \alpha\boldsymbol{A}\boldsymbol{B}^{\mathrm{H}} + \overline{\alpha}\boldsymbol{B}\boldsymbol{A}^{\mathrm{H}} + \beta\boldsymbol{C}$

4. $\boldsymbol{C} \leftarrow \alpha\boldsymbol{A}^{\mathrm{H}}\boldsymbol{B} + \overline{\alpha}\boldsymbol{B}^{\mathrm{H}}\boldsymbol{A} + \beta\boldsymbol{C}$

Table 80. Data Types

A, ***B***, ***C***, α	β	**Subprogram**
Short-precision real	Short-precision real	SSYR2K
Long-precision real	Long-precision real	DSYR2K
Short-precision complex	Short-precision complex	CSYR2K
Long-precision complex	Long-precision complex	ZSYR2K
Short-precision complex	Short-precision real	CHER2K
Long-precision complex	Long-precision real	ZHER2K

Syntax

Fortran	CALL SSYR2K | DSYR2K | CSYR2K | ZSYR2K | CHER2K | ZHER2K (*uplo*, *trans*, *n*, *k*, *alpha*, *a*, *lda*, *b*, *ldb*, *beta*, *c*, *ldc*)
C and C++	ssyr2k | dsyr2k | csyr2k | zsyr2k | cher2k | zher2k (*uplo*, *trans*, *n*, *k*, *alpha*, *a*, *lda*, *b*, *ldb*, *beta*, *c*, *ldc*);
PL/I	CALL SSYR2K | DSYR2K | CSYR2K | ZSYR2K | CHER2K | ZHER2K (*uplo*, *trans*, *n*, *k*, *alpha*, *a*, *lda*, *b*, *ldb*, *beta*, *c*, *ldc*);

On Entry

uplo

indicates the storage mode used for matrix ***C***, where:

If *uplo* = 'U', ***C*** is stored in upper storage mode.

If *uplo* = 'L', ***C*** is stored in lower storage mode.

Specified as: a single character. It must be 'U' or 'L'.

trans
indicates the form of matrices ***A*** and ***B*** to use in the computation, where:

If *trans* = 'N', ***A*** and ***B*** are used, resulting in equation 1 or 3.

If *trans* = 'T', $\boldsymbol{A}^{T}$ and $\boldsymbol{B}^{T}$ are used, resulting in equation 2.

If *trans* = 'C', $\boldsymbol{A}^{H}$ and $\boldsymbol{B}^{H}$ are used, resulting in equation 4.

Specified as: a single character, where:

For SSYR2K and DSYR2K, it must be 'N', 'T', or 'C'.

For CSYR2K and ZSYR2K, it must be 'N' or 'T'.

For CHER2K and ZHER2K, it must be 'N' or 'C'.

n
is the order of matrix ***C***. Specified as: a fullword integer; $0 \le n \le ldc$ and:

If *trans* = 'N', then $n \le lda$ and $n \le ldb$.

k
has the following meaning, where:

If *trans* = 'N', it is the number of columns in matrices ***A*** and ***B***.

If *trans* = 'T' or 'C', it is the number of rows in matrices ***A*** and ***B***.

Specified as: a fullword integer; $k \ge 0$ and:

If *trans* = 'T' or 'C', then $k \le lda$ and $k \le ldb$.

alpha
is the scalar α. Specified as: a number of the data type indicated in Table 80 on page 444.

a
is the rectangular matrix ***A***, where:

If *trans* = 'N', ***A*** is *n* by *k*.

If *trans* = 'T' or 'C', ***A*** is *k* by *n*.

Note: No data should be moved to form $\boldsymbol{A}^{T}$ or $\boldsymbol{A}^{H}$; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 80 on page 444, where:

If *trans* = 'N', its size must be *lda* by (at least) *k*.

If *trans* = 'T' or 'C', its size must be *lda* by (at least) *n*.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and:

If *trans* = 'N', $lda \ge n$.

If *trans* = 'T' or 'C', $lda \ge k$.

b
is the rectangular matrix ***B***, where:

If *trans* = 'N', ***B*** is *n* by *k*.

If *trans* = 'T' or 'C', ***B*** is *k* by *n*.

Note: No data should be moved to form B^T or B^H; that is, the matrix ***B*** should always be stored in its untransposed form.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 80 on page 444, where:

If *trans* = 'N', its size must be *ldb* by (at least) *k*.

If *trans* = 'T' or 'C', its size must be *ldb* by (at least) *n*.

ldb

is the leading dimension of the array specified for *b*. Specified as: a fullword integer; *ldb* > 0 and:

If *trans* = 'N', *ldb* ≥ *n*.

If *trans* = 'T' or 'C', *ldb* ≥ *k*.

beta

is the scalar β. Specified as: a number of the data type indicated in Table 80 on page 444.

c

is matrix ***C*** of order *n*, which is real symmetric, complex symmetric, or complex Hermitian, where:

If *uplo* = 'U', ***C*** is stored in upper storage mode.

If *uplo* = 'L', ***C*** is stored in lower storage mode.

Specified as: an *ldc* by (at least) *n* array, containing numbers of the data type indicated in Table 80 on page 444.

ldc

is the leading dimension of the array specified for *c*. Specified as: a fullword integer; *ldc* > 0 and *ldc* ≥ *n*.

On Return

c

is matrix ***C*** of order *n*, which is real symmetric, complex symmetric, or complex Hermitian, containing the results of the computation, where:

If *uplo* = 'U', ***C*** is stored in upper storage mode.

If *uplo* = 'L', ***C*** is stored in lower storage mode.

Returned as: an *ldc* by (at least) *n* array, containing numbers of the data type indicated in Table 80 on page 444.

Notes

1. These subroutines accept lowercase letters for the *uplo* and *trans* arguments.
2. For SSYR2K and DSYR2K, if you specify 'C' for the *trans* argument, it is interpreted as though you specified 'T'.
3. Matrices ***A*** and ***B*** must have no common elements with matrix ***C***; otherwise, results are unpredictable.
4. The imaginary parts of the diagonal elements of a complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, they are set to zero, except when β is one and α or *k* is zero, in which case no computation is performed.

5. For a description of how symmetric matrices are stored in upper and lower storage mode, see "Symmetric Matrix" on page 65. For a description of how complex Hermitian matrices are stored in upper and lower storage mode, see "Complex Hermitian Matrix" on page 70.

Function: These subroutines can perform the following rank-2k updates. For SSYR2K and DSYR2K, matrix ***C*** is real symmetric. For CSYR2K and ZSYR2K, matrix ***C*** is complex symmetric. They perform:

1. $\boldsymbol{C} \leftarrow \alpha\boldsymbol{A}\boldsymbol{B}^{T} + \alpha\boldsymbol{B}\boldsymbol{A}^{T} + \beta\boldsymbol{C}$
2. $\boldsymbol{C} \leftarrow \alpha\boldsymbol{A}^{T}\boldsymbol{B} + \alpha\boldsymbol{B}^{T}\boldsymbol{A} + \beta\boldsymbol{C}$

For CHER2K and ZHER2K, matrix ***C*** is complex Hermitian. They perform:

3. $\boldsymbol{C} \leftarrow \alpha\boldsymbol{A}\boldsymbol{B}^{H} + \overline{\alpha}\boldsymbol{B}\boldsymbol{A}^{H} + \beta\boldsymbol{C}$

4. $\boldsymbol{C} \leftarrow \alpha\boldsymbol{A}^{H}\boldsymbol{B} + \overline{\alpha}\boldsymbol{B}^{H}\boldsymbol{A} + \beta\boldsymbol{C}$

where:

α and β are scalars.
A and ***B*** are rectangular matrices, which are *n* by *k* for equations 1 and 3, and are *k* by *n* for equations 2 and 4.
C is a matrix of order *n* of the type indicated above, stored in upper or lower storage mode.

See references [32], [38], and [66]. In the following two cases, no computation is performed:

- *n* is 0.
- β is one, and α is zero or *k* is zero.

Assuming the above conditions do not exist, if β is not one, and α is zero or *k* is zero, then $\beta\boldsymbol{C}$ is returned.

Error Conditions

Resource Errors: Unable to allocate internal work area.

Computational Errors: None

Input-Argument Errors

1. *lda*, *ldb*, *ldc* ≤ 0
2. *ldc* < *n*
3. *k*, *n* < 0
4. *uplo* ≠ 'U' or 'L'
5. *trans* ≠ 'N', 'T', or 'C' for SSYR2K and DSYR2K
6. *trans* ≠ 'N' or 'T' for CSYR2K and ZSYR2K
7. *trans* ≠ 'N' or 'C' for CHER2K and ZHER2K
8. *trans* = 'N' and *lda* < *n*
9. *trans* = 'T' or 'C' and *lda* < *k*
10. *trans* = 'N' and *ldb* < *n*
11. *trans* = 'T' or 'C' and *ldb* < *k*

Example 1: This example shows the computation $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}\boldsymbol{B}^{\mathrm{T}}+\alpha \boldsymbol{B}\boldsymbol{A}^{\mathrm{T}}+\beta \boldsymbol{C}$, where $\boldsymbol{A}$ and $\boldsymbol{B}$ are 8 by 2 real rectangular matrices, and $\boldsymbol{C}$ is a real symmetric matrix of order 8, stored in upper storage mode.

Call Statement and Input

```
             UPLO TRANS   N   K   ALPHA   A  LDA  B  LDB  BETA  C   LDC
              |     |     |   |     |     |   |   |   |    |    |    |
CALL SSYR2K( 'U' , 'N' ,  8 , 2 ,  1.0  , A , 9 , B , 8 , 1.0 , C , 10 )
```

```
        ⎡          ⎤
        ⎢ 0.0   8.0 ⎥
        ⎢ 1.0   9.0 ⎥
        ⎢ 2.0  10.0 ⎥
        ⎢ 3.0  11.0 ⎥
A    =  ⎢ 4.0  12.0 ⎥
        ⎢ 5.0  13.0 ⎥
        ⎢ 6.0  14.0 ⎥
        ⎢ 7.0  15.0 ⎥
        ⎢  .     .  ⎥
        ⎣          ⎦
```

```
        ⎡          ⎤
        ⎢ 15.0  7.0 ⎥
        ⎢ 14.0  6.0 ⎥
        ⎢ 13.0  5.0 ⎥
B    =  ⎢ 12.0  4.0 ⎥
        ⎢ 11.0  3.0 ⎥
        ⎢ 10.0  2.0 ⎥
        ⎢  9.0  1.0 ⎥
        ⎢  8.0  0.0 ⎥
        ⎣          ⎦
```

```
        ⎡                                                  ⎤
        ⎢ 0.0  1.0  3.0  6.0  10.0  15.0  21.0  28.0 ⎥
        ⎢  .   2.0  4.0  7.0  11.0  16.0  22.0  29.0 ⎥
        ⎢  .    .   5.0  8.0  12.0  17.0  23.0  30.0 ⎥
        ⎢  .    .    .   9.0  13.0  18.0  24.0  31.0 ⎥
C    =  ⎢  .    .    .    .   14.0  19.0  25.0  32.0 ⎥
        ⎢  .    .    .    .     .   20.0  26.0  33.0 ⎥
        ⎢  .    .    .    .     .     .   27.0  34.0 ⎥
        ⎢  .    .    .    .     .     .     .   35.0 ⎥
        ⎢  .    .    .    .     .     .     .     .  ⎥
        ⎢  .    .    .    .     .     .     .     .  ⎥
        ⎣                                                  ⎦
```

Output

```
        ⎡                                                                ⎤
        ⎢ 112.0  127.0  143.0  160.0  178.0  197.0  217.0  238.0 ⎥
        ⎢    .   138.0  150.0  163.0  177.0  192.0  208.0  225.0 ⎥
        ⎢    .      .   157.0  166.0  176.0  187.0  199.0  212.0 ⎥
        ⎢    .      .      .   169.0  175.0  182.0  190.0  199.0 ⎥
C    =  ⎢    .      .      .      .   174.0  177.0  181.0  186.0 ⎥
        ⎢    .      .      .      .      .   172.0  172.0  173.0 ⎥
        ⎢    .      .      .      .      .      .   163.0  160.0 ⎥
        ⎢    .      .      .      .      .      .      .   147.0 ⎥
        ⎢    .      .      .      .      .      .      .      .  ⎥
        ⎢    .      .      .      .      .      .      .      .  ⎥
        ⎣                                                                ⎦
```

Example 2: This example shows the computation $\boldsymbol{C} \leftarrow \alpha \boldsymbol{A}^{\mathrm{T}}\boldsymbol{B}+\alpha \boldsymbol{B}^{\mathrm{T}}\boldsymbol{A}+\beta \boldsymbol{C}$, where $\boldsymbol{A}$ and $\boldsymbol{B}$ are 3 by 8 real rectangular matrices, and $\boldsymbol{C}$ is a real symmetric matrix of order 8, stored in lower storage mode.

Call Statement and Input

```
             UPLO TRANS   N   K   ALPHA   A  LDA  B  LDB  BETA  C  LDC
              |     |     |   |     |     |   |   |   |    |    |   |
CALL SSYR2K( 'L' , 'T' ,  8 , 3 ,  1.0  , A , 4 , B , 5 , 1.0 , C , 8 )

        ⎡                                                    ⎤
        ⎢ 0.0  3.0  6.0   9.0  12.0  15.0  18.0  21.0 ⎥
A    =  ⎢ 1.0  4.0  7.0  10.0  13.0  16.0  19.0  22.0 ⎥
        ⎢ 2.0  5.0  8.0  11.0  14.0  17.0  20.0  23.0 ⎥
        ⎢  .    .    .     .     .     .     .     .  ⎥
        ⎣                                                    ⎦

        ⎡                                                ⎤
        ⎢ 1.0  2.0  3.0  4.0  5.0  6.0  7.0   8.0 ⎥
        ⎢ 2.0  3.0  4.0  5.0  6.0  7.0  8.0   9.0 ⎥
B    =  ⎢ 3.0  4.0  5.0  6.0  7.0  8.0  9.0  10.0 ⎥
        ⎢  .    .    .    .    .    .    .     .  ⎥
        ⎢  .    .    .    .    .    .    .     .  ⎥
        ⎣                                                ⎦

        ⎡                                                    ⎤
        ⎢ 0.0    .     .     .     .     .     .     .  ⎥
        ⎢ 1.0   8.0    .     .     .     .     .     .  ⎥
        ⎢ 2.0   9.0  15.0    .     .     .     .     .  ⎥
C    =  ⎢ 3.0  10.0  16.0  21.0    .     .     .     .  ⎥
        ⎢ 4.0  11.0  17.0  22.0  26.0    .     .     .  ⎥
        ⎢ 5.0  12.0  18.0  23.0  27.0  30.0    .     .  ⎥
        ⎢ 6.0  13.0  19.0  24.0  28.0  31.0  33.0    .  ⎥
        ⎢ 7.0  14.0  20.0  25.0  29.0  32.0  34.0  35.0 ⎥
        ⎣                                                    ⎦
```

Output

```
        ⎡                                                        ⎤
        ⎢  16.0    .     .     .     .     .      .      .      ⎥
        ⎢  38.0  84.0    .     .     .     .      .      .      ⎥
        ⎢  60.0 124.0 187.0    .     .     .      .      .      ⎥
C    =  ⎢  82.0 164.0 245.0 325.0    .     .      .      .      ⎥
        ⎢ 104.0 204.0 303.0 401.0 498.0    .      .      .      ⎥
        ⎢ 126.0 244.0 361.0 477.0 592.0 706.0     .      .      ⎥
        ⎢ 148.0 284.0 419.0 553.0 686.0 818.0  949.0     .      ⎥
        ⎢ 170.0 324.0 477.0 629.0 780.0 930.0 1079.0 1227.0     ⎥
        ⎣                                                        ⎦
```

Example 3: This example shows the computation $\boldsymbol{C} \leftarrow \alpha\boldsymbol{A}\boldsymbol{B}^{T}+\alpha\boldsymbol{B}\boldsymbol{A}^{T}+\beta\boldsymbol{C}$, where $\boldsymbol{A}$ and $\boldsymbol{B}$ are 3 by 5 complex rectangular matrices, and $\boldsymbol{C}$ is a complex symmetric matrix of order 3, stored in lower storage mode.

Call Statement and Input

```
             UPLO TRANS   N   K   ALPHA   A  LDA  B  LDB  BETA   C  LDC
              |     |     |   |     |     |   |   |   |    |     |   |
CALL CSYR2K( 'L' , 'N' ,  3 , 5 , ALPHA , A , 3 , B , 3 , BETA , C , 4 )

ALPHA    =  (1.0, 1.0)

BETA     =  (1.0, 1.0)

        ⎡                                                        ⎤
        ⎢ (2.0, 5.0) (3.0, 2.0) (4.0, 1.0) (1.0, 7.0) (0.0, 0.0) ⎥
A    =  ⎢ (3.0, 3.0) (8.0, 5.0) (2.0, 5.0) (2.0, 4.0) (1.0, 2.0) ⎥
        ⎢ (1.0, 3.0) (2.0, 1.0) (6.0, 5.0) (3.0, 2.0) (2.0, 2.0) ⎥
        ⎣                                                        ⎦

        ⎡                                                        ⎤
        ⎢ (1.0, 5.0) (6.0, 2.0) (3.0, 1.0) (2.0, 0.0) (1.0, 0.0) ⎥
B    =  ⎢ (2.0, 4.0) (7.0, 5.0) (2.0, 5.0) (2.0, 4.0) (0.0, 0.0) ⎥
        ⎢ (3.0, 5.0) (8.0, 1.0) (1.0, 5.0) (1.0, 0.0) (1.0, 1.0) ⎥
        ⎣                                                        ⎦

        ⎡                                   ⎤
        ⎢ (2.0, 3.0)     .          .       ⎥
C    =  ⎢ (1.0, 9.0) (3.0, 3.0)     .       ⎥
        ⎢ (4.0, 5.0) (6.0, 7.0) (8.0, 3.0) ⎥
        ⎢     .          .          .       ⎥
        ⎣                                   ⎦
```

Output

```
        ⎡                                                   ⎤
        ⎢ (-101.0, 121.0)        .               .          ⎥
C    =  ⎢ (-182.0, 192.0) (-274.0, 248.0)        .          ⎥
        ⎢  (-98.0, 146.0) (-163.0, 205.0) (-151.0, 115.0)   ⎥
        ⎢        .               .               .          ⎥
        ⎣                                                   ⎦
```

Example 4: This example shows the computation:

$$C \leftarrow \alpha A^{H} B + \overline{\alpha} B^{H} A + \beta C$$

where ***A*** and ***B*** are 5 by 3 complex rectangular matrices, and ***C*** is a complex Hermitian matrix of order 3, stored in upper storage mode.

Note: The imaginary parts of the diagonal elements of a complex Hermitian matrix are assumed to be zero, so you do not have to set these values. On output, they are set to zero.

Call Statement and Input

```
             UPLO TRANS   N   K   ALPHA   A  LDA  B  LDB  BETA   C  LDC
              |    |     |   |     |     |   |   |   |    |     |   |
CALL CHER2K( 'U' , 'C' ,  3 , 5 , ALPHA , A , 5 , B , 5 , 1.0  , C , 4 )

ALPHA    =  (1.0, 1.0)

         ┌                                  ┐
         | (2.0, 0.0) (3.0, 2.0) (4.0, 1.0) |
         | (3.0, 3.0) (8.0, 0.0) (2.0, 5.0) |
A    =   | (1.0, 3.0) (2.0, 1.0) (6.0, 0.0) |
         | (3.0, 3.0) (8.0, 0.0) (2.0, 5.0) |
         | (1.0, 9.0) (3.0, 0.0) (6.0, 7.0) |
         └                                  ┘

         ┌                                  ┐
         | (4.0, 5.0) (6.0, 7.0) (8.0, 0.0) |
         | (1.0, 9.0) (3.0, 0.0) (6.0, 7.0) |
B    =   | (3.0, 3.0) (8.0, 0.0) (2.0, 5.0) |
         | (1.0, 3.0) (2.0, 1.0) (6.0, 0.0) |
         | (2.0, 0.0) (3.0, 2.0) (4.0, 1.0) |
         └                                  ┘

         ┌                                  ┐
         | (6.0,  . )  (3.0, 4.0)  (9.0, 1.0) |
C    =   |     .      (10.0,  . ) (12.0, 2.0) |
         |     .           .       (3.0,  . ) |
         |     .           .           .      |
         └                                  ┘
```

Output

```
         ┌                                      ┐
         | (102.0, 0.0)  (56.0, -143.0) (244.0, -96.0) |
C    =   |       .      (174.0,    0.0) (238.0,  78.0) |
         |       .            .         (363.0,   0.0) |
         |       .            .              .         |
         └                                      ┘
```

SGETMI, DGETMI, CGETMI, and ZGETMI—General Matrix Transpose (In-Place)

These subroutines transpose an n by n matrix $\boldsymbol{A}$ in place—that is, in matrix $\boldsymbol{A}$:

$$\boldsymbol{A} \leftarrow \boldsymbol{A}^{\mathrm{T}}$$

Table 81. Data Types

A	**Subroutine**
Short-precision real	SGETMI
Long-precision real	DGETMI
Short-precision complex	CGETMI
Long-precision complex	ZGETMI

Syntax

Fortran	CALL SGETMI | DGETMI | CGETMI | ZGETMI (*a*, *lda*, *n*)
C and C++	sgetmi | dgetmi | cgetmi | zgetmi (*a*, *lda*, *n*);
PL/I	CALL SGETMI | DGETMI | CGETMI | ZGETMI (*a*, *lda*, *n*);

On Entry

a
is the matrix $\boldsymbol{A}$ having n rows and n columns. Specified as: an *lda* by (at least) n array, containing numbers of the data type indicated in Table 81.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and $lda \geq n$.

n
is the number of rows and columns in matrix $\boldsymbol{A}$. Specified as: a fullword integer; $n \geq 0$.

On Return

a
is the n by n matrix $\boldsymbol{A}^{\mathrm{T}}$, containing the results of the matrix transpose operation Returned as: an *lda* by (at least) n array, containing numbers of the data type indicated in Table 81.

Notes

1. To achieve optimal performance in these subroutines, specify an even value for *lda*. An odd value may degrade performance.
2. To achieve optimal performance in CGETMI, align the array specified for *a* on a doubleword boundary.

Function: Matrix $\boldsymbol{A}$ is transposed in place; that is, the n rows and n columns in matrix $\boldsymbol{A}$ are exchanged. For matrix $\boldsymbol{A}$ with elements a_{ij}, where $i, j = 1, n$, the in-place transpose is expressed as $a_{ji} = a_{ij}$ for $i, j = 1, n$.

For the following input matrix $\boldsymbol{A}$:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{n1} & \dots & a_{nn} \end{bmatrix}$$

the in-place matrix transpose operation $\boldsymbol{A} \leftarrow \boldsymbol{A}^{\mathrm{T}}$ is expressed as:

$$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{n1} & \dots & a_{nn} \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} & \dots & a_{n1} \\ . & & . \\ . & & . \\ . & & . \\ a_{1n} & \dots & a_{nn} \end{bmatrix}$$

If n is 0, no computation is performed.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. $n < 0$ or $n > lda$
2. $lda \leq 0$

Example: This example shows an in-place matrix transpose of matrix $\boldsymbol{A}$ having 5 rows and 5 columns.

Call Statement and Input

```
                A     LDA   N
                |      |    |
CALL SGETMI( A(2,3) , 10 , 5 )
```

```
        [                                  ]
        | .   .   .     .     .     .     . |
        | .   .  1.0   6.0  11.0  16.0  21.0 |
        | .   .  2.0   7.0  12.0  17.0  22.0 |
        | .   .  3.0   8.0  13.0  18.0  23.0 |
A    =  | .   .  4.0   9.0  14.0  19.0  24.0 |
        | .   .  5.0  10.0  15.0  20.0  25.0 |
        | .   .   .     .     .     .     . |
        | .   .   .     .     .     .     . |
        | .   .   .     .     .     .     . |
        | .   .   .     .     .     .     . |
        [                                  ]
```

Output

$$A = \begin{bmatrix} . & . & . & . & . & . & . \\ . & . & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 \\ . & . & 6.0 & 7.0 & 8.0 & 9.0 & 10.0 \\ . & . & 11.0 & 12.0 & 13.0 & 14.0 & 15.0 \\ . & . & 16.0 & 17.0 & 18.0 & 19.0 & 20.0 \\ . & . & 21.0 & 22.0 & 23.0 & 24.0 & 25.0 \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \end{bmatrix}$$

SGETMO, DGETMO, CGETMO, and ZGETMO—General Matrix Transpose (Out-of-Place)

These subroutines transpose an *m* by *n* matrix ***A*** out of place, returning the result in matrix ***B***:

$$\boldsymbol{B} \leftarrow \boldsymbol{A}^{T}$$

Table 82. Data Types

A, ***B***	**Subroutine**
Short-precision real	SGETMO
Long-precision real	DGETMO
Short-precision complex	CGETMO
Long-precision complex	ZGETMO

Syntax

Fortran	CALL SGETMO | DGETMO | CGETMO | ZGETMO (*a*, *lda*, *m*, *n*, *b*, *ldb*)
C and C++	sgetmo | dgetmo | cgetmo | zgetmo (*a*, *lda*, *m*, *n*, *b*, *ldb*);
PL/I	CALL SGETMO | DGETMO | CGETMO | ZGETMO (*a*, *lda*, *m*, *n*, *b*, *ldb*);

On Entry

a
is the matrix ***A*** having *m* rows and *n* columns. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 82.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and $lda \geq m$.

m
is the number of rows in matrix ***A*** and the number of columns in matrix ***B***. Specified as: a fullword integer; $m \geq 0$.

n
is the number of columns in matrix ***A*** and the number of rows in matrix ***B***. Specified as: a fullword integer; $n \geq 0$.

b
See On Return.

ldb
is the leading dimension of the array specified for *b*. Specified as: a fullword integer; $ldb > 0$ and $ldb \geq n$.

On Return

b
is the matrix ***B*** having *n* rows and *m* columns, containing the results of the matrix transpose operation, $\boldsymbol{A}^{T}$. Returned as: an *ldb* by (at least) *m* array, containing numbers of the data type indicated in Table 82.

Notes

1. The matrix ***B*** must have no common elements with matrix ***A***; otherwise, results are unpredictable. See "Concepts" on page 55.
2. To achieve optimal performance in CGETMO, align the arrays specified for *a* and *b* on doubleword boundaries.

Function: Matrix ***A*** is transposed out of place; that is, the *m* rows and *n* columns in matrix ***A*** are stored in *n* rows and *m* columns of matrix ***B***. For matrix ***A*** with elements a_{ij}, where $i = 1, m$ and $j = 1, n$, the out-of-place transpose is expressed as $b_{ji} = a_{ij}$ for $i = 1, m$ and $j = 1, n$.

For the following input matrix ***A***:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & \ldots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{m1} & \ldots & a_{mn} \end{bmatrix}$$

the out-of-place matrix transpose operation $\boldsymbol{B} \leftarrow \boldsymbol{A}^{\mathrm{T}}$ is expressed as:

$$\begin{bmatrix} b_{11} & \ldots & b_{1m} \\ . & & . \\ . & & . \\ . & & . \\ b_{n1} & \ldots & b_{nm} \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} & \ldots & a_{m1} \\ . & & . \\ . & & . \\ . & & . \\ a_{1n} & \ldots & a_{mn} \end{bmatrix}$$

If *m* or *n* is 0, no computation is performed.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. $m < 0$ or $m > lda$
2. $n < 0$ or $n > ldb$
3. $lda \leq 0$
4. $ldb \leq 0$

Example 1: This example shows an out-of-place matrix transpose of matrix ***A***, having 5 rows and 4 columns, with the result going into matrix ***B***.

Call Statement and Input

```
                A     LDA   M   N     B     LDB
                |      |    |   |     |      |
CALL SGETMO( A(2,3) , 10 , 5 , 4 , B(2,2) , 6 )

        ⎡                                   ⎤
        | .  .   .     .     .     .   . |
        | .  .  1.0   6.0  11.0  16.0  . |
        | .  .  2.0   7.0  12.0  17.0  . |
        | .  .  3.0   8.0  13.0  18.0  . |
A    =  | .  .  4.0   9.0  14.0  19.0  . |
        | .  .  5.0  10.0  15.0  20.0  . |
        | .  .   .     .     .     .   . |
        | .  .   .     .     .     .   . |
        | .  .   .     .     .     .   . |
        | .  .   .     .     .     .   . |
        ⎣                                   ⎦
```

Output

```
        ⎡                                     ⎤
        | .    .     .     .     .     .   . |
        | .   1.0   2.0   3.0   4.0   5.0  . |
B    =  | .   6.0   7.0   8.0   9.0  10.0  . |
        | .  11.0  12.0  13.0  14.0  15.0  . |
        | .  16.0  17.0  18.0  19.0  20.0  . |
        | .    .     .     .     .     .   . |
        ⎣                                     ⎦
```

Example 2: This example uses the same input matrix ***A*** as in Example 1 to show that transposes can be achieved in the same array as long as the input and output data do not overlap. On output, the input data is not overwritten in the array.

Call Statement and Input

```
                A     LDA   M   N     B     LDB
                |      |    |   |     |      |
CALL SGETMO( A(2,3) , 10 , 5 , 4 , A(7,1) , 10 )

        ⎡                                       ⎤
        |   .     .     .     .     .     .   . |
        |   .     .    1.0   6.0  11.0  16.0  . |
        |   .     .    2.0   7.0  12.0  17.0  . |
        |   .     .    3.0   8.0  13.0  18.0  . |
A    =  |   .     .    4.0   9.0  14.0  19.0  . |
        |   .     .    5.0  10.0  15.0  20.0  . |
        |  1.0   2.0   3.0   4.0   5.0    .   . |
        |  6.0   7.0   8.0   9.0  10.0    .   . |
        | 11.0  12.0  13.0  14.0  15.0    .   . |
        | 16.0  17.0  18.0  19.0  20.0    .   . |
        ⎣                                       ⎦
```

Chapter 10. Linear Algebraic Equations

The linear algebraic equation subroutines, provided in four areas, are described in this chapter.

Overview of the Linear Algebraic Equation Subroutines

This section describes the subroutines in each of the four linear algebraic equation areas:

- Dense linear algebraic equations (Table 83)
- Banded linear algebraic equations (Table 84)
- Sparse linear algebraic equations (Table 85)
- Linear least squares (Table 86)

Note: Some of the linear algebraic equations were designed in accordance with the Level 2 BLAS, Level 3 BLAS, and LAPACK de facto standard. If these subprograms do not comply with the standard as approved, IBM will consider updating them to do so. If IBM updates these subprograms, the updates could require modifications of the calling application program. For details on the Level 2 and 3 BLAS, see references [32] and [34]. For details on the LAPACK routines, see reference [8].

Dense Linear Algebraic Equation Subroutines

The dense linear algebraic equation subroutines provide solutions to linear systems of equations for both real and complex general matrices and their transposes, positive definite real symmetric and complex Hermitian matrices, and triangular matrices. Some of these subroutines correspond to the Level 2 BLAS, Level 3 BLAS, and LAPACK routines described in references [32], [34], and [8].

Table 83 (Page 1 of 2). List of Dense Linear Algebraic Equation Subroutines

Descriptive Name	Short-Precision Subroutine	Long-Precision Subroutine	Page
General Matrix Factorization	SGEF CGEF SGETRF[Δ] CGETRF[Δ]	DGEF ZGEF DGETRF[Δ] ZGETRF[Δ] DGEFP[§]	468 481
General Matrix, Its Transpose, or Its Conjugate Transpose Solve	SGES CGES	DGES ZGES	471
General Matrix, Its Transpose, or Its Conjugate Transpose Multiple Right-Hand Side Solve	SGESM CGESM SGETRS[Δ] CGETRS[Δ]	DGESM ZGESM DGETRS[Δ] ZGETRS[Δ]	475 485
General Matrix Factorization, Condition Number Reciprocal, and Determinant	SGEFCD	DGEFCD	490

© Copyright IBM Corp. 1997

Table 83 (Page 2 of 2). List of Dense Linear Algebraic Equation Subroutines

Descriptive Name	Short-Precision Subroutine	Long-Precision Subroutine	Page
Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization	SPPF SPOF CPOF	DPPF DPOF ZPOF DPPFP§	494
Positive Definite Real Symmetric Matrix Solve	SPPS	DPPS	502
Positive Definite Real Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve	SPOSM CPOSM	DPOSM ZPOSM	505
Positive Definite Real Symmetric Matrix Factorization, Condition Number Reciprocal, and Determinant	SPPFCD SPOFCD	DPPFCD DPOFCD	510
General Matrix Inverse, Condition Number Reciprocal, and Determinant	SGEICD	DGEICD	516
Positive Definite Real Symmetric Matrix Inverse, Condition Number Reciprocal, and Determinant	SPPICD SPOICD	DPPICD DPOICD	521
Solution of a Triangular System of Equations with a Single Right-Hand Side	STRSV◂ CTRSV◂ STPSV◂ CTPSV◂	DTRSV◂ ZTRSV◂ DTPSV◂ ZTPSV◂	528
Solution of Triangular Systems of Equations with Multiple Right-Hand Sides	STRSM♦ CTRSM♦	DTRSM♦ ZTRSM♦	534
Triangular Matrix Inverse	STRI STPI	DTRI DTPI	542

◂ Level 2 BLAS

♦ Level 3 BLAS

Δ LAPACK

§ This subroutine is provided only for migration from earlier releases of ESSL and is not intended for use is new programs. Documentation for this subroutine is no longer provided.

Banded Linear Algebraic Equation Subroutines

The banded linear algebraic equation subroutines provide solutions to linear systems of equations for real general band matrices, real positive definite symmetric band matrices, real or complex general tridiagonal matrices, real positive definite symmetric tridiagonal matrices, and real or complex triangular band matrices.

Table 84 (Page 1 of 2). List of Banded Linear Algebraic Equation Subroutines

Descriptive Name	Short-Precision Subroutine	Long-Precision Subroutine	Page
General Band Matrix Factorization	SGBF	DGBF	548

Table 84 (Page 2 of 2). List of Banded Linear Algebraic Equation Subroutines

Descriptive Name	Short-Precision Subroutine	Long-Precision Subroutine	Page
General Band Matrix Solve	SGBS	DGBS	552
Positive Definite Symmetric Band Matrix Factorization	SPBF SPBCHF	DPBF DPBCHF	555
Positive Definite Symmetric Band Matrix Solve	SPBS SPBCHS	DPBS DPBCHS	559
General Tridiagonal Matrix Factorization	SGTF	DGTF	562
General Tridiagonal Matrix Solve	SGTS	DGTS	565
General Tridiagonal Matrix Combined Factorization and Solve with No Pivoting	SGTNP CGTNP	DGTNP ZGTNP	567
General Tridiagonal Matrix Factorization with No Pivoting	SGTNPF CGTNPF	DGTNPF ZGTNPF	570
General Tridiagonal Matrix Solve with No Pivoting	SGTNPS CGTNPS	DGTNPS ZGTNPS	573
Positive Definite Symmetric Tridiagonal Matrix Factorization	SPTF	DPTF	576
Positive Definite Symmetric Tridiagonal Matrix Solve	SPTS	DPTS	578
Triangular Band Equation Solve	STBSV◂ CTBSV◂	DTBSV◂ ZTBSV◂	580

◂ Level 2 BLAS

Sparse Linear Algebraic Equation Subroutines

The sparse linear algebraic equation subroutines provide direct and iterative solutions to linear systems of equations both for general sparse matrices and their transposes and for sparse symmetric matrices.

Table 85 (Page 1 of 2). List of Sparse Linear Algebraic Equation Subroutines

Descriptive Name	Long- Precision Subroutine	Page
General Sparse Matrix Factorization Using Storage by Indices, Rows, or Columns	DGSF	587
General Sparse Matrix or Its Transpose Solve Using Storage by Indices, Rows, or Columns	DGSS	593
General Sparse Matrix or Its Transpose Factorization, Determinant, and Solve Using Skyline Storage Mode	DGKFS	597
Symmetric Sparse Matrix Factorization, Determinant, and Solve Using Skyline Storage Mode	DSKFS	615
Iterative Linear System Solver for a General or Symmetric Sparse Matrix Stored by Rows	DSRIS	634

Table 85 (Page 2 of 2). List of Sparse Linear Algebraic Equation Subroutines		
Descriptive Name	**Long- Precision Subroutine**	**Page**
Sparse Positive Definite or Negative Definite Symmetric Matrix Iterative Solve Using Compressed-Matrix Storage Mode	DSMCG§	645
Sparse Positive Definite or Negative Definite Symmetric Matrix Iterative Solve Using Compressed-Diagonal Storage Mode	DSDCG	653
General Sparse Matrix Iterative Solve Using Compressed-Matrix Storage Mode	DSMGCG§	661
General Sparse Matrix Iterative Solve Using Compressed-Diagonal Storage Mode	DSDGCG	668
§ These subroutines are provided only for migration from earlier releases of ESSL and are not intended for use in new programs. Use DSRIS instead.		

Linear Least Squares Subroutines

The linear least squares subroutines provide least squares solutions to linear systems of equations for real general matrices. Two methods are provided: one that uses the singular value decomposition and another that uses a QR decomposition with column pivoting.

Table 86. List of Linear Least Squares Subroutines			
Descriptive Name	**Short-Precision Subroutine**	**Long-Precision Subroutine**	**Page**
Singular Value Decomposition for a General Matrix	SGESVF	DGESVF	676
Linear Least Squares Solution for a General Matrix Using the Singular Value Decomposition	SGESVS	DGESVS	684
Linear Least Squares Solution for a General Matrix Using a QR Decomposition with Column Pivoting	SGELLS	DGELLS	689

Dense and Banded Linear Algebraic Equation Considerations

This section provides some key points about using the dense and banded linear algebraic equation subroutines.

Use Considerations

1. To solve a system of equations, you need to use both the factorization and solve subroutines for the type of matrix you have. Each factorization subroutine should be followed in your program by the corresponding solve subroutine. The output from the factorization subroutine should be used as input to the solve subroutine.
2. To solve a system of equations with one or more right-hand sides, follow the call to the factorization subroutine with one or more calls to a solve subroutine or one call to a multiple solve subroutine.

3. The ESSL naming conventions for the dense and banded linear algebraic equation subroutines are similar to those used in the LAPACK documentation. (LAPACK, as well as its documentation, is available from the sources listed in reference [8].) The following ESSL subroutines correspond to the LAPACK subroutines: SGETRF, DGETRF, CGETRF, ZGETRF, SGETRS, DGETRS, CGETRS, and ZGETRS.

Performance and Accuracy Considerations

1. Except in a few instances, the _GTNP subroutines provide better performance than the _GTNPF and _GTNPS subroutines. For details, see the subroutine descriptions.
2. The general subroutines (dense and banded) use partial pivoting for accuracy and fast performance.
3. The short-precision subroutines provide increased accuracy by accumulating intermediate results in long precision. Occasionally, for performance reasons, these intermediate results are stored.
4. There are ESSL-specific rules that apply to the results of computations on the workstation processors using the ANSI/IEEE standards. For details, see "What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?" on page 44.

Sparse Matrix Direct Solver Considerations

This section provides some key points about using the sparse matrix direct solver subroutines.

Use Considerations

1. To solve a sparse system of equations by a direct method, you must use both the factorization and solve subroutines. The factorization subroutine should be followed in your program by the corresponding solve subroutine; that is, the output from the factorization subroutine should be used as input to the solve subroutine.
2. To solve a system of equations with one or more right-hand sides, follow the call to the factorization subroutine with one or more calls to the solve subroutine.
3. The amount of storage required for the arrays depends on the sparsity pattern of the matrix. The requirement that $lna > 2nz$ on entry to DGSF does not guarantee a successful run of the program. Some programs may be terminated because of the large number of fill-ins generated upon factorization. Fill-ins generated in a program depend on the structure of each matrix. If a large number of fill-ins is anticipated when factoring a matrix, the value of *lna* should be large enough to accommodate your problem.

Performance and Accuracy Considerations

1. To make the subroutine more efficient, an input matrix comprised of all nonzero elements is preferable. See the syntax description of each subroutine for details.
2. DGSF optionally checks the validity of the indices and pointers of the input matrix. Use of this option is suggested; however, it may affect performance. For details, see the syntax description for DGSF.
3. In DGSS, if there are multiple sparse right-hand sides to be solved, you should take advantage of the sparsity by selecting a proper value for *jopt* (such as $jopt = 10$ or

11). If there is only one right-hand side to be solved, it is suggested that you do not exploit the sparsity.

4. In DGSF, the value you enter for the lower bound of all elements in the matrix (RPARM(1)) affects the accuracy of the result. Specifying a larger number allows you to gain some performance; however, you may lose some accuracy in the solution.
5. In DGSF, the threshold pivot tolerance (RPARM(2)) is used to select pivots. A value that is close to 0.0 approaches no pivoting. A value close to 1.0 approaches partial pivoting. A value of 0.1 is considered to be a good compromise between numerical stability and sparsity.
6. If the ESSL subroutine performs storage compressions, you receive an attention message. When this occurs, the performance of this subroutine is affected. You can improve the performance by increasing the value specified for *lna*.
7. There are ESSL-specific rules that apply to the results of computations on the workstation processors using the ANSI/IEEE standards. For details, see "What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?" on page 44.

Sparse Matrix Skyline Solver Considerations

This section provides some key points about using the sparse matrix skyline solver subroutines.

Use Considerations

1. To solve a system of equations with one or more right-hand sides, where the matrix is stored in skyline storage mode, you can use either of the following methods. The factored output matrix is the same for both of these methods.
   - Call the skyline subroutine with the combined factor-and-solve option.
   - Call the skyline subroutine with the factor-only option, followed in your program by a call to the same subroutine with the solve-only option. The factored output matrix resulting from the factorization should be used as input to the same subroutine to do the solve. You can solve for the right-hand sides in a single call or in individual calls.

   You also have the option of doing a partial factorization, where the subroutine assumes that the initial part of the input matrix is already factored. It then factors the remaining rows and columns. If you want, you can factor a very large matrix progressively by using this option.
2. Forward elimination can be done with or without scaling the right-hand side by the diagonal matrix elements. To perform the computation without scaling, call DGKFS with the normal solve-only option, and define the upper triangular skyline matrix (AU) as a diagonal. To perform the computation with scaling, call DGKFS with the transpose solve-only, option and define the lower triangular skyline matrix (AL) as a diagonal.
3. Back substitution can be done with or without scaling the right-hand side by the diagonal matrix elements. To perform the computation without scaling, call DGKFS with the transpose solve-only option, and define the upper triangular skyline matrix (AU) as a diagonal. To perform the computation with scaling, call DGKFS with the normal solve-only option, and define the lower triangular skyline matrix (AL) as a diagonal.

Performance and Accuracy Considerations

1. For optimal performance, use diagonal-out skyline storage mode for both your input and output matrices. If you specify profile-in skyline storage mode for your input matrix, and either you do not plan to use the factored output or you plan to do a solve only, it is more efficient to specify diagonal-out skyline storage mode for your output matrix. These rules apply to all the computations.
2. In some cases, elapsed time may be reduced significantly by using the combined factor-and-solve option to solve for all right-hand sides at once, in conjunction with the factorization, rather than doing the factorization and solve separately.
3. If you do a solve only, and you solve for more than one right-hand side, it is most efficient to call the skyline subroutine once with all right-hand sides, rather than once for each right-hand side.
4. The skyline subroutines allow some control over processing of the pivot (diagonal) elements of the matrix during the factorization phase. Pivot processing is controlled by `IPARM(10)` through `IPARM(15)` and `RPARM(10)` through `RPARM(15)`. If a pivot occurs within a range that is designated to be fixed (`IPARM(0) = 1`, `IPARM(10) = 1`, and the appropriate element `IPARM(11)` through `IPARM(15) = 1`), it is replaced with the corresponding element of `RPARM(11)` through `RPARM(15)`. Should this pivot fix-up occur, you receive an attention message. This message indicates that the matrix being factored may be unstable (singular or not definite). The results produced in this situation may be inaccurate, and you should review them carefully.

Sparse Matrix Iterative Solver Considerations

This section provides some key points about using the sparse matrix iterative solver subroutines.

Use Considerations

If you need to solve linear systems with different right-hand sides but with the same matrix using the preconditioned algorithms, you can reuse the incomplete factorization computed during the first call to the subroutine.

Performance and Accuracy Considerations

1. The DSMCG and DSMGCG subroutines are provided for migration purposes from earlier releases of ESSL. You get better performance and a wider choice of algorithms if you use the DSRIS subroutine.
2. To select the sparse matrix subroutine that provides the best performance, you must consider the sparsity pattern of the matrix. From this, you can determine the most efficient storage mode for your sparse matrix. ESSL provides a number of versions of the sparse matrix iterative solve subroutines. They operate on sparse matrices stored in row-wise, diagonal, and compressed-matrix storage modes. These storage modes are described in “Sparse Matrix” on page 92.

   Storage-by-rows is generally applicable. You should use this storage mode unless your matrices are already set up in one of the other storage modes. If, however, your matrix has a regular sparsity pattern—that is, where the nonzero elements are concentrated along a few diagonals—you may want to use compressed-diagonal storage mode. This can save some storage space. Compressed-matrix storage mode is provided for migration purposes from earlier releases of ESSL and is not intended

for use. (You get better performance and a wider choice of algorithms if you use the DSRIS subroutine, which uses storage-by-rows.)

3. The performance achieved in the sparse matrix iterative solver subroutines depends on the value specified for the relative accuracy ε. For details, see Notes for each subroutine.
4. You can select the iterative algorithm you want to use to solve your linear system. The methods include conjugate gradient (CG), conjugate gradient squared (CGS), generalized minimum residual (GMRES), more smoothly converging variant of the CGS method (Bi-CGSTAB), or transpose-free quasi-minimal residual method (TFQMR).
5. For a general sparse or positive definite symmetric matrix, the iterative algorithm may fail to converge for one of the following reasons:
   - The value of ε is too small, asking for too much precision.
   - The maximum number of iterations is too small, allowing too few iterations for the algorithm to converge.
   - The matrix is not positive real; that is, the symmetric part, $(\boldsymbol{A}+\boldsymbol{A}^{\mathrm{T}})/2$, is not positive definite.
   - The matrix is ill-conditioned, which may cause overflows during the computation.
6. These algorithms have a tendency to generate underflows that may hurt overall performance. The system default is to mask underflow, which improves the performance of these subroutines.

Linear Least Squares Considerations

This section provides some key points about using the linear least squares subroutines.

Use Considerations

If you want to use a singular value decomposition method to compute the minimal norm linear least squares solution of $\boldsymbol{AX} \cong \boldsymbol{B}$, calls to SGESVF or DGESVF should be followed by calls to SGESVS or DGESVS, respectively.

Performance and Accuracy Considerations

1. Least squares solutions obtained by using a singular value decomposition require more storage and run time than those obtained using a QR decomposition with column pivoting. The singular value decomposition method, however, is a more reliable way to handle rank deficiency.
2. The short-precision subroutines provide increased accuracy by accumulating intermediate results in long precision. Occasionally, for performance reasons, these intermediate results are stored.
3. The accuracy of the resulting singular values and singular vectors varies between the short- and long-precision versions of each subroutine. The degree of difference depends on the size and conditioning of the matrix computation.
4. There are ESSL-specific rules that apply to the results of computations on the workstation processors using the ANSI/IEEE standards. For details, see "What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?" on page 44.

Dense Linear Algebraic Equation Subroutines

This section contains the dense linear algebraic equation subroutine descriptions.

SGEF, DGEF, CGEF, and ZGEF—General Matrix Factorization

This subroutine factors a square general matrix ***A*** using Gaussian elimination with partial pivoting. To solve the system of equations with one or more right-hand sides, follow the call to these subroutines with one or more calls to SGES/SGESM, DGES/DGESM, CGES/CGESM, or ZGES/ZGESM, respectively. To compute the inverse of matrix ***A***, follow the call to these subroutines with a call to SGEICD or DGEICD, respectively.

Table 87. Data Types

A	**Subroutine**
Short-precision real	SGEF
Long-precision real	DGEF
Short-precision complex	CGEF
Long-precision complex	ZGEF

Note: The output from these factorization subroutines should be used only as input to the following subroutines for performing a solve or inverse: SGES/SGESM/SGEICD, DGES/DGESM/DGEICD, CGES/CGESM, and ZGES/ZGESM, respectively.

Syntax

Fortran	CALL SGEF | DGEF | CGEF | ZGEF (*a*, *lda*, *n*, *ipvt*)
C and C++	sgef | dgef | cgef | zgef (*a*, *lda*, *n*, *ipvt*);
PL/I	CALL SGEF | DGEF | CGEF | ZGEF (*a*, *lda*, *n*, *ipvt*);

On Entry

a
is the *n* by *n* general matrix ***A*** to be factored. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 87.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and $lda \geq n$.

n
is the order of matrix ***A***. Specified as: a fullword integer; $0 \leq n \leq lda$.

ipvt
See On Return.

On Return

a
is the *n* by *n* transformed matrix ***A***, containing the results of the factorization. See "Function" on page 469. Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 87.

ipvt
is the integer vector ***ipvt*** of length *n*, containing the pivot information necessary to construct matrix ***L*** from the information contained in the output array *a*. Returned as: a one-dimensional array of (at least) length *n*, containing fullword integers.

Notes

1. ***ipvt*** is not a permutation vector in the strict sense. It is used to record row interchanges in ***L*** due to partial pivoting.
2. Calling SGEFCD or DGEFCD with *iopt* = 0 is equivalent to calling SGEF or DGEF.

Function: The matrix ***A*** is factored using Gaussian elimination with partial pivoting to compute the ***LU*** factorization of ***A***, where:

ipvt is the vector containing the pivoting information.
L is a unit lower triangular matrix.
U is an upper triangular matrix.

The transformed matrix ***A*** contains ***U*** in the upper triangle. In its strict lower triangle, it contains the multipliers necessary to construct, with the help of ***ipvt***, a matrix ***L***, such that ***A*** = ***LU***.

If *n* is 0, no computation is performed. See references [36] and [38].

Error Conditions

Resource Errors: Unable to allocate internal work area.

Computational Errors: Matrix ***A*** is singular.

- One or more columns of ***L*** and the corresponding diagonal of ***U*** contain all zeros (all columns of ***L*** are checked). The first column, *i*, of ***L*** with a corresponding ***U*** = 0 diagonal element is identified in the computational error message.
- The return code is set to 1.
- *i* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2103 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see "What Can You Do about ESSL Computational Errors?" on page 48.

Input-Argument Errors

1. *lda* ≤ 0
2. *n* < 0
3. *n* > *lda*

Example 1: This example shows a factorization of a real general matrix ***A*** of order 9.

Call Statement and Input

```
           A  LDA  N   IPVT
           |   |   |    |
CALL SGEF( A , 9 , 9 , IPVT )
```

```
                 ⎡                                                                ⎤
                 ⎢ 1.0  1.0  1.0  1.0  0.0  0.0   0.0   0.0   0.0 ⎥
                 ⎢ 1.0  1.0  1.0  1.0  1.0  0.0   0.0   0.0   0.0 ⎥
                 ⎢ 4.0  1.0  1.0  1.0  1.0  1.0   0.0   0.0   0.0 ⎥
                 ⎢ 0.0  5.0  1.0  1.0  1.0  1.0   1.0   0.0   0.0 ⎥
A     =          ⎢ 0.0  0.0  6.0  1.0  1.0  1.0   1.0   1.0   0.0 ⎥
                 ⎢ 0.0  0.0  0.0  7.0  1.0  1.0   1.0   1.0   1.0 ⎥
                 ⎢ 0.0  0.0  0.0  0.0  8.0  1.0   1.0   1.0   1.0 ⎥
                 ⎢ 0.0  0.0  0.0  0.0  0.0  9.0   1.0   1.0   1.0 ⎥
                 ⎢ 0.0  0.0  0.0  0.0  0.0  0.0  10.0  11.0  12.0 ⎥
                 ⎣                                                                ⎦
```

Output

```
        ⎡                                                                                ⎤
        ⎢ 4.0000  1.0000  1.0000  1.0000   1.0000   1.0000   0.0000   0.0000   0.0000 ⎥
        ⎢ 0.0000  5.0000  1.0000  1.0000   1.0000   1.0000   1.0000   0.0000   0.0000 ⎥
        ⎢ 0.0000  0.0000  6.0000  1.0000   1.0000   1.0000   1.0000   1.0000   0.0000 ⎥
        ⎢ 0.0000  0.0000  0.0000  7.0000   1.0000   1.0000   1.0000   1.0000   1.0000 ⎥
A     =  ⎢ 0.0000  0.0000  0.0000  0.0000   8.0000   1.0000   1.0000   1.0000   1.0000 ⎥
        ⎢ 0.0000  0.0000  0.0000  0.0000   0.0000   9.0000   1.0000   1.0000   1.0000 ⎥
        ⎢ 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000  10.0000  11.0000  12.0000 ⎥
        ⎢ 0.2500  0.1500  0.1000  0.0714   0.0536  -0.0694  -0.0306   0.1806   0.3111 ⎥
        ⎢ 0.2500  0.1500  0.1000  0.0714  -0.0714  -0.0556  -0.0194   0.9385  -0.0031 ⎥
        ⎣                                                                                ⎦

IPVT     =  (3, 4, 5, 6, 7, 8, 9, 8, 9)
```

Example 2: This example shows a factorization of a complex general matrix ***A*** of order 4.

Call Statement and Input

```
           A  LDA  N   IPVT
           |   |   |    |
CALL CGEF( A , 4 , 4 , IPVT )
```

```
                 ⎡                                                    ⎤
                 ⎢ (1.0, 2.0) (1.0, 7.0) (2.0, 4.0) (3.0, 1.0) ⎥
A     =          ⎢ (2.0, 0.0) (1.0, 3.0) (4.0, 4.0) (2.0, 3.0) ⎥
                 ⎢ (2.0, 1.0) (5.0, 0.0) (3.0, 6.0) (0.0, 0.0) ⎥
                 ⎢ (8.0, 5.0) (1.0, 9.0) (6.0, 6.0) (8.0, 1.0) ⎥
                 ⎣                                                    ⎦
```

Output

```
        ⎡                                                                                ⎤
        ⎢  (8.0000, 5.0000)   (1.0000, 9.0000)  (6.0000, 6.0000)  (8.0000, 1.0000) ⎥
A     =  ⎢  (0.2022, 0.1236)   (1.9101, 5.0562)  (1.5281, 2.0449) (1.5056, -0.1910) ⎥
        ⎢ (0.2360, -0.0225) (-0.0654, -0.9269) (-0.3462, 6.2692) (-1.6346, 1.3269) ⎥
        ⎢ (0.1798, -0.1124)   (0.2462, 0.1308) (0.4412, -0.3655)  (0.2900, 2.3864) ⎥
        ⎣                                                                                ⎦

IPVT     =  (4, 4, 3, 4)
```

SGES, DGES, CGES, and ZGES—General Matrix, Its Transpose, or Its Conjugate Transpose Solve

These subroutines solve the system $\boldsymbol{Ax} = \boldsymbol{b}$ for $\boldsymbol{x}$, where $\boldsymbol{A}$ is a general matrix and $\boldsymbol{x}$ and $\boldsymbol{b}$ are vectors. Using the *iopt* argument, they can also solve the real system $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x} = \boldsymbol{b}$ or the complex system $\boldsymbol{A}^{\mathrm{H}}\boldsymbol{x} = \boldsymbol{b}$ for $\boldsymbol{x}$. These subroutines use the results of the factorization of matrix $\boldsymbol{A}$, produced by a preceding call to SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, or ZGEF, respectively.

Table 88. Data Types

$\boldsymbol{A}$, $\boldsymbol{b}$, $\boldsymbol{x}$	**Subroutine**
Short-precision real	SGES
Long-precision real	DGES
Short-precision complex	CGES
Long-precision complex	ZGES

Note: The input to these solve subroutines must be the output from the factorization subroutines SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, and ZGEF, respectively.

Syntax

Fortran	CALL SGES | DGES | CGES | ZGES (*a*, *lda*, *n*, *ipvt*, *bx*, *iopt*)
C and C++	sges | dges | cges | zges (*a*, *lda*, *n*, *ipvt*, *bx*, *iopt*);
PL/I	CALL SGES | DGES | CGES | ZGES (*a*, *lda*, *n*, *ipvt*, *bx*, *iopt*);

On Entry

a
is the factorization of matrix $\boldsymbol{A}$, produced by a preceding call to SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, or ZGEF, respectively. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 88.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and $lda \geq n$.

n
is the order of matrix $\boldsymbol{A}$. Specified as: a fullword integer; $0 \leq n \leq lda$.

ipvt
is the integer vector ***ipvt*** of length *n*, produced by a preceding call to SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, or ZGEF, respectively. It contains the pivot information necessary to construct matrix $\boldsymbol{L}$ from the information contained in the array specified for *a*.

Specified as: a one-dimensional array of (at least) length *n*, containing fullword integers.

bx
is the vector $\boldsymbol{b}$ of length *n*, containing the right-hand side of the system. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 88.

iopt
determines the type of computation to be performed, where:

If *iopt* = 0, ***A*** is used in the computation.

If *iopt* = 1, $\boldsymbol{A}^{T}$ is used in SGES and DGES. $\boldsymbol{A}^{H}$ is used in CGES and ZGES.

Note: No data should be moved to form $\boldsymbol{A}^{T}$ or $\boldsymbol{A}^{H}$; that is, the matrix ***A*** should always be stored in its untransposed form.

Specified as: a fullword integer; *iopt* = 0 or 1.

On Return

bx
is the solution vector ***x*** of length *n*, containing the results of the computation.
Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 88 on page 471.

Notes

1. The scalar data specified for input arguments *lda* and *n* for these subroutines must be the same as the corresponding input arguments specified for SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, and ZGEF, respectively.
2. The array data specified for input arguments *a* and *ipvt* for these subroutines must be the same as the corresponding output arguments for SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, and ZGEF, respectively.
3. The vectors and matrices used in this computation must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.

Function: The system $\boldsymbol{Ax} = \boldsymbol{b}$ is solved for ***x***, where ***A*** is a general matrix and ***x*** and ***b*** are vectors. Using the *iopt* argument, this subroutine can also solve the real system $\boldsymbol{A}^{T}\boldsymbol{x} = \boldsymbol{b}$ or the complex system $\boldsymbol{A}^{H}\boldsymbol{x} = \boldsymbol{b}$ for ***x***. These subroutines use the results of the factorization of matrix ***A***, produced by a preceding call to SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, or ZGEF, respectively. The transformed matrix ***A*** consists of the upper triangular matrix ***U*** and the multipliers necessary to construct ***L*** using ***ipvt***, as defined in "Function" on page 469. For a description of how ***A*** is factored, see "SGEF, DGEF, CGEF, and ZGEF—General Matrix Factorization" on page 468.

If *n* is 0, no computation is performed. See references [36] and [38].

Error Conditions

Computational Errors: None

Note: If the factorization performed by SGEF, DGEF, CGEF, ZGEF, SGEFCD, DGEFCD, or DGEFP failed because a pivot element is zero, the results returned by this subroutine are unpredictable, and there may be a divide-by-zero program exception message.

Input-Argument Errors

1. $lda \leq 0$
2. $n < 0$
3. $n > lda$
4. *iopt* ≠ 0 or 1

Example 1

Part 1: This part of the example shows how to solve the system $\boldsymbol{Ax} = \boldsymbol{b}$, where matrix $\boldsymbol{A}$ is the same matrix factored in the "Example 1" on page 469 for SGEF and DGEF.

Call Statement and Input

```
          A  LDA  N   IPVT   BX  IOPT
          |   |   |    |     |    |
CALL SGES( A , 9 , 9 , IPVT , BX , 0  )

IPVT     =  (3, 4, 5, 6, 7, 8, 9, 8, 9)
BX       =  (4.0, 5.0, 9.0, 10.0, 11.0, 12.0, 12.0, 12.0, 33.0)
A        =(same as output A in "Example 1" on page 469)
```

Output

```
BX       =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
```

Part 2: This part of the example shows how to solve the system $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x} = \boldsymbol{b}$, where matrix $\boldsymbol{A}$ is the input matrix factored in "Example 1" on page 469 for SGEF and DGEF. Most of the input is the same in Part 2 as in Part 1.

Call Statement and Input

```
          A  LDA  N   IPVT   BX  IOPT
          |   |   |    |     |    |
CALL SGES( A , 9 , 9 , IPVT , BX , 1  )

IPVT     =  (3, 4, 5, 6, 7, 8, 9, 8, 9)
BX       =  (6.0, 8.0, 10.0, 12.0, 13.0, 14.0, 15.0, 15.0, 15.0)
A        =(same as output A in "Example 1" on page 469)
```

Output

```
BX       =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
```

Example 2

Part 1: This part of the example shows how to solve the system $\boldsymbol{Ax} = \boldsymbol{b}$, where matrix $\boldsymbol{A}$ is the same matrix factored in the "Example 2" on page 470 for CGEF and ZGEF.

Call Statement and Input

```
          A  LDA  N   IPVT   BX  IOPT
          |   |   |    |     |    |
CALL CGES( A , 4 , 4 , IPVT , BX , 0  )

IPVT     =  (4, 4, 3, 4)
BX       =  ((-10.0, 85.0), (-6.0, 61.0), (10.0, 38.0),
             (58.0, 168.0))
A        =(same as output A in "Example 1" on page 469)
```

Output

```
BX       =  ((9.0, 0.0), (5.0, 1.0), (1.0, 6.0), (3.0, 4.0))
```

Part 2: This part of the example shows how to solve the system $A^{H}x = b$, where matrix A is the input matrix factored in "Example 2" on page 470 for CGEF and ZGEF. Most of the input is the same in Part 2 as in Part 1.

Call Statement and Input

```
           A  LDA  N   IPVT   BX  IOPT
           |   |   |    |     |    |
CALL CGES( A , 4 , 4 , IPVT , BX , 1  )

IPVT     =  (4, 4, 3, 4)
BX       =  ((71.0, 12.0), (61.0, -70.0), (123.0, -34.0),
             (68.0, 7.0))
A        =(same as output A in "Example 1" on page 469)
```

Output

```
BX       =  ((9.0, 0.0), (5.0, 1.0), (1.0, 6.0), (3.0, 4.0))
```

SGESM, DGESM, CGESM, and ZGESM—General Matrix, Its Transpose, or Its Conjugate Transpose Multiple Right-Hand Side Solve

These subroutines solve the following systems of equations for multiple right-hand sides, where ***A***, ***X***, and ***B*** are general matrices. SGESM and DGESM solve one of the following:

1. $AX = B$
2. $A^{T}X = B$

CGESM and ZGESM solve one of the following:

1. $AX = B$
2. $A^{T}X = B$
3. $A^{H}X = B$

These subroutines use the results of the factorization of matrix ***A***, produced by a preceding call to SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, or ZGEF, respectively.

Table 89. Data Types

A, ***B***, ***X***	**Subroutine**
Short-precision real	SGESM
Long-precision real	DGESM
Short-precision complex	CGESM
Long-precision complex	ZGESM

Note: The input to these solve subroutines must be the output from the factorization subroutines SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, and ZGEF, respectively.

Syntax

Fortran	CALL SGESM | DGESM | CGESM | ZGESM (*trans*, *a*, *lda*, *n*, *ipvt*, *b*, *ldb*, *nrhs*)
C and C++	sgesm | dgesm | cgesm | zgesm (*trans*, *a*, *lda*, *n*, *ipvt*, *b*, *ldb*, *nrhs*);
PL/I	CALL SGESM | DGESM | CGESM | ZGESM (*trans*, *a*, *lda*, *n*, *ipvt*, *b*, *ldb*, *nrhs*);

On Entry

trans

indicates the form of matrix ***A*** to use in the computation, where:

If *transa* = 'N', ***A*** is used in the computation, resulting in equation 1.

If *transa* = 'T', A^{T} is used in the computation, resulting in equation 2.

If *transa* = 'C', A^{H} is used in the computation, resulting in equation 3.

Specified as: a single character. It must be 'N', 'T', or 'C'.

a
is the factorization of matrix ***A***, produced by a preceding call to SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, or ZGEF, respectively. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 89.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and $lda \geq n$.

n
is the order of matrix ***A***. Specified as: a fullword integer; $0 \leq n \leq lda$.

ipvt
is the integer vector ***ipvt*** of length *n*, produced by a preceding call to SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, or ZGEF, respectively. It contains the pivot information necessary to construct matrix ***L*** from the information contained in the array specified for *a*.

Specified as: a one-dimensional array of (at least) length *n*, containing fullword integers.

b
is the matrix ***B***, containing the *nrhs* right-hand sides of the system. The right-hand sides, each of length *n*, reside in the columns of matrix ***B***. Specified as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 89 on page 475.

ldb
is the leading dimension of the array specified for *b*. Specified as: a fullword integer; $ldb > 0$ and $ldb \geq n$.

nrhs
is the number of right-hand sides in the system to be solved. Specified as: a fullword integer; $nrhs \geq 0$.

On Return

b
is the matrix ***B***, containing the *nrhs* solutions to the system in the columns of ***B***. Specified as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 89 on page 475.

Notes

1. For SGESM and DGESM, if you specify `'C'` for the *trans* argument, it is interpreted as though you specified `'T'`.
2. The scalar data specified for input arguments *lda* and *n* for these subroutines must be the same as the corresponding input arguments specified for SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, and ZGEF, respectively.
3. The array data specified for input arguments *a* and *ipvt* for these subroutines must be the same as the corresponding output arguments for SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, and ZGEF, respectively.
4. The vectors and matrices used in this computation must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.

Function: One of the following systems of equations is solved for multiple right-hand sides:

1. $\boldsymbol{AX} = \boldsymbol{B}$
2. $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{X} = \boldsymbol{B}$

3. $A^H X = B$ (only for CGESM and ZGESM)

where A, B, and X are general matrices. These subroutines use the results of the factorization of matrix A, produced by a preceding call to SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, or ZGEF, respectively. The transformed matrix A consists of the upper triangular matrix U and the multipliers necessary to construct L using ***ipvt***, as defined in "Function" on page 469. For a description of how A is factored, see "SGEF, DGEF, CGEF, and ZGEF—General Matrix Factorization" on page 468.

If *n* or *nrhs* is 0, no computation is performed. See references [36] and [38].

Error Conditions

Computational Errors: None

Note: If the factorization performed by SGEF, DGEF, CGEF, ZGEF, SGEFCD, DGEFCD, or DGEFP failed because a pivot element is zero, the results returned by this subroutine are unpredictable, and there may be a divide-by-zero program exception message.

Input-Argument Errors

1. *trans* ≠ 'N', 'T', or 'C'
2. *lda*, *ldb* ≤ 0
3. *n* < 0
4. *n* > *lda*, *ldb*
5. *nrhs* < 0

Example 1

Part 1: This part of the example shows how to solve the system $AX = B$ for two right-hand sides, where matrix A is the same matrix factored in the "Example 1" on page 469 for SGEF and DGEF.

Call Statement and Input

```
           TRANS  A  LDA  N   IPVT   B  LDB  NRHS
             |    |   |   |    |     |   |    |
CALL SGESM( 'N' , A , 9 , 9 , IPVT , B , 9 ,  2  )

IPVT     =  (3, 4, 5, 6, 7, 8, 9, 8, 9)
A        =(same as output A in "Example 1" on page 469)

         ┌              ┐
         │  4.0   10.0  │
         │  5.0   15.0  │
         │  9.0   24.0  │
         │ 10.0   35.0  │
B    =   │ 11.0   48.0  │
         │ 12.0   63.0  │
         │ 12.0   70.0  │
         │ 12.0   78.0  │
         │ 33.0  266.0  │
         └              ┘
```

Output

```
          ┌          ┐
          │ 1.0  1.0 │
          │ 1.0  2.0 │
          │ 1.0  3.0 │
          │ 1.0  4.0 │
B    =    │ 1.0  5.0 │
          │ 1.0  6.0 │
          │ 1.0  7.0 │
          │ 1.0  8.0 │
          │ 1.0  9.0 │
          └          ┘
```

Part 2: This part of the example shows how to solve the system $A^TX = B$ for two right-hand sides, where matrix A is the input matrix factored in "Example 1" on page 469 for SGEF and DGEF.

Call Statement and Input

```
           TRANS  A  LDA  N   IPVT   B  LDB  NRHS
             |    |   |   |    |     |   |    |
CALL SGESM( 'T' , A , 9 , 9 , IPVT , B , 9 ,  2  )

IPVT     =  (3, 4, 5, 6, 7, 8, 9, 8, 9)
A        =(same as output A in "Example 1" on page 469)
```

```
          ┌           ┐
          │  6.0   15.0 │
          │  8.0   26.0 │
          │ 10.0   40.0 │
          │ 12.0   57.0 │
B    =    │ 13.0   76.0 │
          │ 14.0   97.0 │
          │ 15.0  120.0 │
          │ 15.0  125.0 │
          │ 15.0  129.0 │
          └           ┘
```

Output

```
          ┌          ┐
          │ 1.0  1.0 │
          │ 1.0  2.0 │
          │ 1.0  3.0 │
          │ 1.0  4.0 │
B    =    │ 1.0  5.0 │
          │ 1.0  6.0 │
          │ 1.0  7.0 │
          │ 1.0  8.0 │
          │ 1.0  9.0 │
          └          ┘
```

Example 2

Part 1: This part of the example shows how to solve the system $AX = B$ for two right-hand sides, where matrix A is the same matrix factored in the "Example 2" on page 470 for CGEF and ZGEF.

Call Statement and Input

```
           TRANS  A  LDA  N   IPVT   B  LDB  NRHS
             |    |   |   |    |     |   |    |
CALL CGESM( 'N' , A , 4 , 4 , IPVT , B , 4 ,  2  )

IPVT     =  (4, 4, 3, 4)
A        =(same as output A in "Example 2" on page 470)

        ⌈                                  ⌉
        | (-10.0, 85.0)  (-11.0, 53.0) |
B    =  |  (-6.0, 61.0)   (-6.0, 54.0) |
        |  (10.0, 38.0)    (2.0, 40.0) |
        | (58.0, 168.0)  (15.0, 105.0) |
        ⌊                                  ⌋
```

Output

```
        ⌈                          ⌉
        | (9.0, 0.0)  (1.0, 1.0) |
B    =  | (5.0, 1.0)  (2.0, 2.0) |
        | (1.0, 6.0)  (3.0, 3.0) |
        | (3.0, 4.0)  (4.0, 4.0) |
        ⌊                          ⌋
```

Part 2: This part of the example shows how to solve the system $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{X} = \boldsymbol{B}$ for two right-hand sides, where matrix $\boldsymbol{A}$ is the input matrix factored in "Example 2" on page 470 for CGEF and ZGEF.

Call Statement and Input

```
           TRANS  A  LDA  N   IPVT   B  LDB  NRHS
             |    |   |   |    |     |   |    |
CALL CGESM( 'T' , A , 4 , 4 , IPVT , B , 4 ,  2  )

IPVT     =  (4, 4, 3, 4)
A        =(same as output A in "Example 2" on page 470)

        ⌈                                  ⌉
        |   (71.0, 12.0)   (18.0, 68.0) |
B    =  |  (61.0, -70.0)  (-27.0, 71.0) |
        | (123.0, -34.0)  (-11.0, 97.0) |
        |    (68.0, 7.0)   (28.0, 50.0) |
        ⌊                                  ⌋
```

Output

```
        ⌈                          ⌉
        | (9.0, 0.0)  (1.0, 1.0) |
B    =  | (5.0, 1.0)  (2.0, 2.0) |
        | (1.0, 6.0)  (3.0, 3.0) |
        | (3.0, 4.0)  (4.0, 4.0) |
        ⌊                          ⌋
```

Part 3: This part of the example shows how to solve the system $\boldsymbol{A}^{\mathrm{H}}\boldsymbol{X} = \boldsymbol{B}$ for two right-hand sides, where matrix $\boldsymbol{A}$ is the input matrix factored in "Example 2" on page 470 for CGEF and ZGEF.

Call Statement and Input

```
           TRANS  A  LDA  N   IPVT   B  LDB  NRHS
             |    |   |   |    |     |   |    |
CALL CGESM( 'C' , A , 4 , 4 , IPVT , B , 4 ,  2  )

IPVT     =  (4, 4, 3, 4)
A        =(same as output A in "Example 2" on page 470)

        ⌈                                  ⌉
        |  (58.0, -3.0)   (45.0, 20.0)  |
B    =  | (68.0, -31.0)  (83.0, -20.0)  |
        | (89.0, -22.0)    (98.0, 1.0)  |
        |  (53.0, 15.0)   (45.0, 25.0)  |
        ⌊                                  ⌋
```

Output

```
        ⌈                          ⌉
        | (1.0, 4.0)  (4.0, 5.0)  |
B    =  | (2.0, 3.0)  (3.0, 4.0)  |
        | (3.0, 2.0)  (2.0, 3.0)  |
        | (4.0, 1.0)  (1.0, 2.0)  |
        ⌊                          ⌋
```

SGETRF, DGETRF, CGETRF and ZGETRF—General Matrix Factorization

These subroutines factor general matrix ***A*** using Gaussian elimination with partial pivoting. To solve the system of equations with one or more right-hand sides, follow the call to these subroutines with one or more calls to SGETRS, DGETRS CGETRS, or ZGETRS, respectively. To compute the inverse of matrix ***A***, follow the call to these subroutines with a call to SGEICD or DGEICD, respectively.

Table 90. Data Types

A	**Subroutine**
Short-precision real	SGETRF
Long-precision real	DGETRF
Short-precision complex	CGETRF
Long-precision complex	ZGETRF

Note: The output from these factorization subroutines should be used only as input to the following subroutines for performing a solve or inverse: SGETRS, DGETRS, CGETRS, ZGETRS, SGEICD or DGEICD respectively.

Syntax

Fortran	CALL SGETRF | DGETRF | CGETRF | ZGETRF (*m*, *n*, *a*, *lda*, *ipvt*, *info*)
C and C++	sgetrf | dgetrf | cgetrf | zgetrf (*m*, *n*, *a*, *lda*, *ipvt*, *info*);
PL/I	CALL SGETRF | DGETRF | CGETRF | ZGETRF (*m*, *n*, *a*, *lda*, *ipvt*, *info*);

On Entry

m
the number of rows in general matrix ***A*** used in the computation. Specified as: a fullword integer; $0 \le m \le lda$.

n
the number of columns in general matrix ***A*** used in the computation. Specified as: a fullword integer; $n \ge 0$.

a
is the *m* by *n* general matrix ***A*** to be factored. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 90.

lda
is the leading dimension of matrix ***A***. Specified as: a fullword integer; $lda > 0$ and $lda \ge m$.

ipvt
See On Return.

info
See On Return.

On Return

a
is the *m* by *n* transformed matrix ***A***, containing the results of the factorization. See "Function." Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 90 on page 481.

ipvt
is the integer vector ***ipvt*** of length min(*m*,*n*), containing the pivot information necessary to construct matrix ***L*** from the information contained in the output array A. Returned as: a one-dimensional array of (at least) length min(*m*,*n*), containing fullword integers,where $1 \leq ipvt(i) \leq m$.

info
has the following meaning:

If $info = 0$, the factorization of general matrix ***A*** completed successfully.

If $info > 0$, *info* is set equal to the first *i*, where $\boldsymbol{U}_{ii}$ is singular and its inverse could not be computed.

Specified as: a fullword integer; $info \geq 0$.

Notes

1. In your C program, argument *info* must be passed by reference.
2. The matrix ***A*** and vector ***ipvt*** must have no common elements; otherwise results are unpredictable.
3. The way these subroutines handle singularity differs from LAPACK. These subroutines us the *info* argument to provide information about the singularity of ***A***, like LAPACK, but also provide an error message.
4. On both input and output, matrix ***A*** conforms to LAPACK format.

Function: The matrix ***A*** is factored using Gaussian elimination with partial pivoting to compute the ***LU*** factorization of ***A***, where:

ipvt is the vector containing the pivoting information.
L is a unit lower triangular matrix.
U is an upper triangular matrix.

On output, the transformed matrix ***A*** contains ***U*** in the upper triangle (if $m \geq n$) or upper trapezoid (if $m < n$). In its strict lower triangle (if $m \leq n$) or lower trapezoid (if $m > n$), it contains the multipliers necessary to construct, with the help of ***ipvt***, a matrix ***L***, such that $\boldsymbol{A} = \boldsymbol{LU}$.

If *m* or *n* is 0, no computation is performed and the subroutine returns after doing some parameter checking. See references [36] and [59].

Error Conditions

Resource Errors: Unable to allocate internal work area.

Computational Errors: Matrix ***A*** is singular.

- The first column, *i*, of ***L*** with a corresponding $\boldsymbol{U}_{ii} = 0$ diagonal element is identified in the computational error message.

- The computational error message may occur multiple times with processing continuing after each error, because the default for the number of allowable errors for error code 2146 is set to be unlimited in the ESSL error option table.

Input-Argument Errors

1. $m < 0$
2. $n < 0$
3. $m > lda$
4. $lda \leq 0$

Example 1: This example shows a factorization of a real general matrix ***A*** of order 9.

Call Statement and Input

```
             M   N   A  LDA  IPVT  INFO
             |   |   |   |     |     |
CALL DGETRF( 9 , 9 , A,  9 , IPVT, INFO )
```

```
        ⎡                                                ⎤
        ⎢ 1.0  1.2  1.4  1.6  1.8  2.0  2.2  2.4  2.6 ⎥
        ⎢ 1.2  1.0  1.2  1.4  1.6  1.8  2.0  2.2  2.4 ⎥
        ⎢ 1.4  1.2  1.0  1.2  1.4  1.6  1.8  2.0  2.2 ⎥
        ⎢ 1.6  1.4  1.2  1.0  1.2  1.4  1.6  1.8  2.0 ⎥
A    =  ⎢ 1.8  1.6  1.4  1.2  1.0  1.2  1.4  1.6  1.8 ⎥
        ⎢ 2.0  1.8  1.6  1.4  1.2  1.0  1.2  1.4  1.6 ⎥
        ⎢ 2.2  2.0  1.8  1.6  1.4  1.2  1.0  1.2  1.4 ⎥
        ⎢ 2.4  2.2  2.0  1.8  1.6  1.4  1.2  1.0  1.2 ⎥
        ⎢ 2.6  2.4  2.2  2.0  1.8  1.6  1.4  1.2  1.0 ⎥
        ⎣                                                ⎦
```

Output

```
        ⎡                                                ⎤
        ⎢ 2.6   2.4  2.2  2.0  1.8  1.6  1.4  1.2  1.0 ⎥
        ⎢ 0.4   0.3  0.6  0.8  1.1  1.4  1.7  1.9  2.2 ⎥
        ⎢ 0.5  -0.4  0.4  0.8  1.2  1.6  2.0  2.4  2.8 ⎥
        ⎢ 0.5  -0.3  0.0  0.4  0.8  1.2  1.6  2.0  2.4 ⎥
A    =  ⎢ 0.6  -0.3  0.0  0.0  0.4  0.8  1.2  1.6  2.0 ⎥
        ⎢ 0.7  -0.2  0.0  0.0  0.0  0.4  0.8  1.2  1.6 ⎥
        ⎢ 0.8  -0.2  0.0  0.0  0.0  0.0  0.4  0.8  1.2 ⎥
        ⎢ 0.8  -0.1  0.0  0.0  0.0  0.0  0.0  0.4  0.8 ⎥
        ⎢ 0.9  -0.1  0.0  0.0  0.0  0.0  0.0  0.0  0.4 ⎥
        ⎣                                                ⎦

IPVT     =  (9, 9, 9, 9, 9, 9, 9, 9, 9)
INFO     =  0
```

Example 2: This example shows a factorization of a complex general matrix ***A*** of order 9.

Call Statement and Input

```
             M   N   A  LDA  IPVT  INFO
             |   |   |   |     |     |
CALL ZGETRF( 9 , 9 , A,  9 , IPVT, INFO )
```

```
      ⎡                                                                                                                 ⎤
      ⎢ (2.0, 1.0) (2.4,-1.0) (2.8,-1.0) (3.2,-1.0)  (3.6,-1.0) (4.0,-1.0) (4.4,-1.0) (4.8,-1.0) (5.2,-1.0)           ⎥
      ⎢ (2.4, 1.0) (2.0, 1.0) (2.4,-1.0) (2.8,-1.0)  (3.2,-1.0) (3.6,-1.0) (4.0,-1.0) (4.4,-1.0) (4.8,-1.0)           ⎥
      ⎢ (2.8, 1.0) (2.4, 1.0) (2.0, 1.0) (2.4,-1.0)  (2.8,-1.0) (3.2,-1.0) (3.6,-1.0) (4.0,-1.0) (4.4,-1.0)           ⎥
      ⎢ (3.2, 1.0) (2.8, 1.0) (2.4, 1.0) (2.0, 1.0)  (2.4,-1.0) (2.8,-1.0) (3.2,-1.0) (3.6,-1.0) (4.0,-1.0)           ⎥
A  =  ⎢ (3.6, 1.0) (3.2, 1.0) (2.8, 1.0) (2.4, 1.0)  (2.0, 1.0) (2.4,-1.0) (2.8,-1.0) (3.2,-1.0) (3.6,-1.0)           ⎥
      ⎢ (4.0, 1.0) (3.6, 1.0) (3.2, 1.0) (2.8, 1.0)  (2.4, 1.0) (2.0, 1.0) (2.4,-1.0) (2.8,-1.0) (3.2,-1.0)           ⎥
      ⎢ (4.4, 1.0) (4.0, 1.0) (3.6, 1.0) (3.2, 1.0)  (2.8, 1.0) (2.4, 1.0) (2.0, 1.0) (2.4,-1.0) (2.8,-1.0)           ⎥
      ⎢ (4.8, 1.0) (4.4, 1.0) (4.0, 1.0) (3.6, 1.0)  (3.2, 1.0) (2.8, 1.0) (2.4, 1.0) (2.0, 1.0) (2.4,-1.0)           ⎥
      ⎢ (5.2, 1.0) (4.8, 1.0) (4.4, 1.0) (4.0, 1.0)  (3.6, 1.0) (3.2, 1.0) (2.8, 1.0) (2.4, 1.0) (2.0, 1.0)           ⎥
      ⎣                                                                                                                 ⎦
```

Output

```
      ⎡                                                                                                                 ⎤
      ⎢ (5.2, 1.0) (4.8, 1.0) (4.4, 1.0)   (4.0, 1.0)   (3.6, 1.0)  (3.2, 1.0) (2.8, 1.0) (2.4, 1.0) (2.0, 1.0)       ⎥
      ⎢ (0.4, 0.1) (0.6,-2.0) (1.1,-1.9)   (1.7,-1.9)   (2.3,-1.8)  (2.8,-1.8) (3.4,-1.7) (3.9,-1.7) (4.5,-1.6)       ⎥
      ⎢ (0.5, 0.1) (0.0,-0.1) (0.6,-1.9)   (1.2,-1.8)   (1.8,-1.7)  (2.5,-1.6) (3.1,-1.5) (3.7,-1.4) (4.3,-1.3)       ⎥
      ⎢ (0.6, 0.1) (0.0,-0.1) (-0.1,-0.1)  (0.7,-1.9)   (1.3,-1.7)  (2.0,-1.6) (2.7,-1.5) (3.4,-1.4) (4.0,-1.2)       ⎥
A  =  ⎢ (0.6, 0.1) (0.0,-0.1) (-0.1,-0.1) (-0.1, 0.0)   (0.7,-1.9)  (1.5,-1.7) (2.2,-1.6) (2.9,-1.5) (3.7,-1.3)       ⎥
      ⎢ (0.7, 0.1) (0.0,-0.1)  (0.0, 0.0) (-0.1, 0.0)  (-0.1, 0.0)  (0.8,-1.9) (1.6,-1.8) (2.4,-1.6) (3.2,-1.5)       ⎥
      ⎢ (0.8, 0.0) (0.0, 0.0)  (0.0, 0.0)  (0.0, 0.0)   (0.0, 0.0)  (0.0, 0.0) (0.8,-1.9) (1.7,-1.8) (2.5,-1.8)       ⎥
      ⎢ (0.9, 0.0) (0.0, 0.0)  (0.0, 0.0)  (0.0, 0.0)   (0.0, 0.0)  (0.0, 0.0) (0.0, 0.0) (0.8,-2.0) (1.7,-1.9)       ⎥
      ⎢ (0.9, 0.0) (0.0, 0.0)  (0.0, 0.0)  (0.0, 0.0)   (0.0, 0.0)  (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (0.8,-2.0)       ⎥
      ⎣                                                                                                                 ⎦
```

```
IPVT     =  (9, 9, 9, 9, 9, 9, 9, 9, 9)
INFO     =  0
```

SGETRS, DGETRS, CGETRS, and ZGETRS—General Matrix Multiple Right-Hand Side Solve

SGETRS and DGETRS solve one of the following systems of equations for multiple right-hand sides:

1. $\boldsymbol{AX} = \boldsymbol{B}$
2. $\boldsymbol{A}^{T}\boldsymbol{X} = \boldsymbol{B}$

CGETRS and ZGETRS solve one of the following systems of equations for multiple right-hand sides:

1. $\boldsymbol{AX} = \boldsymbol{B}$
2. $\boldsymbol{A}^{T}\boldsymbol{X} = \boldsymbol{B}$
3. $\boldsymbol{A}^{H}\boldsymbol{X} = \boldsymbol{B}$

In the formulas above:

A represents the general matrix ***A*** containing the ***LU*** factorization.
B represents the general matrix ***B*** containing the right-hand sides in its columns.
X represents the general matrix ***B*** containing the solution vectors in its columns.

These subroutines use the results of the factorization of matrix ***A***, produced by a preceding call to SGETRF, DGETRF, CGETRF, or ZGETRF, respectively.

Table 91. Data Types

A, ***B***	**Subroutine**
Short-precision real	SGETRS
Long-precision real	DGETRS
Short-precision complex	CGETRS
Long-precision complex	ZGETRS

Note: The input to these solve subroutines must be the output from the factorization subroutines SGETRF, DGETRF, CGETRF and ZGETRF, respectively.

Syntax

Fortran	CALL SGETRS | DGETRS | CGETRS | ZGETRS (*transa*, *n*, *nrhs*, *a*, *lda*, *ipvt*, *bx*, *ldb*, *info*)
C and C++	sgetrs | dgetrs | cgetrs | zgetrs (*transa*, *n*, *nrhs*, *a*, *lda*, *ipvt*, *bx*, *ldb*, *info*);
PL/I	CALL SGETRS | DGETRS | CGETRS | ZGETRS (*transa*, *n*, *nrhs*, *a*, *lda*, *ipvt*, *bx*, *ldb*, *info*);

On Entry

transa
indicates the form of matrix ***A*** to use in the computation, where:

If *transa* = 'N', ***A*** is used in the computation, resulting in solution 1.

If *transa* = 'T', $\boldsymbol{A}^{T}$ is used in the computation, resulting in solution 2.

If *transa* = 'C', $\boldsymbol{A}^{H}$ is used in the computation, resulting in solution 3.

Specified as: a single character; *transa* = 'N', 'T', or 'C'.

n
is the order of factored matrix ***A*** and the number of rows in matrix ***B***. Specified as: a fullword integer; $n \geq 0$.

nrhs
the number of right-hand sides—that is, the number of columns in matrix ***B*** used in the computation. Specified as: a fullword integer; $nrhs \geq 0$.

a
is the factorization of matrix ***A***, produced by a preceding call to SGETRF, DGETRF, CGETRF, or ZGETRF, respectively. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 91 on page 485.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and $lda \geq n$.

ipvt
is the integer vector ***ipvt*** of length *n*, produced by a preceding call to SGETRF, DGETRF, CGETRF or ZGETRF, respectively. It contains the pivot information necessary to construct matrix ***L*** from the information contained in the array specified for *a*.

Specified as: a one-dimensional array of (at least) length *n*, containing fullword integers, where $1 \leq ipvt(i) \leq n$.

bx
is the general matrix ***B*** containing the right-hand side of the system. Specified as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 91 on page 485.

ldb
is the leading dimension of the array specified for *b*. Specified as: a fullword integer; $ldb > 0$ and $ldb \geq n$.

info
See On Return.

On Return

bx
is the solution ***X*** containing the results of the computation. Returned as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 91 on page 485.

info
info has the following meaning:

If *info* = 0, the solve of general matrix ***A*** completed successfully.

Notes

1. In your C program, argument *info* must be passed by reference.
2. These subroutines accept lower case letters for the *transa* argument.
3. For SGETRS and DGETRS, if you specify 'C' for the *transa* argument, it is interpreted as though you specified 'T'.
4. The scalar data specified for input argument *n* must be the same for both _GETRF and _GETRS. In addition, the scalar data specified for input argument *m* in _GETRF **must be the same** as input argument *n* in both _GETRF and _GETRS.

   If, however, you do **not** plan to call _GETRS after calling _GETRF, then input arguments *m* and *n* in _GETRF do not need to be equal.

5. The array data specified for input arguments *a* and *ipvt* for these subroutines must be the same as the corresponding output arguments for SGETRF, DGETRF, CGETRF, and ZGETRF, respectively.
6. The matrices and vector used in this computation must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.
7. On both input and output, matrices ***A*** and ***B*** conform to LAPACK format.

Function: One of the following systems of equations is solved for multiple right-hand sides:

1. $\boldsymbol{AX} = \boldsymbol{B}$
2. $\boldsymbol{A}^{T}\boldsymbol{X} = \boldsymbol{B}$
3. $\boldsymbol{A}^{H}\boldsymbol{X} = \boldsymbol{B}$ (only for CGETRS and ZGETRS)

where ***A***, ***B***, and ***X*** are general matrices. These subroutines uses the results of the factorization of matrix ***A***, produced by a preceding call to SGETRF, DGETRF, CGETRF or ZGETRF, respectively. On input, the transformed matrix ***A*** consists of the upper triangular matrix ***U*** and the multipliers necessary to construct ***L*** using ***ipvt***, as defined in "Function" on page 482. For details on the factorization, see "SGETRF, DGETRF, CGETRF and ZGETRF—General Matrix Factorization" on page 481.

If $n = 0$ or $nrhs = 0$, no computation is performed and the subroutine returns after doing some parameter checking. See references [36] and [59].

Error Conditions

Computational Errors: None

Note: If the factorization performed by SGETRF, DGETRF, CGETRF or ZGETRF failed because a pivot element is zero, the results returned by this subroutine are unpredictable, and there may be a divide-by-zero program exception message.

Input-Argument Errors

1. *transa* ≠ 'N', 'T', or 'C'
2. $n < 0$
3. $nrhs < 0$
4. $n > lda$
5. $n > ldb$
6. $lda \leq 0$
7. $ldb \leq 0$

Example 1: This example shows how to solve the system $\boldsymbol{AX} = \boldsymbol{B}$, where matrix ***A*** is the same matrix factored in the "Example 1" on page 483 for DGETRF.

Call Statement and Input

```
           TRANSA  N  NRHS  A  LDA  IPIV  BX LDB  INFO
             |     |    |   |   |     |   |   |     |
CALL DGETRS('N' ,  9 ,  5 , A , 9 , IPIV, B , 9 , INFO)

IPVT     =  (9, 9, 9, 9, 9, 9, 9, 9, 9)
A       =(same as output A in "Example 1" on page 483)
```

```
        ⎡                                        ⎤
        ⎢ 93.0  186.0  279.0  372.0  465.0 ⎥
        ⎢ 84.4  168.8  253.2  337.6  422.0 ⎥
        ⎢ 76.6  153.2  229.8  306.4  383.0 ⎥
        ⎢ 70.0  140.0  210.0  280.0  350.0 ⎥
B    =  ⎢ 65.0  130.0  195.0  260.0  325.0 ⎥
        ⎢ 62.0  124.0  186.0  248.0  310.0 ⎥
        ⎢ 61.4  122.8  184.2  245.6  307.0 ⎥
        ⎢ 63.6  127.2  190.8  254.4  318.0 ⎥
        ⎢ 69.0  138.0  207.0  276.0  345.0 ⎥
        ⎣                                        ⎦
```

Output

```
        ⎡                                    ⎤
        ⎢ 1.0   2.0   3.0   4.0   5.0 ⎥
        ⎢ 2.0   4.0   6.0   8.0  10.0 ⎥
        ⎢ 3.0   6.0   9.0  12.0  15.0 ⎥
        ⎢ 4.0   8.0  12.0  16.0  20.0 ⎥
B    =  ⎢ 5.0  10.0  15.0  20.0  25.0 ⎥
        ⎢ 6.0  12.0  18.0  24.0  30.0 ⎥
        ⎢ 7.0  14.0  21.0  28.0  35.0 ⎥
        ⎢ 8.0  16.0  24.0  32.0  40.0 ⎥
        ⎢ 9.0  18.0  27.0  36.0  45.0 ⎥
        ⎣                                    ⎦

INFO     =  0
```

Example 2: This example shows how to solve the system $\boldsymbol{AX} = \boldsymbol{b}$, where matrix $\boldsymbol{A}$ is the same matrix factored in the "Example 2" on page 483 for ZGETRF.

Call Statement and Input

```
           TRANS   N  NRHS  A  LDA  IPIV  B  LDB  INFO
             |     |    |   |   |     |   |   |     |
CALL ZGETRS('N' ,  9 ,  5 , A , 9 , IPIV, B , 9 , INFO)

IPVT     =  (9, 9, 9, 9, 9, 9, 9, 9, 9)
A        =(same as output A in "Example 2" on page 483)
```

```
        ⎡                                                                    ⎤
        ⎢ (193.0,-10.6)  (200.0, 21.8)  (207.0, 54.2)  (214.0, 86.6)  (221.0,119.0) ⎥
        ⎢ (173.8, -9.4)  (178.8, 20.2)  (183.8, 49.8)  (188.8, 79.4)  (193.8,109.0) ⎥
        ⎢ (156.2, -5.4)  (159.2, 22.2)  (162.2, 49.8)  (165.2, 77.4)  (168.2,105.0) ⎥
        ⎢ (141.0,  1.4)  (142.0, 27.8)  (143.0, 54.2)  (144.0, 80.6)  (145.0,107.0) ⎥
B    =  ⎢ (129.0, 11.0)  (128.0, 37.0)  (127.0, 63.0)  (126.0, 89.0)  (125.0,115.0) ⎥
        ⎢ (121.0, 23.4)  (118.0, 49.8)  (115.0, 76.2)  (112.0,102.6)  (109.0,129.0) ⎥
        ⎢ (117.8, 38.6)  (112.8, 66.2)  (107.8, 93.8)  (102.8,121.4)   (97.8,149.0) ⎥
        ⎢ (120.2, 56.6)  (113.2, 86.2)  (106.2,115.8)   (99.2,145.4)   (92.2,175.0) ⎥
        ⎢ (129.0, 77.4)  (120.0,109.8)  (111.0,142.2). (102.0,174.6)   (93.0,207.0) ⎥
        ⎣                                                                    ⎦
```

Output

```
        ⎡                                                                  ⎤
        ⎢ (1.0,1.0)  (1.0,2.0)  (1.0,3.0) (1.0,4.0) (1.0,5.0) ⎥
        ⎢ (2.0,1.0)  (2.0,2.0)  (2.0,3.0) (2.0,4.0) (2.0,5.0) ⎥
        ⎢ (3.0,1.0)  (3.0,2.0)  (3.0,3.0) (3.0,4.0) (3.0,5.0) ⎥
        ⎢ (4.0,1.0)  (4.0,2.0)  (4.0,3.0) (4.0,4.0) (4.0,5.0) ⎥
B     = ⎢ (5.0,1.0)  (5.0,2.0)  (5.0,3.0) (5.0,4.0) (5.0,5.0) ⎥
        ⎢ (6.0,1.0)  (6.0,2.0)  (6.0,3.0) (6.0,4.0) (6.0,5.0) ⎥
        ⎢ (7.0,1.0)  (7.0,2.0)  (7.0,3.0) (7.0,4.0) (7.0,5.0) ⎥
        ⎢ (8.0,1.0)  (8.0,2.0)  (8.0,3.0) (8.0,4.0) (8.0,5.0) ⎥
        ⎢ (9.0,1.0)  (9.0,2.0)  (9.0,3.0) (9.0,4.0) (9.0,5.0) ⎥
        ⎣                                                                  ⎦

INFO     =  0
```

SGEFCD and DGEFCD—General Matrix Factorization, Condition Number Reciprocal, and Determinant

These subroutines factor general matrix ***A*** using Gaussian elimination. An estimate of the reciprocal of the condition number and the determinant of matrix ***A*** can also be computed. To solve a system of equations with one or more right-hand sides, follow the call to these subroutines with one or more calls to SGES/SGESM or DGES/DGESM, respectively. To compute the inverse of matrix ***A***, follow the call to these subroutines with a call to SGEICD and DGEICD, respectively.

Table 92. Data Types

A, *aux*, *rcond*, ***det***	**Subroutine**
Short-precision real	SGEFCD
Long-precision real	DGEFCD

Note: The output from these factorization subroutines should be used only as input to the following subroutines for performing a solve or inverse: SGES/SGESM/SGEICD and DGES/DGESM/DGEICD, respectively.

Syntax

Fortran	CALL SGEFCD | DGEFCD (*a*, *lda*, *n*, *ipvt*, *iopt*, *rcond*, *det*, *aux*, *naux*)
C and C++	sgefcd | dgefcd (*a*, *lda*, *n*, *ipvt*, *iopt*, *rcond*, *det*, *aux*, *naux*);
PL/I	CALL SGEFCD | DGEFCD (*a*, *lda*, *n*, *ipvt*, *iopt*, *rcond*, *det*, *aux*, *naux*);

On Entry

a
is a general matrix ***A*** of order *n*, whose factorization, reciprocal of condition number, and determinant are computed. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 92.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and $lda \geq n$.

n
is the order of matrix ***A***. Specified as: a fullword integer; $0 \leq n \leq lda$.

ipvt
See On Return.

iopt
indicates the type of computation to be performed, where:

If $iopt = 0$, the matrix is factored.

If $iopt = 1$, the matrix is factored, and the reciprocal of the condition number is computed.

If $iopt = 2$, the matrix is factored, and the determinant is computed.

If $iopt = 3$, the matrix is factored, and the reciprocal of the condition number and the determinant are computed.

Specified as: a fullword integer; $iopt = 0, 1, 2, \text{or } 3$.

rcond
See On Return.

det
See On Return.

aux
has the following meaning:

If *naux = 0* and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is is a storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing numbers of the data type indicated in Table 92 on page 490.

naux
is the size of the work area specified by *aux*—that is, the number of elements in *aux*. Specified as: a fullword integer, where:

If *naux* = 0 and error 2015 is unrecoverable, SGEFCD and DGEFCD dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux* ≥ *n*.

On Return

a
is the transformed matrix ***A*** of order *n*, containing the results of the factorization. See "Function" on page 492. Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 92 on page 490.

ipvt
is the integer vector ***ipvt*** of length *n*, containing the pivot information necessary to construct matrix ***L*** from the information contained in output matrix ***A***. Returned as: a one-dimensional array of (at least) length *n*, containing fullword integers.

rcond
is an estimate of the reciprocal of the condition number, *rcond*, of matrix ***A***. Returned as: a number of the data type indicated in Table 92 on page 490; *rcond* ≥ 0.

det
is the vector ***det***, containing the two components, det_1 and det_2, of the determinant of matrix ***A***. The determinant is:

$$det_1\left(10^{det_2}\right)$$

where $1 \leq det_1 < 10$. Returned as: an array of length 2, containing numbers of the data type indicated in Table 92 on page 490.

Notes

1. In your C program, argument *rcond* must be passed by reference.
2. When *iopt* = 0, these subroutines provide the same function as a call to SGEF or DGEF, respectively.

3. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: Matrix ***A*** is factored using Gaussian elimination with partial pivoting to compute the ***LU*** factorization of ***A***, where:

ipvt is the vector containing the pivoting information.
L is a unit lower triangular matrix.
U is an upper triangular matrix.

The transformed matrix ***A*** contains ***U*** in the upper triangle. In its strict lower triangle, it contains the multipliers necessary to construct, with the help of ***ipvt***, a matrix ***L***, such that ***A*** = ***LU***.

An estimate of the reciprocal of the condition number, *rcond*, and the determinant, ***det***, can also be computed by this subroutine. The estimate of the condition number uses an enhanced version of the algorithm described in references [63] and [64].

If *n* is 0, no computation is performed. See reference [36].

These subroutines call SGEF and DGEF, respectively, to perform the factorization. ***ipvt*** is an output vector of SGEF and DGEF. It is returned for use by SGES/SGESM and DGES/DGESM, the solve subroutines.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors: Matrix ***A*** is singular.

- If your program is not terminated by SGEF and DGEF, then SGEFCD and DGEFCD, respectively, return 0 for *rcond* and *det*.
- One or more columns of ***L*** and the corresponding diagonal of ***U*** contain all zeros (all columns of ***L*** are checked). The first column, *i*, of ***L*** with a corresponding ***U*** = 0 diagonal element is identified in the computational error message, issued by SGEF or DGEF, respectively.
- *i* can be determined at run time by using the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2103 in the ESSL error option table; otherwise, the default value causes your program to be terminated by SGEF or DGEF, respectively, when this error occurs. If your program is not terminated by SGEF or DGEF, respectively, the return code is set to 2. For details, see "What Can You Do about ESSL Computational Errors?" on page 48.

Input-Argument Errors

1. *lda* ≤ 0
2. *n* < 0
3. *n* > *lda*
4. *iopt* ≠ 0, 1, 2, or 3
5. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example: This example shows a factorization of matrix ***A*** of order 9. The input is the same as used in SGEF and DGEF. See "Example 1" on page 469. The reciprocal of the condition number and the determinant of matrix ***A*** are also computed. The values used to estimate the reciprocal of the condition number in this example are obtained with the following values:

$\|\boldsymbol{A}\|_1 = \max(6.0, 8.0, 10.0, 12.0, 13.0, 14.0, 15.0, 15.0, 15.0) = 15.0$

Estimate of $\|\boldsymbol{A}^{-1}\|_1 = 1091.87$

This estimate is equal to the actual *rcond* of $5.436(10^{-5})$, which is computed by SGEICD and DGEICD. (See "Example 1" on page 519.) On output, the value in ***det***, $|\boldsymbol{A}|$, is equal to 336.

Call Statement and Input

```
             A  LDA  N   IPVT  IOPT  RCOND   DET   AUX  NAUX
             |   |   |    |     |     |       |     |    |
CALL DGEFCD( A , 9 , 9 , IPVT , 3  , RCOND , DET , AUX , 9  )
```

```
A         =(same as input A in "Example 1" on page 469)
```

Output

```
A         =(same as output A in "Example 1" on page 469)
IPVT      =  (3, 4, 5, 6, 7, 8, 9, 8, 9)
RCOND     =  0.00005436
DET       =  (3.36, 2.00)
```

SPPF, DPPF, SPOF, DPOF, CPOF, and ZPOF—Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization

The SPPF and DPPF subroutines factor positive definite symmetric matrix ***A***, stored in lower-packed storage mode, using Gaussian elimination ($\boldsymbol{LDL}^{T}$) or the Cholesky factorization method. To solve a system of equations with one or more right-hand sides, follow the call to these subroutines with one or more calls to SPPS or DPPS, respectively. To find the inverse of matrix ***A***, follow the call to these subroutines, performing Cholesky factorization, with a call to SPPICD or DPPICD, respectively.

The SPOF, DPOF, CPOF, and ZPOF subroutines factor matrix ***A*** stored in upper or lower storage mode, where:

- For SPOF and DPOF, ***A*** is a positive definite symmetric matrix.
- For CPOF and ZPOF, ***A*** is a positive definite complex Hermitian matrix.

Matrix ***A*** is factored using Cholesky factorization, ($\boldsymbol{LL}^{T}$ or $\boldsymbol{U}^{T}\boldsymbol{U}$ for SPOF and DPOF and $\boldsymbol{LL}^{H}$ or $\boldsymbol{U}^{H}\boldsymbol{U}$ for CPOF and ZPOF). To solve the system of equations with one or more right-hand sides, follow the call to these subroutines with a call to SPOSM, DPOSM, CPOSM, or ZPOSM. To find the inverse of matrix ***A***, follow the call to SPOF or DPOF with a call to SPOICD or DPOICD.

Table 93. Data Types

A	**Subroutine**
Short-precision real	SPPF and SPOF
Long-precision real	DPPF and DPOF
Short-precision complex	CPOF
Long-precision complex	ZPOF

Note: The output from SPPF and DPPF should be used only as input to the following subroutines for performing a solve or inverse: SPPS/SPPICD and DPPS/DPPICD, respectively. The output from SPOF, DPOF, CPOF, and ZPOF should be used only as input to the following subroutines for performing a solve or inverse: SPOSM/SPOICD, DPOSM/DPOICD, CPOSM, and ZPOSM, respectively.

Syntax

Fortran	CALL SPPF | DPPF (*ap*, *n*, *iopt*) CALL SPOF | DPOF | CPOF | ZPOF (*uplo*, *a*, *lda*, *n*)
C and C++	sppf | dppf (*ap*, *n*, *iopt*); spof | dpof | cpof | zpof (*uplo*, *a*, *lda*, *n*);
PL/I	CALL SPPF | DPPF (*ap*, *n*, *iopt*); CALL SPOF | DPOF | CPOF | ZPOF (*uplo*, *a*, *lda*, *n*);

On Entry

uplo
indicates whether matrix ***A*** is stored in upper or lower storage mode, where:

If *uplo* = 'U', ***A*** is stored in upper storage mode.

If *uplo* = 'L', ***A*** is stored in lower storage mode.

Specified as: a single character. It must be 'U' or 'L'.

ap
is array, referred to as AP, in which matrix ***A***, to be factored, is stored in lower-packed storage mode.

Specified as: a one-dimensional array, containing numbers of the data type indicated in Table 93 on page 494. See Notes.

If *iopt* = 0, the array must have at least $n(n+1)/2+n$ elements.

If *iopt* = 1, the array must have at least $n(n+1)/2$ elements.

a
is the positive definite matrix ***A***, to be factored.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 93 on page 494.

lda
is the leading dimension of the array specified for *a*.

Specified as: a fullword integer; $lda > 0$ and $lda \geq n$.

n
is the order *n* of matrix ***A***.

Specified as: a fullword integer; $n \geq 0$.

iopt
determines the type of computation to be performed, where:

If *iopt* = 0, the matrix is factored using the $\boldsymbol{LDL}^T$ method.

If *iopt* = 1, the matrix is factored using Cholesky factorization.

Specified as: a fullword integer; *iopt* = 0 or 1.

On Return

ap
is the transformed matrix ***A*** of order *n*, containing the results of the factorization. See "Notes" on page 496 and "Function" on page 496.

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 93 on page 494.

If *iopt* = 0, the array contains $n(n+1)/2+n$ elements.

If *iopt* = 1, the array contains $n(n+1)/2$ elements.

a
is the transformed matrix ***A*** of order *n*, containing the results of the factorization. See "Function" on page 496.

Returned as: a two-dimensional array, containing numbers of the data type indicated in Table 93 on page 494.

Notes

1. All subroutines accept lowercase letters for the *uplo* argument.
2. In the input and output arrays specified for *ap*, the first $n(n+1)/2$ elements are matrix elements. The additional n locations, required in the array when $iopt = 0$, are used for working storage by this subroutine and should not be altered between calls to the factorization and solve subroutines.
3. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, they are set to zero.
4. For a description of the storage modes used for the matrices, see:
   - For positive definite symmetric matrices, see "Positive Definite or Negative Definite Symmetric Matrix" on page 69.
   - For positive definite complex Hermitian matrices, see "Positive Definite or Negative Definite Complex Hermitian Matrix" on page 71.

Function: The functions for these subroutines are described in the sections below.

For SPPF and DPPF: If $iopt = 0$, the positive definite symmetric matrix ***A***, stored in lower-packed storage mode, is factored using Gaussian elimination, where ***A*** is expressed as:

$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^{T}$$

where:

L is a unit lower triangular matrix.
$\boldsymbol{L}^{T}$ is the transpose of matrix ***L***.
D is a diagonal matrix.

If $iopt = 1$, the positive definite symmetric matrix ***A*** is factored using Cholesky factorization, where ***A*** is expressed as:

$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^{T}$$

where ***L*** is a lower triangular matrix.

If n is 0, no computation is performed. See references [36] and [38].

For SPOF, DPOF, CPOF, and ZPOF: The positive definite matrix ***A***, stored in upper or lower storage mode, is factored using Cholesky factorization, where ***A*** is expressed as:

$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^{T}$ or $\boldsymbol{A} = \boldsymbol{U}^{T}\boldsymbol{U}$ for SPOF and DPOF
$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^{H}$ or $\boldsymbol{A} = \boldsymbol{U}^{H}\boldsymbol{U}$ for CPOF and ZPOF

where:

L is a lower triangular matrix.
$\boldsymbol{L}^{T}$ is the transpose of matrix ***L***.
$\boldsymbol{L}^{H}$ is the conjugate transpose of matrix ***L***.
U is an upper triangular matrix.
$\boldsymbol{U}^{T}$ is the transpose of matrix ***U***.
$\boldsymbol{U}^{H}$ is the conjugate transpose of matrix ***U***.

If *n* is 0, no computation is performed. See references [8], [64], and [36].

Error Conditions

Resource Errors: Unable to allocate internal work area.

Computational Errors

1. Matrix ***A*** is not positive definite (for SPPF and DPPF when *iopt* = 0).
   - Processing continues to the end of the matrix.
   - One or more elements of ***D*** contain values less than or equal to 0; all elements of ***D*** are checked. The index *i* of the **last** nonpositive element encountered is identified in the computational error message.
   - The return code is set to 1.
   - *i* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2104 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see "What Can You Do about ESSL Computational Errors?" on page 48.
2. Matrix ***A*** is not positive definite (for SPPF and DPPF when *iopt* = 1 and for SPOF, DPOF, CPOF, and ZPOF).
   - Processing stops at the first occurrence of a nonpositive definite diagonal element.
   - The order *i* of the **first** minor encountered having a nonpositive determinant is identified in the computational error message.
   - The return code is set to 1.
   - *i* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2115 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see "What Can You Do about ESSL Computational Errors?" on page 48.

Input-Argument Errors

1. $n < 0$
2. *iopt* $\neq$ 0 or 1
3. *uplo* $\neq$ 'U' or 'L'
4. *lda* $\leq$ 0
5. $n >$ *lda*

Example 1: This example shows a factorization of positive definite symmetric matrix ***A*** of order 9, stored in lower-packed storage mode, where on input matrix ***A*** is:

```
┌                                                   ┐
│ 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 │
│ 1.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0 │
│ 1.0  2.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0 │
│ 1.0  2.0  3.0  4.0  4.0  4.0  4.0  4.0  4.0 │
│ 1.0  2.0  3.0  4.0  5.0  5.0  5.0  5.0  5.0 │
│ 1.0  2.0  3.0  4.0  5.0  6.0  6.0  6.0  6.0 │
│ 1.0  2.0  3.0  4.0  5.0  6.0  7.0  7.0  7.0 │
│ 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  8.0 │
│ 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 │
└                                                   ┘
```

On output, all elements of this matrix ***A*** are 1.0.

Note: The AP arrays are formatted in a triangular arrangement for readability; however, they are stored in lower-packed storage mode.

Call Statement and Input

```
           AP  N  IOPT
           |   |   |
CALL SPPF( AP, 9,  0 )

AP = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0,
      3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0,
      4.0, 4.0, 4.0, 4.0, 4.0, 4.0,
      5.0, 5.0, 5.0, 5.0, 5.0,
      6.0, 6.0, 6.0, 6.0,
      7.0, 7.0, 7.0,
      8.0, 8.0,
      9.0,
       . ,  . ,  . ,  . ,  . ,  . ,  . ,  . ,  .  )
```

Output

```
AP = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0,
      1.0, 1.0,
      1.0,
      1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
```

Example 2: This example shows a factorization of the same positive definite symmetric matrix ***A*** of order 9 used in Example 1, stored in lower-packed storage mode.

Note: The AP arrays are formatted in a triangular arrangement for readability; however, they are stored in lower-packed storage mode.

Call Statement and Input

```
           AP  N  IOPT
           |   |   |
CALL SPPF( AP, 9,  1 )

AP = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0,
      3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0,
      4.0, 4.0, 4.0, 4.0, 4.0, 4.0,
      5.0, 5.0, 5.0, 5.0, 5.0,
      6.0, 6.0, 6.0, 6.0,
      7.0, 7.0, 7.0,
      8.0, 8.0,
      9.0)
```

Output

```
AP = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0, 1.0,
      1.0, 1.0, 1.0,
      1.0, 1.0,
      1.0)
```

Example 3: This example shows a factorization of the same positive definite symmetric matrix ***A*** of order 9 used in Example 1, but stored in lower storage mode.

Call Statement and Input

```
           UPLO  A  LDA  N
            |    |   |   |
CALL SPOF( 'L' , A , 9 , 9 )
```

```
        ⎡                                                   ⎤
        ⎢ 1.0   .    .    .    .    .    .    .    .    ⎥
        ⎢ 1.0  2.0   .    .    .    .    .    .    .    ⎥
        ⎢ 1.0  2.0  3.0   .    .    .    .    .    .    ⎥
        ⎢ 1.0  2.0  3.0  4.0   .    .    .    .    .    ⎥
A    =  ⎢ 1.0  2.0  3.0  4.0  5.0   .    .    .    .    ⎥
        ⎢ 1.0  2.0  3.0  4.0  5.0  6.0   .    .    .    ⎥
        ⎢ 1.0  2.0  3.0  4.0  5.0  6.0  7.0   .    .    ⎥
        ⎢ 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0   .    ⎥
        ⎢ 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0   ⎥
        ⎣                                                   ⎦
```

Output

```
        ⎡                                                   ⎤
        ⎢ 1.0   .    .    .    .    .    .    .    .    ⎥
        ⎢ 1.0  1.0   .    .    .    .    .    .    .    ⎥
        ⎢ 1.0  1.0  1.0   .    .    .    .    .    .    ⎥
        ⎢ 1.0  1.0  1.0  1.0   .    .    .    .    .    ⎥
A    =  ⎢ 1.0  1.0  1.0  1.0  1.0   .    .    .    .    ⎥
        ⎢ 1.0  1.0  1.0  1.0  1.0  1.0   .    .    .    ⎥
        ⎢ 1.0  1.0  1.0  1.0  1.0  1.0  1.0   .    .    ⎥
        ⎢ 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0   .    ⎥
        ⎢ 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0   ⎥
        ⎣                                                   ⎦
```

Example 4: This example shows a factorization of the same positive definite symmetric matrix ***A*** of order 9 used in Example 1, but stored in upper storage mode.

Call Statement and Input

```
           UPLO  A  LDA  N
            |    |   |   |
CALL SPOF( 'U' , A , 9 , 9 )
```

$$A = \begin{bmatrix} 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\ . & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 \\ . & . & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 \\ . & . & . & 4.0 & 4.0 & 4.0 & 4.0 & 4.0 & 4.0 \\ . & . & . & . & 5.0 & 5.0 & 5.0 & 5.0 & 5.0 \\ . & . & . & . & . & 6.0 & 6.0 & 6.0 & 6.0 \\ . & . & . & . & . & . & 7.0 & 7.0 & 7.0 \\ . & . & . & . & . & . & . & 8.0 & 8.0 \\ . & . & . & . & . & . & . & . & 9.0 \end{bmatrix}$$

Output

$$A = \begin{bmatrix} 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\ . & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\ . & . & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\ . & . & . & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\ . & . & . & . & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\ . & . & . & . & . & 1.0 & 1.0 & 1.0 & 1.0 \\ . & . & . & . & . & . & 1.0 & 1.0 & 1.0 \\ . & . & . & . & . & . & . & 1.0 & 1.0 \\ . & . & . & . & . & . & . & . & 1.0 \end{bmatrix}$$

Example 5: This example shows a factorization of positive definite complex Hermitian matrix ***A*** of order 3, stored in lower storage mode, where on input matrix ***A*** is:

$$\begin{bmatrix} (25.0, 0.0) & (-5.0, -5.0) & (10.0, 5.0) \\ (-5.0, 5.0) & (51.0, 0.0) & (4.0, -6.0) \\ (10.0, -5.0) & (4.0, 6.0) & (71.0, 0.0) \end{bmatrix}$$

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, they are set to zero.

Call Statement and Input

```
           UPLO  A  LDA  N
            |    |   |   |
CALL CPOF( 'L' , A , 3 , 3 )
```

$$A = \begin{bmatrix} (25.0, .) & . & . \\ (-5.0, 5.0) & (51.0, .) & . \\ (10.0, -5.0) & (4.0, 6.0) & (71.0, .) \end{bmatrix}$$

Output

```
        ⎡                                    ⎤
        ⎢  (5.0, 0.0)      .           .      ⎥
A    =  ⎢ (-1.0, 1.0)  (7.0, 0.0)      .      ⎥
        ⎢ (2.0, -1.0)  (1.0, 1.0)  (8.0, 0.0) ⎥
        ⎣                                    ⎦
```

Example 6: This example shows a factorization of positive definite complex Hermitian matrix ***A*** of order 3, stored in upper storage mode, where on input matrix ***A*** is:

```
⎡                                          ⎤
⎢  (9.0, 0.0)  (3.0, 3.0) (3.0, -3.0) ⎥
⎢ (3.0, -3.0) (18.0, 0.0) (8.0, -6.0) ⎥
⎢  (3.0, 3.0)  (8.0, 6.0) (43.0, 0.0) ⎥
⎣                                          ⎦
```

Note: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values. On output, they are set to zero.

Call Statement and Input

```
           UPLO  A  LDA  N
            |    |   |   |
CALL CPOF( 'U' , A , 3 , 3 )
```

```
        ⎡                                    ⎤
        ⎢ (9.0,  . )   (3.0,3.0)   (3.0,-3.0) ⎥
A    =  ⎢     .       (18.0, . )   (8.0,-6.0) ⎥
        ⎢     .            .      (43.0,  . ) ⎥
        ⎣                                    ⎦
```

Output

```
        ⎡                                    ⎤
        ⎢ (3.0, 0.0)    (1.0, 1.0)  (1.0, -1.0) ⎥
A    =  ⎢     .         (4.0, 0.0)  (2.0, -1.0) ⎥
        ⎢     .            .         (6.0, 0.0) ⎥
        ⎣                                    ⎦
```

SPPS and DPPS—Positive Definite Real Symmetric Matrix Solve

These subroutines solve the system $\boldsymbol{Ax} = \boldsymbol{b}$ for $\boldsymbol{x}$, where $\boldsymbol{A}$ is a positive definite symmetric matrix, and $\boldsymbol{x}$ and $\boldsymbol{b}$ are vectors. The subroutines use the results of the factorization of matrix $\boldsymbol{A}$, produced by a preceding call to SPPF/SPPFCD or DPPF/DPPFP/DPPFCD, respectively.

Table 94. Data Types

A, ***b***, ***x***	**Subroutine**
Short-precision real	SPPS
Long-precision real	DPPS

Note: The input to these solve subroutines must be the output from the factorization subroutines SPPF/SPPFCD and DPPF/DPPFP/DPPFCD, respectively.

Syntax

Fortran	CALL SPPS | DPPS (*ap*, *n*, *bx*, *iopt*)
C and C++	spps | dpps (*ap*, *n*, *bx*, *iopt*);
PL/I	CALL SPPS | DPPS (*ap*, *n*, *bx*, *iopt*);

On Entry

ap
is the factorization of matrix $\boldsymbol{A}$, produced by a preceding call to SPPF/SPPFCD or DPPF/DPPFP/DPPFCD, respectively. Specified as: a one-dimensional array, containing numbers of the data type indicated in Table 94, where:

If *iopt* = 0, the array must contain $n(n+1)/2+n$ elements.

If *iopt* = 1, the array must contain $n(n+1)/2$ elements.

n
is the order of matrix $\boldsymbol{A}$ used in the factorization, and the lengths of vectors $\boldsymbol{b}$ and $\boldsymbol{x}$. Specified as: a fullword integer; $n \geq 0$.

bx
is the vector $\boldsymbol{b}$ of length *n*, containing the right-hand side of the system. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 94.

iopt
indicates the type of factorization that was performed on matrix $\boldsymbol{A}$, where:

If *iopt* = 0, the matrix was factored using the $\boldsymbol{LDL}^{\mathrm{T}}$ method.

If *iopt* = 1, the matrix was factored using Cholesky factorization.

Specified as: a fullword integer; *iopt* = 0 or 1.

On Return

bx
is the solution vector ***x*** of length *n*, containing the results of the computation. Specified as: a one-dimensional array, containing numbers of the data type indicated in Table 94 on page 502.

Notes

1. The array data specified for input argument *ap* for these subroutines must be the same as the corresponding output argument for SPPF/SPPFCD and DPPF/DPPFP/DPPFCD, respectively.
2. The scalar data specified for input argument *n* for these subroutines must be the same as that specified for SPPF/SPPFCD and DPPF/DPPFP/DPPFCD, respectively.
3. When you call these subroutines after calling SPPF or DPPF, the value of input argument *iopt* must be the same as that specified for SPPF and DPPF.
4. When you call these subroutines after calling SPPFCD or DPPFCD, the value of input argument *iopt* must be 0.
5. When you call these subroutines after calling DPPFP, the value of input argument *iopt* must be 1.
6. In the input array specified for *ap*, the first $n(n+1)/2$ elements are matrix elements. The additional *n* locations, required in the array when $iopt = 0$, are used for working storage by this subroutine and should not be altered between calls to the factorization and solve subroutines.
7. The vectors and matrices used in this computation must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.
8. For a description of how a positive definite symmetric matrix is stored in lower-packed storage mode in an array, see "Symmetric Matrix" on page 65.

Function: The system $\boldsymbol{Ax} = \boldsymbol{b}$ is solved for $\boldsymbol{x}$, where $\boldsymbol{A}$ is a positive definite symmetric matrix, stored in lower-packed storage mode in array AP, and $\boldsymbol{x}$ and $\boldsymbol{b}$ are vectors. These subroutines use the results of the factorization of matrix $\boldsymbol{A}$, produced by a preceding call to SPPF/SPPFCD or DPPF/DPPFP/DPPFCD, respectively.

If *n* is 0, no computation is performed. See references [36] and [38].

Error Conditions

Computational Errors: None

Note: If a call to SPPF, DPPF, SPPFCD, DPPFCD, or DPPFP resulted in a nonpositive definite matrix, error 2104 or 2115, SPPS or DPPS results may be unpredictable or numerically unstable.

Input-Argument Errors

1. $n < 0$
2. $iopt \neq 0$ or 1

Example 1: This example shows how to solve the system $\boldsymbol{Ax} = \boldsymbol{b}$, where matrix $\boldsymbol{A}$ is the same matrix factored in the "Example 1" on page 497 for SPPF and DPPF.

Call Statement and Input

```
           AP   N   BX   IOPT
           |    |   |     |
CALL SPPS ( AP , 9 , BX ,  0 )

AP       =(same as output AP in "Example 1" on page 497 for SPPF and DPPF)
BX       =  (9.0, 17.0, 24.0, 30.0, 35.0, 39.0, 42.0, 44.0, 45.0)
```

Output

```
BX       =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
```

Example 2: This example shows how to solve the same system as in Example 1, where matrix ***A*** is the same matrix factored in the "Example 2" on page 498 for SPPF and DPPF.

Call Statement and Input

```
           AP   N   BX  IOPT
           |    |   |    |
CALL SPPS( AP , 9 , BX , 1 )

AP       =(same as output AP in "Example 2" on page 498 for SPPF and DPPF)
BX       =  (9.0, 17.0, 24.0, 30.0, 35.0, 39.0, 42.0, 44.0, 45.0)
```

Output

```
BX       =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
```

SPOSM, DPOSM, CPOSM, and ZPOSM—Positive Definite Real Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve

These subroutines solve the system $AX = B$ for X, using multiple right-hand sides, where X and B are general matrices and:

- For SPOSM and DPOSM, A is a positive definite symmetric matrix.
- For CPOSM and ZPOSM, A is a positive definite complex Hermitian matrix.

These subroutines use the results of the factorization of matrix A, produced by a preceding call to SPOF/SPOFCD, DPOF/DPOFCD, CPOF, or ZPOF, respectively.

Table 95. Data Types

A, *B*, *X*	Subroutine
Short-precision real	SPOSM
Long-precision real	DPOSM
Short-precision complex	CPOSM
Long-precision complex	ZPOSM

Note: The input to these solve subroutines must be the output from the factorization subroutines SPOF/SPOFCD, DPOF/DPOFCD, CPOF, and ZPOF, respectively.

Syntax

Fortran	CALL SPOSM | DPOSM | CPOSM | ZPOSM (*uplo*, *a*, *lda*, *n*, *b*, *ldb*, *nrhs*)
C and C++	sposm | dposm | cposm | zposm (*uplo*, *a*, *lda*, *n*, *b*, *ldb*, *nrhs*);
PL/I	CALL SPOSM | DPOSM | CPOSM | ZPOSM (*uplo*, *a*, *lda*, *n*, *b*, *ldb*, *nrhs*);

On Entry

uplo
indicates whether the original matrix A is stored in upper or lower storage mode, where:

If *uplo* = 'U', A is stored in upper storage mode.

If *uplo* = 'L', A is stored in lower storage mode.

Specified as: a single character. It must be 'U' or 'L'.

a
is the factorization of positive definite matrix A, produced by a preceding call to SPOF/SPOFCD, DPOF/DPOFCD, CPOF, or ZPOF, respectively. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 95.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and $lda \geq n$.

n
is the order of matrix A. Specified as: a fullword integer; $0 \leq n \leq lda$.

b
is the matrix ***B***, containing the *nrhs* right-hand sides of the system. The right-hand sides, each of length *n*, reside in the columns of matrix ***B***. Specified as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 95.

ldb
is the leading dimension of the array specified for *b*. Specified as: a fullword integer; $ldb > 0$ and $ldb \geq n$.

nrhs
is the number of right-hand sides in the system to be solved. Specified as: a fullword integer; $nrhs \geq 0$.

On Return

b
is the matrix ***B***, containing the *nrhs* solutions to the system in the columns of ***B***. Specified as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in Table 95 on page 505.

Notes

1. All subroutines accept lowercase letters for the *uplo* argument.
2. The scalar data specified for input arguments *uplo*, *lda*, and *n* for these subroutines must be the same as the corresponding input arguments specified for SPOF/SPOFCD, DPOF/DPOFCD, CPOF, and ZPOF, respectively.
3. The array data specified for input argument *a* for these subroutines must be the same as the corresponding output arguments for SPOF/SPOFCD, DPOF/DPOFCD, CPOF, and ZPOF, respectively.
4. The vectors and matrices used in this computation must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.
5. For a description of how the matrices are stored:
   - For positive definite symmetric matrices, see "Positive Definite or Negative Definite Symmetric Matrix" on page 69.
   - For positive definite complex Hermitian matrices, see "Positive Definite or Negative Definite Complex Hermitian Matrix" on page 71.

Function: The system $\boldsymbol{AX} = \boldsymbol{B}$ is solved for $\boldsymbol{X}$, using multiple right-hand sides, where $\boldsymbol{X}$ and $\boldsymbol{B}$ are general matrices, and $\boldsymbol{A}$ is a positive definite symmetric matrix for SPOSM and DPOSM and a positive definite complex Hermitian matrix for CPOSM and ZPOSM. These subroutines use the results of the factorization of matrix $\boldsymbol{A}$, produced by a preceding call to SPOF/SPOFCD, DPOF/DPOFCD, CPOF, or ZPOF, respectively. For a description of how $\boldsymbol{A}$ is factored, see "SPPF, DPPF, SPOF, DPOF, CPOF, and ZPOF—Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization" on page 494.

If *n* or *nrhs* is 0, no computation is performed. See references [8] and [36].

Error Conditions

Computational Errors: None

Note: If the factorization performed by SPOF, DPOF, CPOF, ZPOF, SPOFCD, or DPOFCD failed because matrix $\boldsymbol{A}$ was not positive definite, the results returned

by this subroutine are unpredictable, and there may be a divide-by-zero program exception message.

Input-Argument Errors

1. *uplo* ≠ 'U' or 'L'
2. *lda*, *ldb* ≤ 0
3. *n* < 0
4. *n* > *lda*
5. *n* > *ldb*
6. *nrhs* < 0

Example 1: This example shows how to solve the system ***AX*** = ***B*** for two right-hand sides, where matrix ***A*** is the same matrix factored in the "Example 3" on page 499 for SPOF.

Call Statement and Input

```
            UPLO  A  LDA  N   B  LDB  NRHS
             |    |   |   |   |   |    |
CALL SPOSM( 'L' , A , 9 , 9 , B , 9 ,  2  )
```

A =(same as output A in "Example 3" on page 499)

```
        ⎡              ⎤
        ⎢  9.0   45.0  ⎥
        ⎢ 17.0   89.0  ⎥
        ⎢ 24.0  131.0  ⎥
        ⎢ 30.0  170.0  ⎥
B    =  ⎢ 35.0  205.0  ⎥
        ⎢ 39.0  235.0  ⎥
        ⎢ 42.0  259.0  ⎥
        ⎢ 44.0  276.0  ⎥
        ⎢ 45.0  285.0  ⎥
        ⎣              ⎦
```

Output

```
        ⎡          ⎤
        ⎢ 1.0  1.0 ⎥
        ⎢ 1.0  2.0 ⎥
        ⎢ 1.0  3.0 ⎥
        ⎢ 1.0  4.0 ⎥
B    =  ⎢ 1.0  5.0 ⎥
        ⎢ 1.0  6.0 ⎥
        ⎢ 1.0  7.0 ⎥
        ⎢ 1.0  8.0 ⎥
        ⎢ 1.0  9.0 ⎥
        ⎣          ⎦
```

Example 2: This example shows how to solve the system $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{X} = \boldsymbol{B}$ for two right-hand sides, where matrix $\boldsymbol{A}$ is the input matrix factored in "Example 4" on page 500 for SPOF.

Call Statement and Input

```
            UPLO  A  LDA  N   B  LDB  NRHS
             |    |   |   |   |   |    |
CALL SPOSM( 'U' , A , 9 , 9 , B , 9 ,  2  )
```

A =(same as output A in "Example 4" on page 500)

```
        ⎡              ⎤
        ⎢  9.0   45.0  ⎥
        ⎢ 17.0   89.0  ⎥
        ⎢ 24.0  131.0  ⎥
        ⎢ 30.0  170.0  ⎥
B    =  ⎢ 35.0  205.0  ⎥
        ⎢ 39.0  235.0  ⎥
        ⎢ 42.0  259.0  ⎥
        ⎢ 44.0  276.0  ⎥
        ⎢ 45.0  285.0  ⎥
        ⎣              ⎦
```

Output

```
        ⎡          ⎤
        ⎢ 1.0  1.0 ⎥
        ⎢ 1.0  2.0 ⎥
        ⎢ 1.0  3.0 ⎥
        ⎢ 1.0  4.0 ⎥
B    =  ⎢ 1.0  5.0 ⎥
        ⎢ 1.0  6.0 ⎥
        ⎢ 1.0  7.0 ⎥
        ⎢ 1.0  8.0 ⎥
        ⎢ 1.0  9.0 ⎥
        ⎣          ⎦
```

Example 3: This example shows how to solve the system $\boldsymbol{AX} = \boldsymbol{B}$ for two right-hand sides, where matrix $\boldsymbol{A}$ is the same matrix factored in the "Example 5" on page 500 for CPOF.

Call Statement and Input

```
            UPLO  A  LDA  N   B  LDB  NRHS
             |    |   |   |   |   |    |
CALL CPOSM( 'L' , A , 3 , 3 , B , 3 ,  2  )
```

A =(same as output A in "Example 5" on page 500)

```
        ⎡                                  ⎤
        ⎢  (60.0, -55.0)    (70.0, 10.0)   ⎥
B    =  ⎢   (34.0, 58.0)  (-51.0, 110.0)   ⎥
        ⎢ (13.0, -152.0)    (75.0, 63.0)   ⎥
        ⎣                                  ⎦
```

Output

```
        ⎡                          ⎤
        ⎢ (2.0, -1.0)   (2.0, 0.0) ⎥
B    =  ⎢  (1.0, 1.0)  (-1.0, 2.0) ⎥
        ⎢ (0.0, -2.0)   (1.0, 1.0) ⎥
        ⎣                          ⎦
```

Example 4: This example shows how to solve the system ***AX = B*** for two right-hand sides, where matrix ***A*** is the input matrix factored in "Example 6" on page 501 for CPOF.

Call Statement and Input

```
            UPLO  A  LDA  N   B  LDB  NRHS
             |    |   |   |   |   |    |
CALL CPOSM( 'U' , A , 3 , 3 , B , 3 ,  2  )
```

```
A          =(same as output A in "Example 6" on page 501)
```

```
        ⎡                           ⎤
        ⎢ (33.0, -18.0)   (15.0, -3.0) ⎥
B    =  ⎢ (45.0, -45.0)    (8.0, -2.0) ⎥
        ⎢  (152.0, 1.0)  (43.0, -29.0) ⎥
        ⎣                           ⎦
```

Output

```
        ⎡                          ⎤
        ⎢ (2.0, -1.0)   (2.0, 0.0) ⎥
B    =  ⎢ (1.0, -1.0)   (0.0, 1.0) ⎥
        ⎢  (3.0, 0.0)  (1.0, -1.0) ⎥
        ⎣                          ⎦
```

SPPFCD, DPPFCD, SPOFCD, and DPOFCD—Positive Definite Real Symmetric Matrix Factorization, Condition Number Reciprocal, and Determinant

The SPPFCD and DPPFCD subroutines factor positive definite symmetric matrix ***A***, stored in lower-packed storage mode, using Gaussian elimination (***LDL***^{T}). The reciprocal of the condition number and the determinant of matrix ***A*** can also be computed. To solve the system of equations with one or more right-hand sides, follow the call to these subroutines with one or more calls to SPPS or DPPS, respectively.

The SPOFCD and DPOFCD subroutines factor positive definite symmetric matrix ***A***, stored in upper or lower storage mode, using Cholesky factorization (***LL***^{T} or ***U***^{T}***U***). The reciprocal of the condition number and the determinant of matrix ***A*** can also be computed. To solve the system of equations with one or more right-hand sides, follow the call to these subroutines with a call to SPOSM or DPOSM, respectively. To find the inverse of matrix ***A***, follow the call to these subroutines with a call to SPOICD or DPOICD, respectively.

Table 96. Data Types

A, *aux*, *rcond*, ***det***	**Subroutine**
Short-precision real	SPPFCD and SPOFCD
Long-precision real	DPPFCD and DPOFCD

Note: The output factorization from SPPFCD and DPPFCD should be used only as input to the solve subroutines SPPS and DPPS, respectively. The output from SPOFCD and DPOFCD should be used only as input to the following subroutines for performing a solve or inverse: SPOSM/SPOICD and DPOSM/DPOICD, respectively.

Syntax

Fortran	CALL SPPFCD | DPPFCD (*ap*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*) CALL SPOFCD | DPOFCD (*uplo*, *a*, *lda*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*)
C and C++	sppfcd | dppfcd (*ap*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*); spofcd | dpofcd (*uplo*, *a*, *lda*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*);
PL/I	CALL SPPFCD | DPPFCD (*ap*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*); CALL SPOFCD | DPOFCD (*uplo*, *a*, *lda*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*);

On Entry

uplo

indicates whether matrix ***A*** is stored in upper or lower storage mode, where:

If *uplo* = 'U', ***A*** is stored in upper storage mode.

If *uplo* = 'L', ***A*** is stored in lower storage mode.

Specified as: a single character. It must be 'U' or 'L'.

ap
is the array, referred to as AP, in which the matrix ***A***, to be factored, is stored in lower-packed storage mode. Specified as: a one-dimensional array of (at least) length $n(n+1)/2+n$, containing numbers of the data type indicated in Table 96.

a
is the positive definite symmetric matrix ***A***, to be factored. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 96 on page 510.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and $lda \geq n$.

n
is the order *n* of matrix ***A***. Specified as: a fullword integer, where:

For SPPFCD and DPPFCD, $n \geq 0$.

For SPOFCD and DPOFCD, $0 \leq n \leq lda$.

iopt
indicates the type of computation to be performed, where:

If $iopt = 0$, the matrix is factored.

If $iopt = 1$, the matrix is factored, and the reciprocal of the condition number is computed.

If $iopt = 2$, the matrix is factored, and the determinant is computed.

If $iopt = 3$, the matrix is factored and the reciprocal of the condition number and the determinant are computed.

Specified as: a fullword integer; $iopt = 0$, 1, 2, or 3.

rcond
See On Return.

det
See On Return.

aux
has the following meaning:

If $naux = 0$ and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, is the storage work area used by these subroutines. Its size is specified by *naux*. Specified as: an area of storage, containing numbers of the data type indicated in Table 96 on page 510.

naux
is the size of the work area specified by *aux*—that is, the number of elements in *aux*. Specified as: a fullword integer, where:

If $naux = 0$ and error 2015 is unrecoverable, SPPFCD, DPPFCD, SPOFCD, and DPOFCD dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, $naux \geq n$.

On Return

ap
is the transformed matrix ***A*** of order *n*, containing the results of the factorization. See "Function" on page 512. Returned as: a one-dimensional array of (at least) length $n(n+1)/2+n$, containing numbers of the data type indicated in Table 96 on page 510.

a
is the transformed matrix ***A*** of order *n*, containing the results of the factorization. See "Function" on page 512. Returned as: a two-dimensional array, containing numbers of the data type indicated in Table 96 on page 510.

rcond
is the estimate of the reciprocal of the condition number, *rcond*, of matrix ***A***. Returned as: a number of the data type indicated in Table 96 on page 510; $rcond \geq 0$.

det
is the vector ***det***, containing the two components det_1 and det_2 of the determinant of matrix ***A***. The determinant is:

$$det_1\left(10^{det_2}\right)$$

where $1 \leq det_1 < 10$. Returned as: an array of length 2, containing numbers of the data type indicated in Table 96 on page 510.

Notes

1. All subroutines accept lowercase letters for the *uplo* argument.
2. In your C program, argument *rcond* must be passed by reference.
3. When *iopt* = 0, SPPFCD and DPPFCD provide the same function as a call to SPPF or DPPF, respectively. When *iopt* = 0, SPOFCD and DPOFCD provide the same function as a call to SPOF or DPOF, respectively.
4. See "Notes" on page 503 for information on specifying a value for *iopt* in the SPPS and DPPS subroutines after calling SPPFCD and DPPFCD, respectively.
5. In the input and output arrays specified for *ap*, the first $n(n+1)/2$ elements are matrix elements. The additional *n* locations in the array are used for working storage by this subroutine and should not be altered between calls to the factorization and solve subroutines.
6. For a description of how a positive definite symmetric matrix is stored in lower-packed storage mode in an array, see "Symmetric Matrix" on page 65. For a description of how a positive definite symmetric matrix is stored in upper or lower storage mode, see "Positive Definite or Negative Definite Symmetric Matrix" on page 69.
7. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: The functions for these subroutines are described in the sections below.

For SPPFCD and DPPFCD: The positive definite symmetric matrix ***A***, stored in lower-packed storage mode, is factored using Gaussian elimination, where ***A*** is expressed as:

$$\boldsymbol{A} = \boldsymbol{LDL}^{\mathrm{T}}$$

.

where:

L is a unit lower triangular matrix.
$\boldsymbol{L}^{T}$ is the transpose of matrix ***L***.
D is a diagonal matrix.

An estimate of the reciprocal of the condition number, *rcond*, and the determinant, ***det***, can also be computed by this subroutine. The estimate of the condition number uses an enhanced version of the algorithm described in references [63] and [64].

If *n* is 0, no computation is performed. See references [36] and [38].

These subroutines call SPPF and DPPF, respectively, to perform the factorization using Gaussian elimination ($\boldsymbol{LDL}^{T}$). If you want to use the Cholesky factorization method, you must call SPPF and DPPF directly.

For SPOFCD and DPOFCD: The positive definite symmetric matrix ***A***, stored in upper or lower storage mode, is factored using Cholesky factorization, where ***A*** is expressed as:

$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^{T} \text{ or } \boldsymbol{A} = \boldsymbol{U}^{T}\boldsymbol{U}$$

where:

L is a lower triangular matrix.
$\boldsymbol{L}^{T}$ is the transpose of matrix ***L***.
U is an upper triangular matrix.
$\boldsymbol{U}^{T}$ is the transpose of matrix ***U***.

If specified, the estimate of the reciprocal of the condition number and the determinant can also be computed. The estimate of the condition number uses an enhanced version of the algorithm described in references [63] and [64].

If *n* is 0, no computation is performed. See references [8] and [36].

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors

1. Matrix ***A*** is not positive definite (for SPPFCD and DPPFCD).
   - If matrix ***A*** is singular (at least one of the diagonal elements are 0), then *rcond* and *det*, if you requested them, are set to 0.
   - If matrix ***A*** is nonsingular and nonpositive definite (none of the diagonal elements are 0 and at least one diagonal element is negative), then *rcond* and ***det***, if you requested them, are computed.
   - One or more elements of ***D*** contain values less than or equal to 0; all elements of ***D*** are checked. The index *i* of the last nonpositive element encountered is identified in the computational error message, issued by SPPF or DPPF, respectively.
   - *i* can be determined at run time by using the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2104 in the ESSL error option table; otherwise, the default value causes your program to be terminated by SPPF or DPPF, respectively, when this error occurs. If your program is not terminated by SPPF

or DPPF, respectively, the return code is set to 2. For details, see "What Can You Do about ESSL Computational Errors?" on page 48.

2. Matrix ***A*** is not positive definite (for SPOFCD and DPOFCD).
   - If matrix ***A*** is singular (at least one of the diagonal elements are 0), then *rcond* and *det*, if you requested them, are set to 0.
   - If matrix ***A*** is nonsingular and nonpositive definite (none of the diagonal elements are 0 and at least one diagonal element is negative), then *rcond* and ***det***, if you requested them, are computed.
   - Processing stops at the first occurrence of a nonpositive definite diagonal element.
   - The order *i* of the **first** minor encountered having a nonpositive determinant is identified in the computational error message.
   - *i* can be determined at run time by using the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2115 in the ESSL error option table; otherwise, the default value causes your program to be terminated by SPPF or DPPF, respectively, when this error occurs. If your program is not terminated by SPPF or DPPF, respectively, the return code is set to 2. For details, see "What Can You Do about ESSL Computational Errors?" on page 48.

Input-Argument Errors

1. *uplo* ≠ 'U' or 'L'
2. *lda* ≤ 0
3. *lda* < *n*
4. *n* < 0
5. *iopt* ≠ 0, 1, 2, or 3
6. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example 1: This example computes the factorization, reciprocal of the condition number, and determinant of matrix ***A***. The input is the same as used in "Example 1" on page 497 for SPPF.

The values used to estimate the reciprocal of the condition number are obtained with the following values:

$\|A\|_1$ = max(9.0, 17.0, 24.0, 30.0, 35.0, 39.0, 42.0, 44.0, 45.0) = 45.0
Estimate of $\|A\|$ = 4.0

On output, the value in ***det***, $|A|$, is equal to 1.

Call Statement and Input

```
             AP   N  IOPT  RCOND   DET   AUX  NAUX
             |    |   |      |      |     |    |
CALL DPPFCD( AP , 9 , 3  , RCOND , DET , AUX , 9  )
```

```
AP       =(same as input AP in "Example 1" on page 497)
```

Output

```
AP       =(same as output AP in "Example 1" on page 497)
RCOND    =  0.0055555
DET      =  (1.0, 0.0)
```

Example 2: This example computes the factorization, reciprocal of the condition number, and determinant of matrix ***A***. The input is the same as used in "Example 3" on page 499 for SPOF.

The values used to estimate the reciprocal of the condition number are obtained with the following values:

$\|A\|_1$ = max(9.0, 17.0, 24.0, 30.0, 35.0, 39.0, 42.0, 44.0, 45.0) = 45.0
Estimate of $\|A\|$ = 4.0

On output, the value in ***det***, $|A|$, is equal to 1.

Call Statement and Input

```
             UPLO A  LDA  N IOPT  RCOND   DET   AUX  NAUX
              |   |   |   |   |     |      |     |    |
CALL SPOFCD( 'L', A , 9 , 9 , 3 , RCOND , DET , AUX , 9  )
```

```
A         =(same as input A in "Example 3" on page 499)
```

Output

```
A         =(same as output A in "Example 3" on page 499)
RCOND     =  0.0055555
DET       =  (1.0, 0.0)
```

Example 3: This example computes the factorization, reciprocal of the condition number, and determinant of matrix ***A***. The input is the same as used in "Example 4" on page 500 for SPOF.

The values used to estimate the reciprocal of the condition number are obtained with the following values:

$\|A\|_1$ = max(9.0, 17.0, 24.0, 30.0, 35.0, 39.0, 42.0, 44.0, 45.0) = 45.0
Estimate of $\|A\|$ = 4.0

On output, the value in ***det***, $|A|$, is equal to 1.

Call Statement and Input

```
             UPLO A  LDA  N IOPT  RCOND   DET   AUX  NAUX
              |   |   |   |   |     |      |     |    |
CALL SPOFCD( 'U', A , 9 , 9 , 3 , RCOND , DET , AUX , 9  )
```

```
A         =(same as input A in "Example 4" on page 500)
```

Output

```
A         =(same as output A in "Example 4" on page 500)
RCOND     =  0.0055555
DET       =  (1.0, 0.0)
```

SGEICD and DGEICD—General Matrix Inverse, Condition Number Reciprocal, and Determinant

These subroutines find the inverse, the reciprocal of the condition number, and the determinant of matrix ***A***.

Table 97. Data Types

A, *aux*, *rcond*, ***det***	**Subroutine**
Short-precision real	SGEICD
Long-precision real	DGEICD

Note: If you call these subroutines with *iopt* = 0, 1, 2, or 3 the input must be the output from the factorization subroutines SGEF/SGEFCD/SGETRF or DGEF/DGEFCD/DGEFP/DGETRF, respectively.

Syntax

Fortran	CALL SGEICD | DGEICD (*a*, *lda*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*)
C and C++	sgeicd | dgeicd (*a*, *lda*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*);
PL/I	CALL SGEICD | DGEICD (*a*, *lda*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*);

On Entry

a
has the following meaning, where:

If *iopt* = 0, 1, 2, or 3, it is matrix ***A*** of order *n*, whose inverse, reciprocal of condition number, and determinant are computed.

If *iopt* = 4, it is the transformed matrix ***A*** of order *n*, resulting from the factorization performed in a previous call to SGEF/SGEFCD or DGEF/DGEFCD/DGEFP, respectively, whose inverse is computed.

Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 97.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and $lda \geq n$.

n
is the order of matrix ***A***. Specified as: a fullword integer; $0 \leq n \leq lda$.

iopt
indicates the type of computation to be performed, where:

If *iopt* = 0, the inverse is computed for matrix ***A***.

If *iopt* = 1, the inverse and the reciprocal of the condition number are computed for matrix ***A***.

If *iopt* = 2, the inverse and the determinant are computed for matrix ***A***.

If *iopt* = 3, the inverse, the reciprocal of the condition number, and the determinant are computed for matrix ***A***.

If *iopt* = 4, the inverse is computed using the factored matrix ***A***.

Specified as: a fullword integer; *iopt* = 0, 1, 2, 3, 4.

rcond

See On Return.

det

See On Return.

aux

has the following meaning, and its size is specified by *naux*:

If *iopt* = 0, 1, 2, or 3, then if *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored. Otherwise, it is the storage work area used by this subroutine.

If *iopt* = 4, *aux* has the following meaning:

- For SGEICD, the first *n* locations in *aux* must contain the ***ipvt*** integer vector of length *n*, resulting from a previous call to SGEF, SGETRF, or SGEFCD.
- For DGEICD, the first ceiling(*n*/2) locations in *aux* must contain the ***ipvt*** integer vector of length *n*, resulting from a previous call to DGEF, DGETRF, DGEFCD, or DGEFP.

Specified as: an area of storage, containing numbers of the data type indicated in Table 97 on page 516.

naux

is the size of the work area specified by *aux*—that is, the number of elements in *aux*. Specified as: a fullword integer, where:

If *iopt* ≠ 4, then if *naux* = 0 and error 2015 is unrecoverable, SGEICD and DGEICD dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise *naux* must have the following value:

For the RS/6000 POWER or PowerPC processors, *naux* ≥ 100*n*.

For the RS/6000 POWER2 processors, *naux* ≥ 200*n*.

Note: *naux* values specified for releases prior to ESSL Version 2 Release 2 will still work, but you may not achieve optimal performance.

On Return

a

is the resulting inverse of matrix ***A*** of order *n*. Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 97 on page 516.

rcond

is the reciprocal of the condition number, *rcond*, of matrix ***A***. Returned as: a real number of the data type indicated in Table 97 on page 516; *rcond* ≥ 0.

det

is the vector ***det***, containing the two components det_1 and det_2 of the determinant of matrix ***A***. The determinant is:

$$det_1\left(10^{det_2}\right)$$

where $1 \leq det_1 < 10$. Returned as: an array of length 2, containing numbers of the data type indicated in Table 97 on page 516.

Notes

1. In your C program, argument *rcond* must be passed by reference.
2. If *iopt* = 4, the following input arguments for SGEICD and DGEICD must be set to the same values in the previous call to SGEF/SGEFCD or DGEF/DGEFCD/DGEFP, respectively:

For _GEF_	For _GEICD
Input arguments *n* and *lda*	Input arguments *n* and *lda*
Output arguments *a* and *ipvt*	Input arguments *a* and *aux*

3. You have the option of having the value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: The inverse, the reciprocal of the condition number, and the determinant of a general square matrix $\boldsymbol{A}$ are computed using partial pivoting to preserve accuracy, where:

- $\boldsymbol{A}^{-1}$ is the inverse of matrix $\boldsymbol{A}$, where $\boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{A}^{-1}\boldsymbol{A} = \boldsymbol{I}$, and $\boldsymbol{I}$ is the identity matrix.
- $1/(\|\boldsymbol{A}\|_1)(\|\boldsymbol{A}^{-1}\|_1)$ is the reciprocal of the condition number, where $\|\boldsymbol{A}\|_1$ is the one-norm of matrix $\boldsymbol{A}$.
- $|\boldsymbol{A}|$ is the determinant of matrix $\boldsymbol{A}$, where $|\boldsymbol{A}|$ is expressed as:

$$det_1\left(10^{det_2}\right)$$

The *iopt* argument is used to determine the combination of output items produced by this subroutine: the inverse, the reciprocal of the condition number, and the determinant.

If *n* is 0, no computation is performed. See references [36], [38], and [44].

Error Conditions

Resource Errors: If *iopt* = 0, 1, 2, or 3, then error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors: Matrix $\boldsymbol{A}$ is singular or nearly singular.

- The index *i* of the first pivot element having a value equal to 0, is identified in the computational error message.
- These subroutines return 0 for *rcond* and ***det***, if you requested them.
- The return code is set to 2.
- *i* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2105 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see "What Can You Do about ESSL Computational Errors?" on page 48.

Input-Argument Errors

1. $lda \leq 0$
2. $n < 0$

3. $n > lda$
4. $iopt \neq 0, 1, 2, 3,$ or 4
5. Error 2015 is recoverable or $naux \neq 0$, and $naux$ is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example 1: This example computes the inverse, the reciprocal of the condition number, and the determinant of matrix $\boldsymbol{A}$. The values used to compute the reciprocal of the condition number in this example are obtained with the following values:

$$\|\boldsymbol{A}\|_1 = \max(6.0, 8.0, 10.0, 12.0, 13.0, 14.0, 15.0, 15.0, 15.0) = 15.0$$
$$\|\boldsymbol{A}^{-1}\|_1 = 1226.33$$

On output, the value in ***det***, $|\boldsymbol{A}|$, is equal to 336.

Call Statement and Input

```
             A  LDA  N  IOPT  RCOND   DET   AUX   NAUX
             |   |   |   |      |      |     |     |
CALL DGEICD( A , 9 , 9 , 3  , RCOND , DET , AUX , 293 )
```

```
        ⎡                                                    ⎤
        ⎢ 1.0  1.0  1.0  1.0  0.0  0.0   0.0   0.0   0.0 ⎥
        ⎢ 1.0  1.0  1.0  1.0  1.0  0.0   0.0   0.0   0.0 ⎥
        ⎢ 4.0  1.0  1.0  1.0  1.0  1.0   0.0   0.0   0.0 ⎥
        ⎢ 0.0  5.0  1.0  1.0  1.0  1.0   1.0   0.0   0.0 ⎥
A    =  ⎢ 0.0  0.0  6.0  1.0  1.0  1.0   1.0   1.0   0.0 ⎥
        ⎢ 0.0  0.0  0.0  7.0  1.0  1.0   1.0   1.0   1.0 ⎥
        ⎢ 0.0  0.0  0.0  0.0  8.0  1.0   1.0   1.0   1.0 ⎥
        ⎢ 0.0  0.0  0.0  0.0  0.0  9.0   1.0   1.0   1.0 ⎥
        ⎢ 0.0  0.0  0.0  0.0  0.0  0.0  10.0  11.0  12.0 ⎥
        ⎣                                                    ⎦
```

Output

```
        ⎡                                                                    ⎤
        ⎢    0.333   -0.667   0.333  0.000  0.000  0.000   0.042 -0.042  0.000 ⎥
        ⎢   56.833  -52.167  -1.167 -0.500 -0.500 -0.357   6.836 -0.479 -0.500 ⎥
        ⎢  -55.167   51.833   0.833  0.500  0.500  0.214  -6.735  0.521  0.500 ⎥
        ⎢   -1.000    1.000   0.000  0.000  0.000  0.143  -0.143  0.000  0.000 ⎥
A    =  ⎢   -1.000    1.000   0.000  0.000  0.000  0.000   0.000  0.000  0.000 ⎥
        ⎢   -1.000    1.000   0.000  0.000  0.000  0.000  -0.125  0.125  0.000 ⎥
        ⎢ -226.000  206.000   5.000  3.000  2.000  1.429 -27.179  1.750  2.000 ⎥
        ⎢  560.000 -520.000 -10.000 -6.000 -4.000 -2.857  67.857 -5.000 -5.000 ⎥
        ⎢ -325.000  305.000   5.000  3.000  2.000  1.429 -39.554  3.125  3.000 ⎥
        ⎣                                                                    ⎦

RCOND    =  0.00005436
DET      =  (3.36, 2.00)
```

Example 2: This example computes the inverse of matrix ***A***, where *iopt* = 4 and matrix ***A*** is the transformed matrix factored in "Example 1" on page 469 by SGEF. The input contents of AUX, shown here, is the same as the output contents of IPVT in that example.

Call Statement and Input

```
             A  LDA  N  IOPT  RCOND   DET   AUX   NAUX
             |   |   |   |      |      |     |     |
CALL SGEICD( A , 9 , 9 , 4  , RCOND , DET , AUX , 300 )

A        =(same as output A in "Example 1" on page 469)
AUX      =  (3, 4, 5, 6, 7, 8, 9, 8, 9)
```

Output

```
        ⎡                                                                    ⎤
        |    0.333   -0.667   0.333  0.000  0.000  0.000   0.042 -0.042  0.000 |
        |   56.833  -52.167  -1.167 -0.500 -0.500 -0.357   6.836 -0.479 -0.500 |
        |  -55.167   51.833   0.833  0.500  0.500  0.214  -6.735  0.521  0.500 |
        |   -1.000    1.000   0.000  0.000  0.000  0.143  -0.143  0.000  0.000 |
A    =  |   -1.000    1.000   0.000  0.000  0.000  0.000   0.000  0.000  0.000 |
        |   -1.000    1.000   0.000  0.000  0.000  0.000  -0.125  0.125  0.000 |
        | -226.000  206.000   5.000  3.000  2.000  1.429 -27.179  1.750  2.000 |
        |  560.000 -520.000 -10.000 -6.000 -4.000 -2.857  67.857 -5.000 -5.000 |
        | -325.000  305.000   5.000  3.000  2.000  1.429 -39.554  3.125  3.000 |
        ⎣                                                                    ⎦
```

SPPICD, DPPICD, SPOICD, and DPOICD—Positive Definite Real Symmetric Matrix Inverse, Condition Number Reciprocal, and Determinant

These subroutines find the inverse, the reciprocal of the condition number, and the determinant of positive definite symmetric matrix ***A*** using Cholesky factorization, where:

- For SPPICD and DPPICD, ***A*** is stored in lower-packed storage mode.
- For SPOICD and DPOICD, ***A*** is stored in upper or lower storage mode.

Table 98. Data Types

A, *aux*, *rcond*, ***det***	**Subroutine**
Short-precision real	SPPICD and SPOICD
Long-precision real	DPPICD and DPOICD

Note: If you call these subroutines with *iopt* = 4, the input must be the output from the factorization subroutines SPPF, DPPF, SPOF/SPOFCD, or DPOF/DPOFCD, respectively, where Cholesky factorization was performed.

Syntax

Fortran	CALL SPPICD | DPPICD (*ap*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*) CALL SPOICD | DPOICD (*uplo*, *a*, *lda*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*)
C and C++	sppicd | dppicd (*ap*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*); spoicd | dpoicd (*uplo*, *a*, *lda*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*);
PL/I	CALL SPPICD | DPPICD (*ap*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*); CALL SPOICD | DPOICD (*uplo*, *a*, *lda*, *n*, *iopt*, *rcond*, *det*, *aux*, *naux*);

On Entry

uplo

indicates whether matrix ***A*** is stored in upper or lower storage mode, where:

If *uplo* = 'U', ***A*** is stored in upper storage mode.

If *uplo* = 'L', ***A*** is stored in lower storage mode.

Specified as: a single character. It must be 'U' or 'L'.

ap

is the array, referred to as AP, where:

If *iopt* = 0, 1, 2, or 3, then AP contains the positive definite real symmetric matrix ***A***, whose inverse, condition number reciprocal, and determinant are computed, where matrix ***A*** is stored in lower-packed storage mode.

If *iopt* = 4, then AP contains the transformed matrix ***A*** of order *n*, resulting from the Cholesky factorization performed in a previous call to SPPF or DPPF, respectively, whose inverse is computed.

Specified as: a one-dimensional array of (at least) length *n*(*n*+1)/2, containing numbers of the data type indicated in Table 98.

a
has the following meaning, where:

If *iopt* = 0, 1, 2, or 3, it is the positive definite real symmetric matrix ***A***, whose inverse, condition number reciprocal, and determinant are computed, where matrix ***A*** is stored in upper or lower storage mode.

If *iopt* = 4, it is the transformed matrix ***A*** of order *n*, containing results of the factorization from a previous call to SPOF/SPOFCD or DPOF/DPOFCD, respectively, whose inverse is computed.

Specified as: an *n* by (at least) *n* array, containing numbers of the data type indicated in Table 98 on page 521.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and $lda \geq n$.

n
is the order *n* of matrix ***A***. Specified as: a fullword integer; $n \geq 0$.

iopt
indicates the type of computation to be performed, where:

If *iopt* = 0, the inverse is computed for matrix ***A***.

If *iopt* = 1, the inverse and the reciprocal of the condition number are computed for matrix ***A***.

If *iopt* = 2, the inverse and the determinant are computed for matrix ***A***.

If *iopt* = 3, the inverse, the reciprocal of the condition number, and the determinant are computed for matrix ***A***.

If *iopt* = 4, the inverse is computed for the (Cholesky) factored matrix ***A***.

Specified as: a fullword integer; *iopt* = 0, 1, 2, 3, or 4.

rcond
See On Return.

det
See On Return.

aux
has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is the storage work area used by this subroutine. Its size is specified by *naux*. Specified as: an area of storage, containing numbers of the data type indicated in Table 98 on page 521.

naux
is the size of the work area specified by *aux*—that is, the number of elements in *aux*. Specified as: a fullword integer, where:

If *naux* = 0 and error 2015 is unrecoverable, SPPICD, DPPICD, SPOICD, AND DPOICD dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, $naux \geq n$.

On Return

ap
is the resulting array, referred to as AP, containing the inverse of the matrix in lower-packed storage mode. Returned as: a one-dimensional array of (at least) length $n(n+1)/2$, containing numbers of the data type indicated in Table 98 on page 521.

a
is the transformed matrix $\boldsymbol{A}$ of order n, containing the inverse of the matrix in upper or lower storage mode. Returned as: a two-dimensional array, containing numbers of the data type indicated in Table 98 on page 521.

rcond
is the reciprocal of the condition number, *rcond*, of matrix $\boldsymbol{A}$. Returned as: a real number of the data type indicated in Table 98 on page 521; $rcond \geq 0$.

det
is the vector ***det***, containing the two components det_1 and det_2 of the determinant of matrix $\boldsymbol{A}$. The determinant is:

$$det_1\left(10^{det_2}\right)$$

where $1 \leq det_1 < 10$. Returned as: an array of length 2, containing numbers of the data type indicated in Table 98 on page 521.

Notes

1. For these subroutines, when you specify *iopt* = 4, you must do the following:
   - For SPPICD and DPPICD, use Cholesky factorization in the previous call to SPPF and DPPF, respectively.
   - For SPOICD and DPOICD, specify the same storage mode for matrix $\boldsymbol{A}$ that was specified in the previous call to SPOF/SPOFCD and DPOF/DPOFCD, respectively.
   - The scalar data specified for input arguments *uplo*, *lda*, and *n* for these subroutines must be the same as the corresponding input arguments specified for SPOF/SPOFCD and DPOF/DPOFCD, respectively.
2. All subroutines accept lowercase letters for the *uplo* argument.
3. In your C program, argument *rcond* must be passed by reference.
4. For a description of how a positive definite symmetric matrix is stored in lower-packed storage mode in an array, see "Symmetric Matrix" on page 65. For a description of how a positive definite symmetric matrix is stored in upper or lower storage mode, see "Positive Definite or Negative Definite Symmetric Matrix" on page 69.
5. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: These subroutines find the inverse, the reciprocal of the condition number, and the determinant of positive definite symmetric matrix $\boldsymbol{A}$ using Cholesky factorization, where:

- $\boldsymbol{A}^{-1}$ is the inverse of matrix $\boldsymbol{A}$, where $\boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{A}^{-1}\boldsymbol{A} = \boldsymbol{I}$, and $\boldsymbol{I}$ is the identity matrix.

- $1/(\|\boldsymbol{A}\|_1)(\|\boldsymbol{A}^{-1}\|_1)$ is the reciprocal of the condition number, where $\|\boldsymbol{A}\|_1$ is the one-norm of matrix $\boldsymbol{A}$.
- $|\boldsymbol{A}|$ is the determinant of matrix $\boldsymbol{A}$, where $|\boldsymbol{A}|$ is expressed as:

$$det_1\left(10^{det_2}\right)$$

The *iopt* argument is used to determine the combination of output items produced by this subroutine: the inverse, the reciprocal of the condition number, and the determinant.

If *n* is 0, no computation is performed. See references [36], [38], and [44].

Error Conditions

Resource Errors

- Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.
- Unable to allocate internal work area.

Computational Errors: Matrix $\boldsymbol{A}$ is not positive definite.

- These subroutines do not perform the inverse, determinant, and reciprocal of the condition number computations.
- For *iopt* = 1, 2, or 3, the leading minor of order *i* has a nonpositive determinant. The order *i* is identified in the computational error message, issued by SPPF, DPPF, SPOF, or DPOF, respectively.

  For *iopt* = 4 for SPPICD and DPPICD, if the Cholesky factorization performed by SPPF or DPPF, respectively, failed due to a nonpositive definite matrix $\boldsymbol{A}$, the results from STPI or DTPI, respectively, are unpredictable, and a computational error message may be issued.

  For *iopt* = 4 for SPOICD and DPOICD, if the factorization performed by SPOF/SPOFCD or DPOF/DPOFCD, respectively, failed due to a nonpositive definite matrix $\boldsymbol{A}$, the results from STRI or DTRI, respectively, are unpredictable, and a computational error message may be issued.
- *i* can be determined at run time by using the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2115 in the ESSL error option table; otherwise, the default value causes your program to be terminated by SPPF, DPPF, SPOF, or DPOF, respectively, when this error occurs. If your program is not terminated by SPPF, DPPF, SPOF, or DPOF, respectively, the return code is set to 2. For details, see "What Can You Do about ESSL Computational Errors?" on page 48.

Input-Argument Errors

1. *uplo* ≠ 'U' or 'L'
2. $n < 0$
3. $lda \leq 0$
4. $lda < n$
5. *iopt* ≠ 0, 1, 2, 3, or 4
6. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example 1: This example uses SPPICD to compute the inverse, reciprocal of the condition number, and determinant of matrix $\boldsymbol{A}$. Where $\boldsymbol{A}$ is:

$$\begin{bmatrix} 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\ 1.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 \\ 1.0 & 2.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 \\ 1.0 & 2.0 & 3.0 & 4.0 & 4.0 & 4.0 & 4.0 & 4.0 & 4.0 \\ 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 5.0 & 5.0 & 5.0 & 5.0 \\ 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 6.0 & 6.0 & 6.0 \\ 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 7.0 & 7.0 \\ 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 8.0 & 8.0 \\ 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 8.0 & 9.0 \end{bmatrix}$$

The values used to compute the reciprocal of the condition number in this example are obtained with the following values:

$\|\boldsymbol{A}\|_1 = \max(9.0, 17.0, 24.0, 30.0, 35.0, 39.0, 42.0, 44.0, 45.0) = 45.0$
$\|\boldsymbol{A}^{-1}\|_1 = 4.0$

On output, the value in ***det***, $|\boldsymbol{A}|$, is equal to 1, and `RCOND` = 1/180.

Note: The `AP` arrays are formatted in a triangular arrangement for readability; however, they are stored in lower-packed storage mode.

Call Statement and Input

```
             AP   N  IOPT RCOND   DET   AUX  NAUX
             |    |   |     |      |     |    |
CALL SPPICD( AP , 9 , 3 , RCOND , DET , AUX , 9  )

AP   =   (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
          2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0,
          3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0,
          4.0, 4.0, 4.0, 4.0, 4.0, 4.0,
          5.0, 5.0, 5.0, 5.0, 5.0,
          6.0, 6.0, 6.0, 6.0,
          7.0, 7.0, 7.0,
          8.0, 8.0,
          9.0)
```

Output

```
AP   =   (2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
          2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
          2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
          2.0, -1.0, 0.0, 0.0, 0.0, 0.0,
          2.0, -1.0, 0.0, 0.0, 0.0,
          2.0, -1.0, 0.0, 0.0,
          2.0, -1.0, 0.0,
          2.0, -1.0,
          1.0)

RCOND    =  0.005556
DET      =  (1.0, 0.0)
```

Example 2: This example uses SPPICD to compute the inverse of matrix ***A***, where *iopt* = 4, and matrix ***A*** is the transformed matrix factored in "Example 1" on page 497 by SPPF.

Note: The AP arrays are formatted in a triangular arrangement for readability; however, they are stored in lower-packed storage mode.

Call Statement and Input

```
            AP   N  IOPT RCOND   DET   AUX NAUX
            |    |   |     |      |     |    |
CALL SPPICD(AP , 9 , 4 , RCOND , DET , AUX , 9)
```

AP =(same as output AP in "Example 2" on page 498 for SPPF)

Output

```
AP   =   (2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
          2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
          2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
          2.0, -1.0, 0.0, 0.0, 0.0, 0.0,
          2.0, -1.0, 0.0, 0.0, 0.0,
          2.0, -1.0, 0.0, 0.0,
          2.0, -1.0, 0.0,
          2.0, -1.0,
          1.0)
```

Example 3: This example uses SPOICD to compute the inverse, reciprocal of the condition number, and determinant of the same matrix ***A*** used in Example 1; however, matrix ***A*** is stored in upper storage mode in this example.

The values used to compute the reciprocal of the condition number in this example are obtained with the following values:

$\|\boldsymbol{A}\|_1$ = max(9.0, 17.0, 24.0, 30.0, 35.0, 39.0, 42.0, 44.0, 45.0) = 45.0
$\|\boldsymbol{A}^{-1}\|_1$ = 4.0

On output, the value in ***det***, $|\boldsymbol{A}|$, is equal to 1, and RCOND = 1/180.

Call Statement and Input

```
             UPLO  A  LDA   N  IOPT RCOND   DET   AUX  NAUX
              |    |   |    |   |     |      |     |    |
CALL SPOICD( 'U' , A , 9 ,  9 , 3 , RCOND , DET , AUX , 9  )
```

```
        ⎡                                                ⎤
        ⎢ 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 ⎥
        ⎢  .   2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0 ⎥
        ⎢  .    .   3.0  3.0  3.0  3.0  3.0  3.0  3.0 ⎥
        ⎢  .    .    .   4.0  4.0  4.0  4.0  4.0  4.0 ⎥
A    =  ⎢  .    .    .    .   5.0  5.0  5.0  5.0  5.0 ⎥
        ⎢  .    .    .    .    .   6.0  6.0  6.0  6.0 ⎥
        ⎢  .    .    .    .    .    .   7.0  7.0  7.0 ⎥
        ⎢  .    .    .    .    .    .    .   8.0  8.0 ⎥
        ⎢  .    .    .    .    .    .    .    .   9.0 ⎥
        ⎣                                                ⎦
```

Output

```
        ⎡                                                                ⎤
        ⎢ 2.0  -1.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0 ⎥
        ⎢  .    2.0  -1.0   0.0   0.0   0.0   0.0   0.0   0.0 ⎥
        ⎢  .     .    2.0  -1.0   0.0   0.0   0.0   0.0   0.0 ⎥
        ⎢  .     .     .    2.0  -1.0   0.0   0.0   0.0   0.0 ⎥
A    =  ⎢  .     .     .     .    2.0  -1.0   0.0   0.0   0.0 ⎥
        ⎢  .     .     .     .     .    2.0  -1.0   0.0   0.0 ⎥
        ⎢  .     .     .     .     .     .    2.0  -1.0   0.0 ⎥
        ⎢  .     .     .     .     .     .     .    2.0  -1.0 ⎥
        ⎢  .     .     .     .     .     .     .     .    1.0 ⎥
        ⎣                                                                ⎦

RCOND    =  0.005555556
DET      =  (1.0, 0.0)
```

Example 4: This example uses SPOICD to compute the inverse of matrix ***A***, where *iopt* = 4, and matrix ***A*** is the transformed matrix factored in "Example 1" on page 497 by SPOF.

Call Statement and Input

```
             UPLO  A  LDA   N  IOPT RCOND   DET   AUX  NAUX
              |    |   |    |   |     |      |     |    |
CALL SPOICD( 'U' , A , 9 ,  9 , 4 , RCOND , DET , AUX , 9  )
```

A =(same as output A in "Example 4" on page 500 for SPOF)

Output

```
        ⎡                                                                ⎤
        ⎢ 2.0  -1.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0 ⎥
        ⎢  .    2.0  -1.0   0.0   0.0   0.0   0.0   0.0   0.0 ⎥
        ⎢  .     .    2.0  -1.0   0.0   0.0   0.0   0.0   0.0 ⎥
        ⎢  .     .     .    2.0  -1.0   0.0   0.0   0.0   0.0 ⎥
A    =  ⎢  .     .     .     .    2.0  -1.0   0.0   0.0   0.0 ⎥
        ⎢  .     .     .     .     .    2.0  -1.0   0.0   0.0 ⎥
        ⎢  .     .     .     .     .     .    2.0  -1.0   0.0 ⎥
        ⎢  .     .     .     .     .     .     .    2.0  -1.0 ⎥
        ⎢  .     .     .     .     .     .     .     .    1.0 ⎥
        ⎣                                                                ⎦
```

STRSV, DTRSV, CTRSV, ZTRSV, STPSV, DTPSV, CTPSV, and ZTPSV—Solution of a Triangular System of Equations with a Single Right-Hand Side

STRSV, DTRSV, STPSV, and DTPSV perform one of the following solves for a triangular system of equations with a single right-hand side, using the vector $\boldsymbol{x}$ and triangular matrix $\boldsymbol{A}$ or its transpose:

Solution	Equation
1. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-1}\boldsymbol{x}$	$\boldsymbol{Ax} = \boldsymbol{b}$
2. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-T}\boldsymbol{x}$	$\boldsymbol{A}^{T}\boldsymbol{x} = \boldsymbol{b}$

CTRSV, ZTRSV, CTPSV, and ZTPSV perform one of the following solves for a triangular system of equations with a single right-hand side, using the vector $\boldsymbol{x}$ and and triangular matrix $\boldsymbol{A}$, its transpose, or its conjugate transpose:

Solution	Equation
1. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-1}\boldsymbol{x}$	$\boldsymbol{Ax} = \boldsymbol{b}$
2. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-T}\boldsymbol{x}$	$\boldsymbol{A}^{T}\boldsymbol{x} = \boldsymbol{b}$
3. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-T}\boldsymbol{x}$	$\boldsymbol{A}^{H}\boldsymbol{x} = \boldsymbol{b}$

Matrix $\boldsymbol{A}$ can be either upper or lower triangular, where:

- For the _TRSV subroutines, it is stored in upper- or lower-triangular storage mode, respectively.
- For the _TPSV subroutines, it is stored in upper- or lower-triangular-packed storage mode, respectively.

Note: The term $\boldsymbol{b}$ used in the systems of equations listed above represents the right-hand side of the system. It is important to note that in these subroutines the right-hand side of the equation is actually provided in the input-output argument *x*.

Table 99. Data Types

$\boldsymbol{A}$, $\boldsymbol{x}$	Subroutine
Short-precision real	STRSV and STPSV
Long-precision real	DTRSV and DTPSV
Short-precision complex	CTRSV and CTPSV
Long-precision complex	ZTRSV and ZTPSV

Syntax

Fortran	CALL STRSV | DTRSV | CTRSV | ZTRSV (*uplo*, *transa*, *diag*, *n*, *a*, *lda*, *x*, *incx*) CALL STPSV | DTPSV | CTPSV | ZTPSV (*uplo*, *transa*, *diag*, *n*, *ap*, *x*, *incx*)
C and C++	strsv | dtrsv | ctrsv | ztrsv (*uplo*, *transa*, *diag*, *n*, *a*, *lda*, *x*, *incx*); stpsv | dtpsv | ctpsv | ztpsv (*uplo*, *transa*, *diag*, *n*, *ap*, *x*, *incx*);

PL/I	CALL STRSV | DTRSV | CTRSV | ZTRSV (*uplo*, *transa*, *diag*, *n*, *a*, *lda*, *x*, *incx*); CALL STPSV | DTPSV | CTPSV | ZTPSV (*uplo*, *transa*, *diag*, *n*, *ap*, *x*, *incx*);

On Entry

uplo
indicates whether matrix ***A*** is an upper or lower triangular matrix, where:

If *uplo* = 'U', ***A*** is an upper triangular matrix.

If *uplo* = 'L', ***A*** is a lower triangular matrix.

Specified as: a single character. It must be 'U' or 'L'.

transa
indicates the form of matrix ***A*** used in the system of equations, where:

If *transa* = 'N', ***A*** is used, resulting in solution 1.

If *transa* = 'T', $\boldsymbol{A}^{T}$ is used, resulting in solution 2.

If *transa* = 'C', $\boldsymbol{A}^{H}$ is used, resulting in solution 3.

Specified as: a single character. It must be 'N', 'T', or 'C'.

diag
indicates the characteristics of the diagonal of matrix ***A***, where:

If *diag* = 'U', ***A*** is a unit triangular matrix.

If *diag* = 'N', ***A*** is not a unit triangular matrix.

Specified as: a single character. It must be 'U' or 'N'.

n
is the order of triangular matrix ***A***. Specified as: a fullword integer; $n \geq 0$ and $n \leq lda$.

a
is the upper or lower triangular matrix ***A*** of order *n*, stored in upper- or lower-triangular storage mode, respectively. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 99 on page 528.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and $lda \geq n$.

ap
is the upper or lower triangular matrix ***A*** of order *n*, stored in upper- or lower-triangular-packed storage mode, respectively. Specified as: a one-dimensional array of (at least) length $n(n+1)/2$, containing numbers of the data type indicated in Table 99 on page 528.

x
is the vector ***x*** of length *n*, containing the right-hand side of the triangular system to be solved. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 99 on page 528.

incx
is the stride for vector ***x***. Specified as: a fullword integer; $incx > 0$ or $incx < 0$.

On Return

x
is the solution vector ***x*** of length *n*, containing the results of the computation. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 99 on page 528.

Notes

1. These subroutines accept lowercase letters for the *uplo*, *transa*, and *diag* arguments.
2. For STRSV, DTRSV, STPSV, and DTPSV, if you specify `'C'` for the *transa* argument, it is interpreted as though you specified `'T'`.
3. Matrix ***A*** and vector ***x*** must have no common elements; otherwise, results are unpredictable.
4. ESSL assumes certain values in your array for parts of a triangular matrix. As a result, you do not have to set these values. For unit diagonal matrices, the elements of the diagonal are assumed to be 1.0 for real matrices and (1.0, 0.0) for complex matrices. When using upper- or lower-triangular storage, the unreferenced elements in the lower and upper triangular part, respectively, are assumed to be zero.
5. For a description of triangular matrices and how they are stored in upper- and lower-triangular storage mode and in upper- and lower-triangular-packed storage mode, see "Triangular Matrix" on page 74.

Function: These subroutines solve a triangular system of equations with a single right-hand side. The solution ***x*** may be any of the following, where triangular matrix ***A***, its transpose, or its conjugate transpose is used, and where ***A*** can be either upper- or lower-triangular:

1. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-1}\boldsymbol{x}$
2. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-T}\boldsymbol{x}$
3. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-T}\boldsymbol{x}$ (only for CTRSV, ZTRSV, CTPSV, and ZTPSV)

where:

x is a vector of length *n*.
A is an upper or lower triangular matrix of order *n*. For _TRSV, it is stored in upper- or lower-triangular storage mode, respectively. For _TPSV, it is stored in upper- or lower-triangular-packed storage mode, respectively.

If *n* is 0, no computation is performed. See references [32], [36], and [38].

Error Conditions

Computational Errors: None

Input-Argument Errors

1. *uplo* ≠ 'L' or 'U'
2. *transa* ≠ 'T', 'N', or 'C'
3. *diag* ≠ 'N' or 'U'
4. $n < 0$
5. $lda \leq 0$
6. $lda < n$
7. $incx = 0$

Example 1: This example shows the solution $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-1}\boldsymbol{x}$. Matrix $\boldsymbol{A}$ is a real 4 by 4 lower unit triangular matrix, stored in lower-triangular storage mode. Vector $\boldsymbol{x}$ is a vector of length 4.

Note: Because matrix $\boldsymbol{A}$ is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of 1.0 for the diagonal elements.

Call Statement and Input

```
           UPLO TRANSA DIAG  N   A  LDA  X  INCX
            |     |     |    |   |   |   |   |
CALL STRSV( 'L' , 'N' , 'U' , 4 , A , 4 , X , 1  )
```

```
        ⎡                ⎤
        |  .    .    .   . |
        | 1.0   .    .   . |
A   =   | 2.0  3.0   .   . |
        | 3.0  4.0  3.0  . |
        ⎣                ⎦
```

```
X       =  (1.0, 3.0, 11.0, 24.0)
```

Output

```
X       =  (1.0, 2.0, 3.0, 4.0)
```

Example 2: This example shows the solution $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-T}\boldsymbol{x}$. Matrix $\boldsymbol{A}$ is a real 4 by 4 upper nonunit triangular matrix, stored in upper-triangular storage mode. Vector $\boldsymbol{x}$ is a vector of length 4.

Call Statement and Input

```
           UPLO TRANSA DIAG  N   A  LDA  X  INCX
            |     |     |    |   |   |   |   |
CALL STRSV( 'U' , 'T' , 'N' , 4 , A , 4 , X , 1  )
```

```
        ⎡                ⎤
        | 1.0  2.0  3.0  2.0 |
A   =   |  .   2.0  2.0  5.0 |
        |  .    .   3.0  3.0 |
        |  .    .    .   1.0 |
        ⎣                ⎦
```

```
X       =  (5.0, 18.0, 32.0, 41.0)
```

Output

```
X       =  (5.0, 4.0, 3.0, 2.0)
```

Example 3: This example shows the solution $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-T}\boldsymbol{x}$. Matrix $\boldsymbol{A}$ is a complex 4 by 4 upper unit triangular matrix, stored in upper-triangular storage mode. Vector $\boldsymbol{x}$ is a vector of length 4.

Note: Because matrix $\boldsymbol{A}$ is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of (1.0, 0.0) for the diagonal elements.

Call Statement and Input

```
            UPLO TRANSA DIAG  N   A  LDA  X  INCX
             |     |     |    |   |   |   |   |
CALL CTRSV( 'U' , 'C' , 'U' , 4 , A , 4 , X , 1  )

        ┌                                        ┐
        | . (2.0, 2.0) (3.0,  3.0) (2.0, 2.0)    |
A    =  | .     .      (2.0,  2.0) (5.0, 5.0)    |
        | .     .         .        (3.0, 3.0)    |
        | .     .         .           .          |
        └                                        ┘

X       =  ((5.0, 5.0), (24.0, 4.0), (49.0, 3.0), (80.0, 2.0))
```

Output

```
X       =  ((5.0, 5.0), (4.0, 4.0), (3.0, 3.0), (2.0, 2.0))
```

Example 4: This example shows the solution $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-1}\boldsymbol{x}$. Matrix $\boldsymbol{A}$ is a real 4 by 4 lower unit triangular matrix, stored in lower-triangular-packed storage mode. Vector $\boldsymbol{x}$ is a vector of length 4. Matrix $\boldsymbol{A}$ is:

```
┌                    ┐
| 1.0   .    .    .  |
| 1.0  1.0   .    .  |
| 2.0  3.0  1.0   .  |
| 3.0  4.0  3.0  1.0 |
└                    ┘
```

Note: Because matrix $\boldsymbol{A}$ is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of 1.0 for the diagonal elements.

Call Statement and Input

```
            UPLO TRANSA DIAG  N   AP   X  INCX
             |     |     |    |   |    |   |
CALL STPSV( 'L' , 'N' , 'U' , 4 , AP , X , 1  )

AP      =  ( . , 1.0, 2.0, 3.0, . , 3.0, 4.0, . , 3.0, . )
X       =  (1.0, 3.0, 11.0, 24.0)
```

Output

```
X       =  (1.0, 2.0, 3.0, 4.0)
```

Example 5: This example shows the solution $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-T}\boldsymbol{x}$. Matrix $\boldsymbol{A}$ is a real 4 by 4 upper nonunit triangular matrix, stored in upper-triangular-packed storage mode. Vector $\boldsymbol{x}$ is a vector of length 4. Matrix $\boldsymbol{A}$ is:

$$\begin{bmatrix} 1.0 & 2.0 & 3.0 & 2.0 \\ . & 2.0 & 2.0 & 5.0 \\ . & . & 3.0 & 3.0 \\ . & . & . & 1.0 \end{bmatrix}$$

Call Statement and Input

```
            UPLO TRANSA DIAG  N   AP   X  INCX
             |     |     |    |   |    |   |
CALL STPSV( 'U' , 'T' , 'N' , 4 , AP , X , 1  )

AP       =  (1.0, 2.0, 2.0, 3.0, 2.0, 3.0, 2.0, 5.0, 3.0, 1.0)
X        =  (5.0, 18.0, 32.0, 41.0)
```

Output

```
X        =  (5.0, 4.0, 3.0, 2.0)
```

Example 6: This example shows the solution $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-T}\boldsymbol{x}$. Matrix $\boldsymbol{A}$ is a complex 4 by 4 upper unit triangular matrix, stored in upper-triangular-packed storage mode. Vector $\boldsymbol{x}$ is a vector of length 4. Matrix $\boldsymbol{A}$ is:

$$\begin{bmatrix} (1.0, 0.0) & (2.0, 2.0) & (3.0, 3.0) & (2.0, 2.0) \\ . & (1.0, 0.0) & (2.0, 2.0) & (5.0, 5.0) \\ . & . & (1.0, 0.0) & (3.0, 3.0) \\ . & . & . & (1.0, 0.0) \end{bmatrix}$$

Note: Because matrix $\boldsymbol{A}$ is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of (1.0, 0.0) for the diagonal elements.

Call Statement and Input

```
            UPLO TRANSA DIAG  N   AP   X  INCX
             |     |     |    |   |    |   |
CALL CTPSV( 'U' , 'C' , 'U' , 4 , AP , X , 1  )

AP       =  ( . , (2.0, 2.0), . , (3.0, 3.0), (2.0, 2.0), . ,
              (2.0, 2.0), (5.0, 5.0), (3.0, 3.0), . )
X        =  ((5.0, 5.0), (24.0, 4.0), (49.0, 3.0), (80.0, 2.0))
```

Output

```
X        =  ((5.0, 5.0), (4.0, 4.0), (3.0, 3.0), (2.0, 2.0))
```

STRSM, DTRSM, CTRSM, and ZTRSM—Solution of Triangular Systems of Equations with Multiple Right-Hand Sides

STRSM and DTRSM perform one of the following solves for a triangular system of equations with multiple right-hand sides, using scalar α, rectangular matrix $\boldsymbol{B}$, and triangular matrix $\boldsymbol{A}$ or its transpose:

Solution	Equation
1. $\boldsymbol{B} \leftarrow \alpha(\boldsymbol{A}^{-1})\boldsymbol{B}$	$\boldsymbol{AX} = \alpha\boldsymbol{B}$
2. $\boldsymbol{B} \leftarrow \alpha(\boldsymbol{A}^{-T})\boldsymbol{B}$	$\boldsymbol{A}^{T}\boldsymbol{X} = \alpha\boldsymbol{B}$
3. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{B}(\boldsymbol{A}^{-1})$	$\boldsymbol{XA} = \alpha\boldsymbol{B}$
4. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{B}(\boldsymbol{A}^{-T})$	$\boldsymbol{XA}^{T} = \alpha\boldsymbol{B}$

CTRSM and ZTRSM perform one of the following solves for a triangular system of equations with multiple right-hand sides, using scalar α, rectangular matrix $\boldsymbol{B}$, and triangular matrix $\boldsymbol{A}$, its transpose, or its conjugate transpose:

Solution	Equation
1. $\boldsymbol{B} \leftarrow \alpha(\boldsymbol{A}^{-1})\boldsymbol{B}$	$\boldsymbol{AX} = \alpha\boldsymbol{B}$
2. $\boldsymbol{B} \leftarrow \alpha(\boldsymbol{A}^{-T})\boldsymbol{B}$	$\boldsymbol{A}^{T}\boldsymbol{X} = \alpha\boldsymbol{B}$
3. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{B}(\boldsymbol{A}^{-1})$	$\boldsymbol{XA} = \alpha\boldsymbol{B}$
4. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{B}(\boldsymbol{A}^{-T})$	$\boldsymbol{XA}^{T} = \alpha\boldsymbol{B}$
5. $\boldsymbol{B} \leftarrow \alpha(\boldsymbol{A}^{-T})\boldsymbol{B}$	$\boldsymbol{A}^{H}\boldsymbol{X} = \alpha\boldsymbol{B}$
6. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{B}(\boldsymbol{A}^{-T})$	$\boldsymbol{XA}^{H} = \alpha\boldsymbol{B}$

Note: The term $\boldsymbol{X}$ used in the systems of equations listed above represents the output solution matrix. It is important to note that in these subroutines the solution matrix is actually returned in the input-output argument *b*.

Table 100. Data Types

$\boldsymbol{A}$, $\boldsymbol{B}$, α	Subroutine
Short-precision real	STRSM
Long-precision real	DTRSM
Short-precision complex	CTRSM
Long-precision complex	ZTRSM

Syntax

Fortran	CALL STRSM | DTRSM | CTRSM | ZTRSM (*side*, *uplo*, *transa*, *diag*, *m*, *n*, *alpha*, *a*, *lda*, *b*, *ldb*)
C and C++	strsm | dtrsm | ctrsm | ztrsm (*side*, *uplo*, *transa*, *diag*, *m*, *n*, *alpha*, *a*, *lda*, *b*, *ldb*);
PL/I	CALL STRSM | DTRSM | CTRSM | ZTRSM (*side*, *uplo*, *transa*, *diag*, *m*, *n*, *alpha*, *a*, *lda*, *b*, *ldb*);

On Entry

side
indicates whether the triangular matrix ***A*** is located to the left or right of rectangular matrix ***B*** in the system of equations, where:

If *side* = 'L', ***A*** is to the left of ***B***, resulting in solution 1, 2, or 5.

If *side* = 'R', ***A*** is to the right of ***B***, resulting in solution 3, 4, or 6.

Specified as: a single character. It must be 'L' or 'R'.

uplo
indicates whether matrix ***A*** is an upper or lower triangular matrix, where:

If *uplo* = 'U', ***A*** is an upper triangular matrix.

If *uplo* = 'L', ***A*** is a lower triangular matrix.

Specified as: a single character. It must be 'U' or 'L'.

transa
indicates the form of matrix ***A*** used in the system of equations, where:

If *transa* = 'N', ***A*** is used, resulting in solution 1 or 3.

If *transa* = 'T', $\boldsymbol{A}^{T}$ is used, resulting in solution 2 or 4.

If *transa* = 'C', $\boldsymbol{A}^{H}$ is used, resulting in solution 5 or 6.

Specified as: a single character. It must be 'N', 'T', or 'C'.

diag
indicates the characteristics of the diagonal of matrix ***A***, where:

If *diag* = 'U', ***A*** is a unit triangular matrix.

If *diag* = 'N', ***A*** is not a unit triangular matrix.

Specified as: a single character. It must be 'U' or 'N'.

m
is the number of rows in rectangular matrix ***B***, and:

If *side* = 'L', *m* is the order of triangular matrix ***A***.

Specified as: a fullword integer, where:

If *side* = 'L', $0 \le m \le lda$ and $m \le ldb$.

If *side* = 'R', $0 \le m \le ldb$.

n
is the number of columns in rectangular matrix ***B***, and:

If *side* = 'R', *n* is the order of triangular matrix ***A***.

Specified as: a fullword integer; $n \ge 0$, and:

If *side* = 'R', $n \le lda$.

alpha
is the scalar α. Specified as: a number of the data type indicated in Table 100 on page 534.

a
is the triangular matrix ***A***, of which only the upper or lower triangular portion is used, where:

If *side* = 'L', ***A*** is order *m*.

If *side* = 'R', ***A*** is order *n*.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 100 on page 534, where:

If *side* = 'L', its size must be *lda* by (at least) *m*.

If *side* = 'R', it size must be *lda* by (at least) *n*.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; *lda* > 0, and:

If *side* = 'L', *lda* ≥ *m*.

If *side* = 'R', *lda* ≥ *n*.

b
is the *m* by *n* rectangular matrix ***B***, which contains the right-hand sides of the triangular system to be solved. Specified as: an *ldb* by (at least) *n* array, containing numbers of the data type indicated in Table 100 on page 534.

ldb
is the leading dimension of the array specified for *b*. Specified as: a fullword integer; *ldb* > 0 and *ldb* ≥ *m*.

On Return

b
is the *m* by *n* matrix ***B***, containing the results of the computation.

Returned as: an *ldb* by (at least) *n* array, containing numbers of the data type indicated in Table 100 on page 534.

Notes

1. These subroutines accept lowercase letters for the *transa*, *side*, *diag*, and *uplo* arguments.
2. For STRSM and DTRSM, if you specify 'C' for the *transa* argument, it is interpreted as though you specified 'T'.
3. Matrices ***A*** and ***B*** must have no common elements or results are unpredictable.
4. If matrix ***A*** is upper triangular (*uplo* = 'U'), these subroutines refer to only the upper triangular portion of the matrix. If matrix ***A*** is lower triangular, (*uplo* = 'L'), these subroutines refer to only the lower triangular portion of the matrix. The unreferenced elements are assumed to be zero.
5. The elements of the diagonal of a unit triangular matrix are always one, so you do not need to set these values. The ESSL subroutines always assume that the values in these positions are 1.0 for STRSM and DTRSM and (1.0, 0.0) for CTRSM and ZTRSM.
6. For a description of triangular matrices and how they are stored, see "Triangular Matrix" on page 74.

Function: These subroutines solve a triangular system of equations with multiple right-hand sides. The solution ***B*** may be any of the following, where ***A*** is a triangular matrix and ***B*** is a rectangular matrix:

1. $\boldsymbol{B} \leftarrow \alpha(\boldsymbol{A}^{-1})\boldsymbol{B}$
2. $\boldsymbol{B} \leftarrow \alpha(\boldsymbol{A}^{-T})\boldsymbol{B}$
3. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{B}(\boldsymbol{A}^{-1})$
4. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{B}(\boldsymbol{A}^{-T})$

5. $\boldsymbol{B} \leftarrow \alpha(\boldsymbol{A}^{-H})\boldsymbol{B}$ (only for CTRSM and ZTRSM)
6. $\boldsymbol{B} \leftarrow \alpha\boldsymbol{B}(\boldsymbol{A}^{-H})$ (only for CTRSM and ZTRSM)

where:

α is a scalar.
B is an *m* by *n* rectangular matrix.
A is an upper or lower triangular matrix, where:
If *side* = 'L', it has order *m*, and equation 1, 2, or 5 is performed.
If *side* = 'R', it has order *n*, and equation 3, 4, or 6 is performed.

If *n* or *m* is 0, no computation is performed. See references [32] and [36].

Error Conditions

Resource Errors: Unable to allocate internal work area.

Computational Errors: None

Note: If the triangular matrix ***A*** is singular, the results returned by this subroutine are unpredictable, and there may be a divide-by-zero program exception message.

Input-Argument Errors

1. *m* < 0
2. *n* < 0
3. *lda*, *ldb* ≤ 0
4. *side* ≠ 'L' or 'R'
5. *uplo* ≠ 'L' or 'U'
6. *transa* ≠ 'T', 'N', or 'C'
7. *diag* ≠ 'N' or 'U'
8. *side* = 'L' and *m* > *lda*
9. *m* > *ldb*
10. *side* = 'R' and *n* > *lda*

Example 1: This example shows the solution $\boldsymbol{B} \leftarrow \alpha(\boldsymbol{A}^{-1})\boldsymbol{B}$, where ***A*** is a real 5 by 5 upper triangular matrix that is not unit triangular, and ***B*** is a real 5 by 3 rectangular matrix.

Call Statement and Input

```
           SIDE  UPLO TRANSA  DIAG  M   N   ALPHA   A  LDA  B  LDB
            |     |     |      |    |   |     |     |   |   |   |
CALL STRSM( 'L' , 'U' , 'N'  , 'N' , 5 , 3 ,  1.0  , A , 7 , B , 6 )
```

$$A = \begin{bmatrix} 3.0 & -1.0 & 2.0 & 2.0 & 1.0 \\ . & -2.0 & 4.0 & -1.0 & 3.0 \\ . & . & -3.0 & 0.0 & 2.0 \\ . & . & . & 4.0 & -2.0 \\ . & . & . & . & 1.0 \\ . & . & . & . & . \\ . & . & . & . & . \end{bmatrix}$$

```
          ┌                  ┐
          │   6.0  10.0   -2.0 │
          │ -16.0  -1.0    6.0 │
B    =    │  -2.0   1.0   -4.0 │
          │  14.0   0.0  -14.0 │
          │  -1.0   2.0    1.0 │
          │    .     .      .  │
          └                  ┘
```

Output

```
          ┌                  ┐
          │  2.0  3.0   1.0 │
          │  5.0  5.0   4.0 │
B    =    │  0.0  1.0   2.0 │
          │  3.0  1.0  -3.0 │
          │ -1.0  2.0   1.0 │
          │   .    .     .  │
          └                  ┘
```

Example 2: This example shows the solution $\boldsymbol{B} \leftarrow \alpha(\boldsymbol{A}^{-T})\boldsymbol{B}$, where $\boldsymbol{A}$ is a real 5 by 5 upper triangular matrix that is not unit triangular, and $\boldsymbol{B}$ is a real 5 by 4 rectangular matrix.

Call Statement and Input

```
            SIDE  UPLO TRANSA  DIAG  M   N    ALPHA  A  LDA  B  LDB
             |     |     |      |    |   |     |     |   |   |   |
CALL STRSM( 'L' , 'U' , 'T'  , 'N' , 5 , 4 ,  1.0  , A , 7 , B , 6 )
```

```
          ┌                        ┐
          │ -1.0  -4.0  -2.0   2.0   3.0 │
          │   .   -2.0   2.0   2.0   2.0 │
          │   .     .   -3.0  -1.0   4.0 │
A    =    │   .     .     .    1.0   0.0 │
          │   .     .     .     .   -2.0 │
          │   .     .     .     .     .  │
          │   .     .     .     .     .  │
          └                        ┘
```

```
          ┌                    ┐
          │ -1.0  -2.0   -3.0   -4.0 │
          │  2.0  -2.0  -14.0  -12.0 │
B    =    │ 10.0   5.0   -8.0   -7.0 │
          │ 14.0  15.0    1.0    8.0 │
          │ -3.0   4.0    3.0   16.0 │
          │   .     .      .      .  │
          └                    ┘
```

Output

```
          ┌                  ┐
          │  1.0   2.0   3.0   4.0 │
          │  3.0   3.0  -1.0   2.0 │
B    =    │ -2.0  -1.0   0.0   1.0 │
          │  4.0   4.0  -3.0  -3.0 │
          │  2.0   2.0   2.0   2.0 │
          │   .     .     .     .  │
          └                  ┘
```

Example 3: This example shows the solution $B \leftarrow \alpha B(A^{-1})$, where A is a real 5 by 5 lower triangular matrix that is not unit triangular, and B is a real 3 by 5 rectangular matrix.

Call Statement and Input

```
           SIDE  UPLO TRANSA  DIAG  M   N    ALPHA  A  LDA  B  LDB
            |     |     |      |    |   |     |     |   |   |   |
CALL STRSM( 'R' , 'L' , 'N'  , 'N' , 3 , 5 ,  1.0  , A , 7 , B , 4 )

        ⎡                          ⎤
        | 2.0   .     .     .     . |
        | 2.0  3.0    .     .     . |
        | 2.0  1.0   1.0    .     . |
A   =   | 0.0  3.0   0.0  -2.0    . |
        | 2.0  4.0  -1.0   2.0  -1.0 |
        |  .    .     .     .     . |
        |  .    .     .     .     . |
        ⎣                          ⎦

        ⎡                          ⎤
        | 10.0   4.0   0.0  0.0   1.0 |
B   =   | 10.0  14.0  -4.0  6.0  -3.0 |
        | -8.0   2.0  -5.0  4.0  -2.0 |
        |   .     .     .    .     .  |
        ⎣                          ⎦
```

Output

```
        ⎡                          ⎤
        |  3.0   4.0  -1.0  -1.0  -1.0 |
B   =   |  2.0   1.0  -1.0   0.0   3.0 |
        | -2.0  -1.0  -3.0   0.0   2.0 |
        |   .     .     .     .     .  |
        ⎣                          ⎦
```

Example 4: This example shows the solution $B \leftarrow \alpha B(A^{-1})$, where A is a real 6 by 6 upper triangular matrix that is unit triangular, and B is a real 1 by 6 rectangular matrix.

Note: Because matrix A is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of 1.0 for the diagonal element.

Call Statement and Input

```
           SIDE  UPLO TRANSA  DIAG  M   N    ALPHA  A  LDA  B  LDB
            |     |     |      |    |   |     |     |   |   |   |
CALL STRSM( 'R' , 'U' , 'N'  , 'U' , 1 , 6 ,  1.0  , A , 7 , B , 2 )

        ⎡                          ⎤
        | .  2.0  -3.0  1.0   2.0   4.0 |
        | .   .    0.0  1.0   1.0  -2.0 |
        | .   .     .   4.0  -1.0   1.0 |
A   =   | .   .     .    .    0.0  -1.0 |
        | .   .     .    .     .    2.0 |
        | .   .     .    .     .     .  |
        | .   .     .    .     .     .  |
        ⎣                          ⎦
```

```
B    =  | 1.0  4.0  -2.0  10.0  2.0  -6.0 |
        |  .    .     .     .    .     .  |
```

Output

```
B    =  | 1.0  2.0  1.0  3.0  -1.0  -2.0 |
        |  .    .    .    .     .     .  |
```

Example 5: This example shows the solution $B \leftarrow \alpha B(A^{-1})$, where A is a complex 5 by 5 lower triangular matrix that is not unit triangular, and B is a complex 3 by 5 rectangular matrix.

Call Statement and Input

```
           SIDE  UPLO TRANSA  DIAG  M   N    ALPHA   A  LDA  B  LDB
            |     |     |      |    |   |      |     |   |   |   |
CALL CTRSM( 'R' , 'L' , 'N'  , 'N' , 3 , 5 ,  ALPHA , A , 7 , B , 4 )

ALPHA    =  (1.0, 0.0)

        | (2.0, -3.0)     .           .           .           .       |
        | (2.0, -4.0) (3.0, -1.0)     .           .           .       |
        | (2.0,  2.0) (1.0,  2.0)  (1.0,  1.0)     .           .       |
A    =  | (0.0,  0.0) (3.0, -1.0)  (0.0, -1.0) (-2.0,  1.0)     .       |
        | (2.0,  2.0) (4.0,  0.0) (-1.0,  2.0)  (2.0, -4.0) (-1.0, -4.0) |
        |     .           .           .           .           .       |
        |     .           .           .           .           .       |

        | (22.0, -41.0)  (7.0, -26.0)  (9.0, 0.0)  (-15.0, -3.0)  (-15.0, 8.0) |
B    =  | (29.0, -18.0) (24.0, -10.0)  (9.0, 6.0) (-12.0, -24.0) (-19.0, -8.0) |
        |  (-15.0, 2.0) (-3.0, -21.0) (-2.0, 4.0)  (-4.0, -12.0) (-10.0, -6.0) |
        |        .           .           .           .           .        |
```

Output

```
        |  (3.0, 0.0)   (4.0, 0.0) (-1.0, -2.0) (-1.0, -1.0) (-1.0, -4.0) |
B    =  | (2.0, -1.0)   (1.0, 2.0) (-1.0, -3.0)   (0.0, 2.0)  (3.0, -4.0) |
        | (-2.0, 1.0) (-1.0, -3.0)  (-3.0, 1.0)   (0.0, 0.0)  (2.0, -2.0) |
        |     .           .           .           .           .     |
```

Example 6: This example shows the solution $B \leftarrow \alpha(A^{-T})B$, where A is a complex 5 by 5 upper triangular matrix that is not unit triangular, and B is a complex 5 by 1 rectangular matrix.

Call Statement and Input

```
           SIDE  UPLO TRANSA  DIAG  M   N    ALPHA   A  LDA  B  LDB
            |     |     |      |    |   |      |     |   |   |   |
CALL CTRSM( 'L' , 'U' , 'C'  , 'N' , 5 , 1 ,  ALPHA , A , 6 , B , 6 )

ALPHA    =  (1.0, 0.0)

        ┌                                                              ┐
        │ (-4.0, 1.0) (4.0, -3.0)  (-1.0, 3.0)   (0.0, 0.0)  (-1.0, 0.0) │
        │      .      (-2.0, 0.0) (-3.0, -1.0) (-2.0, -1.0)   (4.0, 3.0) │
A    =  │      .           .       (-5.0, 3.0) (-3.0, -3.0) (-5.0, -5.0) │
        │      .           .            .       (4.0, -4.0)   (2.0, 0.0) │
        │      .           .            .           .        (2.0, -1.0) │
        │      .           .            .           .             .      │
        └                                                              ┘

        ┌             ┐
        │ (-8.0, -19.0) │
        │  (8.0,  21.0) │
B    =  │ (44.0,  -8.0) │
        │ (13.0,  -7.0) │
        │ (19.0,   2.0) │
        │      .        │
        └             ┘
```

Output

```
        ┌             ┐
        │  (3.0, 4.0) │
        │ (-4.0, 2.0) │
B    =  │ (-5.0, 0.0) │
        │  (1.0, 3.0) │
        │  (3.0, 1.0) │
        │      .      │
        └             ┘
```

STRI, DTRI, STPI, and DTPI—Triangular Matrix Inverse

These subroutines find the inverse of triangular matrix ***A***:

$$A \leftarrow A^{-1}$$

Matrix ***A*** can be either upper or lower triangular, where:

- For the _TRI subroutines, it is stored in upper- or lower-triangular storage mode, respectively.
- For the _TPI subroutines, it is stored in upper- or lower-triangular-packed storage mode, respectively.

Table 101. Data Types

A	**Subroutine**
Short-precision real	STRI and STPI
Long-precision real	DTRI and DTPI

Syntax

Fortran	CALL STRI | DTRI (*uplo*, *diag*, *a*, *lda*, *n*) CALL STPI | DTPI (*uplo*, *diag*, *ap*, *n*)
C and C++	stri | dtri (*uplo*, *diag*, *a*, *lda*, *n*); stpi | dtpi (*uplo*, *diag*, *ap*, *n*);
PL/I	CALL STRI | DTRI (*uplo*, *diag*, *a*, *lda*, *n*); CALL STPI | DTPI (*uplo*, *diag*, *ap*, *n*);

On Entry

uplo
indicates whether matrix ***A*** is an upper or lower triangular matrix, where:

If *uplo* = 'U', ***A*** is an upper triangular matrix.

If *uplo* = 'L', ***A*** is a lower triangular matrix.

Specified as: a single character. It must be 'U' or 'L'.

diag
indicates the characteristics of the diagonal of matrix ***A***, where:

If *diag* = 'U', ***A*** is a unit triangular matrix.

If *diag* = 'N', ***A*** is not a unit triangular matrix.

Specified as: a single character. It must be 'U' or 'N'.

a
is the upper or lower triangular matrix ***A*** of order *n*, stored in upper- or lower-triangular storage mode, respectively. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 101.

lda
is the leading dimension of the arrays specified for *a*. Specified as: a fullword integer; $lda > 0$ and $lda \geq n$.

ap
is the upper or lower triangular matrix ***A*** of order *n*, stored in upper- or lower-triangular-packed storage mode, respectively. Specified as: a one-dimensional array of (at least) length $n(n+1)/2$, containing numbers of the data type indicated in Table 101 on page 542.

n
is the order of matrix ***A***. Specified as: a fullword integer; $n \geq 0$, where:

On Return

a
is the inverse of the upper or lower triangular matrix ***A*** of order *n*, stored in upper- or lower-triangular storage mode, respectively. Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 101 on page 542.

ap
is the inverse of the upper or lower triangular matrix ***A*** of order *n*, stored in upper- or lower-triangular-packed storage mode, respectively. Returned as: a one-dimensional array of (at least) length $n(n+1)/2$, containing numbers of the data type indicated in Table 101 on page 542.

Notes

1. These subroutines accept lowercase letters for the *uplo* and *diag* arguments.
2. If matrix ***A*** is upper triangular (*uplo* = 'U'), these subroutines refer to only the upper triangular portion of the matrix. If matrix ***A*** is lower triangular, (*uplo* = 'L'), these subroutines refer to only the lower triangular portion of the matrix. The unreferenced elements are assumed to be zero.
3. The elements of the diagonal of a unit triangular matrix are always one, so you do not need to set these values.
4. For a description of triangular matrices and how they are stored in upper- and lower-triangular storage mode and in upper- and lower-triangular-packed storage mode, see "Triangular Matrix" on page 74.

Function: These subroutines find the inverse of triangular matrix ***A***, where ***A*** is either upper or lower triangular:

$$\boldsymbol{A} \leftarrow \boldsymbol{A}^{-1}$$

where:

A is the triangular matrix of order *n*.
$\boldsymbol{A}^{-1}$ the inverse of the triangular matrix of order *n*.

If *n* is 0, no computation is performed. See references [8] and [36].

Error Conditions

Resource Errors: Unable to allocate internal work area.

Computational Errors: Matrix ***A*** is singular.

- One or more of the diagonal elements of matrix ***A*** are zero. The first column, *i*, of matrix ***A***, in which a zero diagonal element is found, is identified in the computational error message.
- The return code is set to 1.
- *i* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2145 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see "What Can You Do about ESSL Computational Errors?" on page 48.

Input-Argument Errors

1. *uplo* ≠ 'U' or 'L'
2. *diag* ≠ 'U' or 'N'
3. $n < 0$
4. $lda \leq 0$
5. $lda < n$

Example 1: This example shows how the inverse of matrix ***A*** is computed, where ***A*** is a 5 by 5 upper triangular matrix that is not unit triangular and is stored in upper-triangular storage mode. Matrix ***A*** is:

```
┌                                    ┐
│ 1.00  3.00  4.00   5.00   6.00 │
│ 0.00  2.00  8.00   9.00   1.00 │
│ 0.00  0.00  4.00   8.00   4.00 │
│ 0.00  0.00  0.00  -2.00   6.00 │
│ 0.00  0.00  0.00   0.00  -1.00 │
└                                    ┘
```

and where the following inverse matrix is computed. Matrix $\boldsymbol{A}^{-1}$ is:

```
┌                                      ┐
│ 1.00  -1.50   2.00   3.75   35.00 │
│ 0.00   0.50  -1.00  -1.75  -14.00 │
│ 0.00   0.00   0.25   1.00    7.00 │
│ 0.00   0.00   0.00  -0.50   -3.00 │
│ 0.00   0.00   0.00   0.00   -1.00 │
└                                      ┘
```

Call Statement and Input

```
           UPLO  DIAG  A  LDA  N
            |     |    |   |   |
CALL STRI( 'U' , 'N' , A , 5 , 5)
```

```
        ⎡                                    ⎤
        ⎢ 1.00  3.00  4.00   5.00   6.00 ⎥
        ⎢  .    2.00  8.00   9.00   1.00 ⎥
A    =  ⎢  .     .    4.00   8.00   4.00 ⎥
        ⎢  .     .     .    -2.00   6.00 ⎥
        ⎢  .     .     .      .    -1.00 ⎥
        ⎣                                    ⎦
```

Output

```
        ⎡                                    ⎤
        ⎢ 1.00  -1.50   2.00   3.75   35.00 ⎥
        ⎢  .     0.50  -1.00  -1.75  -14.00 ⎥
A    =  ⎢  .      .     0.25   1.00    7.00 ⎥
        ⎢  .      .      .    -0.50   -3.00 ⎥
        ⎢  .      .      .      .     -1.00 ⎥
        ⎣                                    ⎦
```

Example 2: This example shows how the inverse of matrix ***A*** is computed, where ***A*** is a 5 by 5 lower triangular matrix that is unit triangular and is stored in lower-triangular storage mode. Matrix ***A*** is:

```
⎡                                    ⎤
⎢ 1.0  0.0  0.0  0.0  0.0 ⎥
⎢ 3.0  1.0  0.0  0.0  0.0 ⎥
⎢ 4.0  8.0  1.0  0.0  0.0 ⎥
⎢ 5.0  9.0  8.0  1.0  0.0 ⎥
⎢ 6.0  1.0  4.0  6.0  1.0 ⎥
⎣                                    ⎦
```

and where the following inverse matrix is computed. Matrix $\boldsymbol{A}^{-1}$ is:

```
⎡                                    ⎤
⎢    1.0     0.0   0.0   0.0  0.0 ⎥
⎢   -3.0     1.0   0.0   0.0  0.0 ⎥
⎢   20.0    -8.0   1.0   0.0  0.0 ⎥
⎢ -138.0    55.0  -8.0   1.0  0.0 ⎥
⎢  745.0  -299.0  44.0  -6.0  1.0 ⎥
⎣                                    ⎦
```

Note: Because matrix ***A*** is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of 1.0 for the diagonal elements.

Call Statement and Input

```
           UPLO  DIAG  A  LDA  N
            |     |    |   |   |
CALL STRI( 'L' , 'U' , A , 5 , 5)
```

```
        ⎡                          ⎤
        ⎢  .    .    .    .   . ⎥
        ⎢ 3.0   .    .    .   . ⎥
A    =  ⎢ 4.0  8.0   .    .   . ⎥
        ⎢ 5.0  9.0  8.0   .   . ⎥
        ⎢ 6.0  1.0  4.0  6.0  . ⎥
        ⎣                          ⎦
```

Output

```
        ⎡                                   ⎤
        ⎢     .       .     .     .   .  ⎥
        ⎢   -3.0      .     .     .   .  ⎥
A    =  ⎢   20.0    -8.0    .     .   .  ⎥
        ⎢ -138.0    55.0  -8.0    .   .  ⎥
        ⎢  745.0  -299.0  44.0  -6.0  .  ⎥
        ⎣                                   ⎦
```

Example 3: This example shows how the inverse of matrix ***A*** is computed, where ***A*** is the same matrix shown in Example 1 and is stored in upper-triangular-packed storage mode. The inverse matrix computed here is the same as the inverse matrix shown in Example 1 and is stored in upper-triangular-packed storage mode.

Call Statement and Input

```
           UPLO  DIAG  AP   N
            |     |    |    |
CALL STPI( 'U' , 'N' , AP , 5)

AP       =  (1.00, 3.00, 2.00, 4.00, 8.00, 4.00, 5.00, 9.00, 8.00,
             -2.00, 6.00, 1.00, 4.00, 6.00, -1.00)
```

Output

```
AP       =  (1.00, -1.50, 0.50, 2.00, -1.00, 0.25, 3.75, -1.75, 1.00,
             -0.50, 35.00, -14.00, 7.00, -3.00, -1.00)
```

Example 4: This example shows how the inverse of matrix ***A*** is computed, where ***A*** is the same matrix shown in Example 2 and is stored in lower-triangular-packed storage mode. The inverse matrix computed here is the same as the inverse matrix shown in Example 2 and is stored in lower-triangular-packed storage mode.

Note: Because matrix ***A*** is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of 1.0 for the diagonal elements.

Call Statement and Input

```
           UPLO  DIAG  AP   N
            |     |    |    |
CALL STPI( 'L' , 'U' , AP , 5)

AP       =  ( . , 3.0, 4.0, 5.0, 6.0, . , 8.0, 9.0, 1.0, . , 8.0, 4.0,
             . , 6.0, . )
```

Output

```
AP       =  ( . , -3.0, 20.0, -138.0, 745.0, . , -8.0, 55.0, -299.0,
             . , -8.0, 44.0, . , -6.0, . )
```

Banded Linear Algebraic Equation Subroutines

This section contains the banded linear algebraic equation subroutine descriptions.

SGBF and DGBF—General Band Matrix Factorization

These subroutines factor general band matrix ***A***, stored in general-band storage mode, using Gaussian elimination. To solve the system of equations with one or more right-hand sides, follow the call to these subroutines with one or more calls to SGBS or DGBS, respectively.

Table 102. Data Types

A	**Subroutine**
Short-precision real	SGBF
Long-precision real	DGBF

Note: The output from these factorization subroutines should be used only as input to the solve subroutines SGBS and DGBS, respectively.

Syntax

Fortran	CALL SGBF | DGBF (*agb*, *lda*, *n*, *ml*, *mu*, *ipvt*)
C and C++	sgbf | dgbf (*agb*, *lda*, *n*, *ml*, *mu*, *ipvt*);
PL/I	CALL SGBF | DGBF (*agb*, *lda*, *n*, *ml*, *mu*, *ipvt*);

On Entry

agb
is the general band matrix ***A*** of order *n*, stored in general-band storage mode, to be factored. It has an upper band width *mu* and a lower band width *ml*. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 102, where $lda \geq 2ml+mu+16$.

lda
is the leading dimension of the array specified for *agb*. Specified as: a fullword integer; $lda > 0$ and $lda \geq 2ml+mu+16$.

n
is the order of the matrix ***A***. Specified as: a fullword integer; $n > ml$ and $n > mu$.

ml
is the lower band width *ml* of the matrix ***A***. Specified as: a fullword integer; $0 \leq ml < n$.

mu
is the upper band width *mu* of the matrix ***A***. Specified as: a fullword integer; $0 \leq mu < n$.

ipvt
See On Return.

On Return

agb
is the transformed matrix ***A*** of order *n*, containing the results of the factorization. See "Function" on page 549. Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 102.

ipvt
is the integer vector ***ipvt*** of length *n*, containing the pivot information necessary to construct matrix ***L*** from the information contained in the output array *agb*. Returned as: a one-dimensional array of (at least) length *n*, containing fullword integers.

Notes

1. ***ipvt*** is not a permutation vector in the strict sense. It is used to record column interchanges in ***L*** due to partial pivoting and to improve performance.
2. The entire *lda* by *n* array specified for *agb* must remain unchanged between calls to the factorization and solve subroutines.
3. This subroutine can be used for tridiagonal matrices (*ml* = *mu* = 1); however, the tridiagonal subroutines SGTF/DGTF and SGTS/DGTS are faster.
4. For a description of how a general band matrix is stored in general-band storage mode in an array, see "General Band Matrix" on page 77.

Function: The general band matrix ***A***, stored in general-band storage mode, is factored using Gaussian elimination with partial pivoting to compute the ***LU*** factorization of ***A***, where:

ipvt is a vector containing the pivoting information.
L is a unit lower triangular band matrix.
U is an upper triangular band matrix.

The transformed matrix ***A*** contains ***U*** in packed format, along with the multipliers necessary to construct, with the help of ***ipvt***, a matrix ***L***, such that ***A*** = ***LU***. This factorization can then be used by SGBS or DGBS, respectively, to solve the system of equations. See reference [38].

Error Conditions

Resource Errors: Unable to allocate internal work area.

Computational Errors: Matrix ***A*** is singular.

- One or more columns of ***L*** and the corresponding diagonal of ***U*** contain all zeros (all columns of ***L*** are checked). The last column, *i*, of ***L*** with a corresponding ***U*** = 0 diagonal element is identified in the computational error message.
- The return code is set to 1.
- *i* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2103 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see "What Can You Do about ESSL Computational Errors?" on page 48.

Input-Argument Errors

1. $lda \leq 0$
2. $ml < 0$
3. $ml \geq n$
4. $mu < 0$
5. $mu \geq n$
6. $lda < 2ml+mu+16$

Example: This example shows a factorization of a general band matrix ***A*** of order 9, with a lower band width of 2 and an upper band width of 3. On input matrix ***A*** is:

```
┌                                                   ┐
│ 1.0  1.0  1.0  1.0  0.0  0.0   0.0   0.0   0.0 │
│ 1.0  1.0  1.0  1.0  1.0  0.0   0.0   0.0   0.0 │
│ 4.0  1.0  1.0  1.0  1.0  1.0   0.0   0.0   0.0 │
│ 0.0  5.0  1.0  1.0  1.0  1.0   1.0   0.0   0.0 │
│ 0.0  0.0  6.0  1.0  1.0  1.0   1.0   1.0   0.0 │
│ 0.0  0.0  0.0  7.0  1.0  1.0   1.0   1.0   1.0 │
│ 0.0  0.0  0.0  0.0  8.0  1.0   1.0   1.0   1.0 │
│ 0.0  0.0  0.0  0.0  0.0  9.0   1.0   1.0   1.0 │
│ 0.0  0.0  0.0  0.0  0.0  0.0  10.0  11.0  12.0 │
└                                                   ┘
```

Matrix ***A*** is stored in general-band storage mode in the two-dimensional array AGB of size LDA by N, where LDA = 2*ml*+*mu*+16 = 23. The array AGB is declared as AGB(1:23,1:9).

Note: Matrix ***A*** is the same matrix used in the examples in subroutines SGEF and DGEF (see "Example 1" on page 469) and SGEFCD and DGEFCD (see "Example" on page 493).

Call Statement and Input

```
                AGB  LDA   N   ML  MU  IPVT )
                 |    |    |   |   |    |
CALL SGBF( AGB , 23 , 9 , 2 , 3 , IPVT )
```

```
      ┌                                                                                   ┐
      │ 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 │
      │ 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 │
      │ 0.0000  0.0000  0.0000  1.0000  1.0000  1.0000   1.0000   1.0000   1.0000 │
      │ 0.0000  0.0000  1.0000  1.0000  1.0000  1.0000   1.0000   1.0000   1.0000 │
      │ 0.0000  1.0000  1.0000  1.0000  1.0000  1.0000   1.0000   1.0000   1.0000 │
      │ 1.0000  1.0000  1.0000  1.0000  1.0000  1.0000   1.0000   1.0000  12.0000 │
      │ 1.0000  1.0000  1.0000  1.0000  1.0000  1.0000   1.0000  11.0000   0.0000 │
      │ 4.0000  5.0000  6.0000  7.0000  8.0000  9.0000  10.0000   0.0000   0.0000 │
      │ 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 │
      │ 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 │
      │ 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 │
AGB = │ 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 │
      │ 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 │
      │ 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 │
      │ 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 │
      │ 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 │
      │ 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 │
      │ 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 │
      │ 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 │
      │ 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 │
      │ 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 │
      │ 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 │
      │ 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000 │
      └                                                                                   ┘
```

Output

```
                 ⎡                                                                                   ⎤
                 │ 0.0000  0.0000  0.0000  0.0000   0.0000   1.0000   1.0000   1.0000   1.0000 │
                 │ 0.0000  0.0000  0.0000  0.0000   1.0000   1.0000   1.0000   1.0000   1.0000 │
                 │ 0.0000  0.0000  0.0000  1.0000   1.0000   1.0000   1.0000   1.0000   1.0000 │
                 │ 0.0000  0.0000  1.0000  1.0000   1.0000   1.0000   1.0000   1.0000  12.0000 │
                 │ 0.0000  1.0000  1.0000  1.0000   1.0000   1.0000   1.0000  11.0000   0.3111 │
                 │ 0.2500  0.2000  0.1600  0.1400   0.1250   0.1100   0.1000   5.5380  -325.00 │
                 │ 0.0000  0.1500  0.0000  0.0714   0.0000  -0.0556  -0.0306   0.9385   0.0000 │
                 │ 0.2500  0.1500  0.1000  0.0714  -0.0714  -0.0694  -0.0194   0.0000   0.0000 │
                 │ 0.2500  0.0000  0.1000  0.0000   0.0536   0.0000   0.0000   0.0000   0.0000 │
                 │ 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 │
                 │ 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 │
AGB  =           │ 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 │
                 │ 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 │
                 │ 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 │
                 │ 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 │
                 │ 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 │
                 │ 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 │
                 │ 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 │
                 │ 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 │
                 │ 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 │
                 │ 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 │
                 │ 0.0000  0.0000  0.0000  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000 │
                 ⎣                                                                                   ⎦

                      IPVT     =  (2, -65534, -131070, -196606, -262142, -327678, -327678,
                                   -327680, -327680)
```

SGBS and DGBS—General Band Matrix Solve

These subroutines solve the system $\boldsymbol{Ax} = \boldsymbol{b}$ for $\boldsymbol{x}$, where $\boldsymbol{A}$ is a general band matrix, and $\boldsymbol{x}$ and $\boldsymbol{b}$ are vectors. They use the results of the factorization of matrix $\boldsymbol{A}$, produced by a preceding call to SGBF or DGBF, respectively.

Table 103. Data Types

A, ***b***, ***x***	**Subroutine**
Short-precision real	SGBS
Long-precision real	DGBS

Note: The input to these solve subroutines must be the output from the factorization subroutines SGBF and DGBF, respectively.

Syntax

Fortran	CALL SGBS | DGBS (*agb*, *lda*, *n*, *ml*, *mu*, *ipvt*, *bx*)
C and C++	sgbs | dgbs (*agb*, *lda*, *n*, *ml*, *mu*, *ipvt*, *bx*);
PL/I	CALL SGBS | DGBS (*agb*, *lda*, *n*, *ml*, *mu*, *ipvt*, *bx*);

On Entry

agb
is the factorization of general band matrix ***A***, produced by a preceding call to SGBF or DGBF. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 103, where $lda \geq 2ml+mu+16$.

lda
is the leading dimension of the array specified for *agb*. Specified as: a fullword integer; $lda > 0$ and $lda \geq 2ml+mu+16$.

n
is the order of the matrix ***A***. Specified as: a fullword integer; $n > ml$ and $n > mu$.

ml
is the lower band width *ml* of the matrix ***A***. Specified as: a fullword integer; $0 \leq ml < n$.

mu
is the upper band width *mu* of the matrix ***A***. Specified as: a fullword integer; $0 \leq mu < n$.

ipvt
is the integer vector ***ipvt*** of length *n*, produced by a preceding call to SGBF or DGBF. It contains the pivot information necessary to construct matrix ***L*** from the information contained in the array specified for *agb*.

Specified as: a one-dimensional array of (at least) length *n*, containing fullword integers.

bx
is the vector ***b*** of length *n*, containing the right-hand side of the system. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 103.

On Return

bx
is the solution vector $\boldsymbol{x}$ of length n, containing the results of the computation. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 103 on page 552.

Notes

1. The scalar data specified for input arguments *lda*, *n*, *ml*, and *mu* for these subroutines must be the same as that specified for SGBF and DGBF, respectively.
2. The array data specified for input arguments *agb* and *ipvt* for these subroutines must be the same as the corresponding output arguments for SGBF and DGBF, respectively.
3. The entire *lda* by *n* array specified for *agb* must remain unchanged between calls to the factorization and solve subroutines.
4. The vectors and matrices used in this computation must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.
5. This subroutine can be used for tridiagonal matrices ($ml = mu = 1$); however, the tridiagonal subroutines, SGTF/DGTF and SGTS/DGTS, are faster.
6. For a description of how a general band matrix is stored in general-band storage mode in an array, see "General Band Matrix" on page 77.

Function: The real system $\boldsymbol{Ax} = \boldsymbol{b}$ is solved for $\boldsymbol{x}$, where $\boldsymbol{A}$ is a real general band matrix, stored in general-band storage mode, and $\boldsymbol{x}$ and $\boldsymbol{b}$ are vectors. These subroutines use the results of the factorization of matrix $\boldsymbol{A}$, produced by a preceding call to SGBF or DGBF, respectively. The transformed matrix $\boldsymbol{A}$, used by this computation, consists of the upper triangular matrix $\boldsymbol{U}$ and the multipliers necessary to construct $\boldsymbol{L}$ using ***ipvt***, as defined in "Function" on page 549. See reference [38].

Error Conditions

Computational Errors

Note: If the factorization performed by SGBF or DGBF failed due to a singular matrix argument, the results returned by this subroutine are unpredictable, and there may be a divide-by-zero program exception message.

Input-Argument Errors

1. $lda \leq 0$
2. $ml < 0$
3. $ml \geq n$
4. $mu < 0$
5. $mu \geq n$
6. $lda < 2ml+mu+16$

Example: This example shows how to solve the system $\boldsymbol{Ax} = \boldsymbol{b}$, where general band matrix $\boldsymbol{A}$ is the same matrix factored in "Example" on page 550 for SGBF and DGBF. The input for AGB and IPVT in this example is the same as the output for that example.

Call Statement and Input

```
           AGB  LDA   N   ML  MU  IPVT   BX
            |    |    |   |   |    |     |
CALL SGBS( AGB , 23 , 9 , 2 , 3 , IPVT , BX )

IPVT     =  (2, -65534, -131070, -196606, -262142, -327678, -327678,
             -327680, -327680)
BX       =  (4.0000, 5.0000, 9.0000, 10.0000, 11.0000, 12.0000,
             12.0000, 12.0000, 33.0000)
AGB      =(same as output AGB in "Example" on page 550)
```

Output

```
BX       =  (1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000,
             0.9999, 1.0001)
```

SPBF, DPBF, SPBCHF, and DPBCHF—Positive Definite Symmetric Band Matrix Factorization

These subroutines factor positive definite symmetric band matrix ***A***, stored in lower-band-packed storage mode, using:

- Gaussian elimination for SPBF and DPBF
- Cholesky factorization for SPBCHF and DPBCHF

To solve the system of equations with one or more right-hand sides, follow the call to these subroutines with one or more calls to SPBS, DPBS, SPBCHS, or DPBCHS, respectively.

Table 104. Data Types

A	**Subroutine**
Short-precision real	SPBF and SPBCHF
Long-precision real	DPBF and DPBCHF

Notes:

1. The output from these factorization subroutines should be used only as input to the solve subroutines SPBS, DPBS, SPBCHS, and DPBCHS, respectively.
2. For optimal performance:
   - For wide band widths, use _PBCHF.
   - For narrow band widths, use either _PBF or _PBCHF.
   - For very narrow band widths:
     – Use either SPBF or SPBCHF.
     – Use DPBF.

Syntax

Fortran	CALL SPBF | DPBF | SPBCHF | DPBCHF (*apb*, *lda*, *n*, *m*)
C and C++	spbf | dpbf | spbchf | dpbchf (*apb*, *lda*, *n*, *m*);
PL/I	CALL SPBF | DPBF | SPBCHF | DPBCHF (*apb*, *lda*, *n*, *m*);

On Entry

apb
is the positive definite symmetric band matrix ***A*** of order *n*, stored in lower-band-packed storage mode, to be factored. It has a half band width of *m*. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 104. See "Notes" on page 556.

lda
is the leading dimension of the array specified for *apb*. Specified as: a fullword integer; $lda > 0$ and $lda > m$.

n
is the order *n* of matrix ***A***. Specified as: a fullword integer; $n > m$.

m
is the half band width of the matrix ***A***. Specified as: a fullword integer; $0 \leq m < n$.

On Return

apb
is the transformed matrix ***A*** of order *n*, containing the results of the factorization. See "Function." Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 104 on page 555. For further details, see "Notes."

Notes

1. These subroutines can be used for tridiagonal matrices ($m = 1$); however, the tridiagonal subroutines, SPTF/DPTF and SPTS/DPTS, are faster.
2. For SPBF and DPBF when $m > 0$, location APB(2,*n*) is sometimes set to 0.
3. For a description of how a positive definite symmetric band matrix is stored in lower-band-packed storage mode in an array, see "Positive Definite Symmetric Band Matrix" on page 85.

Function: The positive definite symmetric band matrix ***A***, stored in lower-band-packed storage mode, is factored using Gaussian elimination in SPBF and DPBF and Cholesky factorization in SPBCHF and DPBCHF. The transformed matrix ***A*** contains the results of the factorization in packed format. This factorization can then be used by SPBS, DPBS, SPBCHS, and DPBCHS, respectively, to solve the system of equations.

For performance reasons, divides are done in a way that reduces the effective exponent range for which DPBF works properly, when processing narrow band widths; therefore, you may want to scale your problem.

Error Conditions

Resource Errors: Unable to allocate internal work area.

Computational Errors

1. Matrix ***A*** is not positive definite (for SPBF and DPBF).
   - One or more elements of ***D*** contain values less than or equal to 0; all elements of ***D*** are checked. The index *i* of the last nonpositive element encountered is identified in the computational error message.
   - The return code is set to 1.
   - *i* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2104 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see Chapter 4 on page 111.
2. Matrix ***A*** is not positive definite (for SPBCHF and DPBCHF).
   - The leading minor of order *i* has a nonpositive determinant. The order *i* is identified in the computational error message.
   - The return code is set to 1.
   - *i* can be determined at run time by using the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2115 in the ESSL error option table; otherwise, the default value causes your program to be terminate when this error occurs. For details, see Chapter 4 on page 111.

Input-Argument Errors

1. $lda \leq 0$
2. $m < 0$
3. $m \geq n$
4. $m \geq lda$

Example 1: This example shows a factorization of a real positive definite symmetric band matrix ***A*** of order 9, using Gaussian elimination, where on input, matrix ***A*** is:

```
┌                                                   ┐
│ 1.0  1.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0 │
│ 1.0  2.0  2.0  1.0  0.0  0.0  0.0  0.0  0.0 │
│ 1.0  2.0  3.0  2.0  1.0  0.0  0.0  0.0  0.0 │
│ 0.0  1.0  2.0  3.0  2.0  1.0  0.0  0.0  0.0 │
│ 0.0  0.0  1.0  2.0  3.0  2.0  1.0  0.0  0.0 │
│ 0.0  0.0  0.0  1.0  2.0  3.0  2.0  1.0  0.0 │
│ 0.0  0.0  0.0  0.0  1.0  2.0  3.0  2.0  1.0 │
│ 0.0  0.0  0.0  0.0  0.0  1.0  2.0  3.0  2.0 │
│ 0.0  0.0  0.0  0.0  0.0  0.0  1.0  2.0  3.0 │
└                                                   ┘
```

and on output, matrix ***A*** is:

```
┌                                                   ┐
│ 1.0  1.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0 │
│ 1.0  1.0  1.0  1.0  0.0  0.0  0.0  0.0  0.0 │
│ 1.0  1.0  1.0  1.0  1.0  0.0  0.0  0.0  0.0 │
│ 0.0  1.0  1.0  1.0  1.0  1.0  0.0  0.0  0.0 │
│ 0.0  0.0  1.0  1.0  1.0  1.0  1.0  0.0  0.0 │
│ 0.0  0.0  0.0  1.0  1.0  1.0  1.0  1.0  0.0 │
│ 0.0  0.0  0.0  0.0  1.0  1.0  1.0  1.0  1.0 │
│ 0.0  0.0  0.0  0.0  0.0  1.0  1.0  1.0  1.0 │
│ 0.0  0.0  0.0  0.0  0.0  0.0  1.0  1.0  1.0 │
└                                                   ┘
```

where array location `APB(2,9)` is set to 0.0.

Call Statement and Input

```
          APB  LDA  N   M
           |    |   |   |
CALL SPBF( APB , 3 , 9 , 2 )
```

```
        ┌                                                 ┐
        │ 1.0  2.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0 │
APB  =  │ 1.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0   .  │
        │ 1.0  1.0  1.0  1.0  1.0  1.0  1.0   .    .  │
        └                                                 ┘
```

Output

```
        ┌                                                 ┐
        │ 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 │
APB  =  │ 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  0.0 │
        │ 1.0  1.0  1.0  1.0  1.0  1.0  1.0   .    .  │
        └                                                 ┘
```

Example 2: This example shows a Cholesky factorization of the same matrix used in Example 1.

Call Statement and Input

```
             APB  LDA  N   M
              |    |   |   |
CALL SPBCHF( APB , 3 , 9 , 2 )
```

APB =(same as input APB in Example 1)

Output

```
        ⎡                                                ⎤
        ⎢ 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 ⎥
APB  =  ⎢ 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0   .  ⎥
        ⎢ 1.0  1.0  1.0  1.0  1.0  1.0  1.0   .    .  ⎥
        ⎣                                                ⎦
```

SPBS, DPBS, SPBCHS, and DPBCHS—Positive Definite Symmetric Band Matrix Solve

These subroutines solve the system $\boldsymbol{Ax} = \boldsymbol{b}$ for $\boldsymbol{x}$, where $\boldsymbol{A}$ is a positive definite symmetric band matrix, and $\boldsymbol{x}$ and $\boldsymbol{b}$ are vectors. They use the results of the factorization of matrix $\boldsymbol{A}$, produced by a preceding call to SPBF, DPBF, SPBCHF, and DPBCHF, respectively, where:

- Gaussian elimination was used by SPBF and DPBF.
- Cholesky factorization was used by SPBCHF and DPBCHF.

Table 105. Data Types

$\boldsymbol{A}$, $\boldsymbol{b}$, $\boldsymbol{x}$	**Subroutine**
Short-precision real	SPBS and SPBCHS
Long-precision real	DPBS and DPBCHS

Notes:

1. The input to these solve subroutines must be the output from the factorization subroutines SPBF, DPBF, SPBCHF, and DPBCHF, respectively.
2. For performance tradeoffs, see "SPBF, DPBF, SPBCHF, and DPBCHF—Positive Definite Symmetric Band Matrix Factorization" on page 555.

Syntax

Fortran	CALL SPBS | DPBS | SPBCHS | DPBCHS (*apb*, *lda*, *n*, *m*, *bx*)
C and C++	spbs | dpbs | spbchs | dpbchs (*apb*, *lda*, *n*, *m*, *bx*);
PL/I	CALL SPBS | DPBS | SPBCHS | DPBCHS (*apb*, *lda*, *n*, *m*, *bx*);

On Entry

apb
is the factorization of matrix $\boldsymbol{A}$, produced by a preceding call to SPBF or DPBF. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 105. See "Notes" on page 560.

lda
is the leading dimension of the array specified for *apb*. Specified as: a fullword integer; $lda > 0$ and $lda > m$.

n
is the order *n* of matrix $\boldsymbol{A}$. Specified as: a fullword integer; $n > m$.

m
is the half band width of the matrix $\boldsymbol{A}$. Specified as: a fullword integer; $0 \leq m < n$.

bx
is the vector $\boldsymbol{b}$ of length *n*, containing the right-hand side of the system. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 105.

On Return

bx
is the solution vector ***x*** of length *n*, containing the results of the computation. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 105 on page 559.

Notes

1. The scalar data specified for input arguments *lda*, *n*, and *m* for these subroutines must be the same as that specified for SPBF, DPBF, SPBCHF, and DPBCHF, respectively.
2. The array data specified for input argument *apb* for these subroutines must be the same as the corresponding output argument for SPBF, DPBF, SPBCHF, and DPBCHF, respectively.
3. These subroutines can be used for tridiagonal matrices ($m = 1$); however, the tridiagonal subroutines, SPTF/DPTF and SPTS/DPTS, are faster.
4. The vectors and matrices used in this computation must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.
5. For a description of how a positive definite symmetric band matrix is stored in lower-band-packed storage mode in an array, see "Positive Definite Symmetric Band Matrix" on page 85.

Function: The system $\boldsymbol{Ax} = \boldsymbol{b}$ is solved for $\boldsymbol{x}$, where $\boldsymbol{A}$ is a positive definite symmetric band matrix, stored in lower-band-packed storage mode, and $\boldsymbol{x}$ and $\boldsymbol{b}$ are vectors. These subroutines use the results of the factorization of matrix $\boldsymbol{A}$, produced by a preceding call to SPBF, DPBF, SPBCHF, or DPBCHF, respectively.

Error Conditions

Computational Errors: None

Note: If the factorization subroutine resulted in a nonpositive definite matrix, error 2104 for SPBF and DPBF or error 2115 for SPBCHF and DPBCHF, results of these subroutines may be unpredictable.

Input-Argument Errors

1. $lda \leq 0$
2. $m < 0$
3. $m \geq n$
4. $m \geq lda$

Example 1: This example shows how to solve the system $\boldsymbol{Ax} = \boldsymbol{b}$, where matrix $\boldsymbol{A}$ is the same matrix factored in the "Example 1" on page 557 for SPBF and DPBF, using Gaussian elimination.

Call Statement and Input

```
           APB  LDA  N   M   BX
            |    |   |   |   |
CALL SPBS( APB , 3 , 9 , 2 , BX )
```

```
APB       =(same as output APB in "Example 1" on page 557)
BX        =  (3.0, 6.0, 9.0, 9.0, 9.0, 9.0, 9.0, 8.0, 6.0)
```

Output

```
BX        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
```

This example shows how to solve the system ***Ax = b***, where matrix ***A*** is the same matrix factored in the "Example 2" on page 558 for SPBCHF and DPBCHF, using Cholesky factorization.

Call Statement and Input

```
             APB  LDA  N   M   BX
              |    |   |   |   |
CALL SPBCHS( APB , 3 , 9 , 2 , BX )
```

```
APB       =(same as output APB in "Example 2" on page 558)
BX        =  (3.0, 6.0, 9.0, 9.0, 9.0, 9.0, 9.0, 8.0, 6.0)
```

Output

```
BX        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
```

SGTF and DGTF—General Tridiagonal Matrix Factorization

These subroutines compute the standard Gaussian factorization with partial pivoting for tridiagonal matrix ***A***, stored in tridiagonal storage mode. To solve a tridiagonal system with one or more right-hand sides, follow the call to these subroutines with one or more calls to SGTS or DGTS, respectively.

Table 106. Data Types

c, d, e, f	**Subroutine**
Short-precision real	SGTF
Long-precision real	DGTF

Note: The output from these factorization subroutines should be used only as input to the solve subroutines SGTS and DGTS, respectively.

Syntax

Fortran	CALL SGTF | DGTF (*n*, *c*, *d*, *e*, *f*, *ipvt*)
C and C++	sgtf | dgtf (*n*, *c*, *d*, *e*, *f*, *ipvt*);
PL/I	CALL SGTF | DGTF (*n*, *c*, *d*, *e*, *f*, *ipvt*);

On Entry

n
is the order *n* of tridiagonal matrix ***A***. Specified as: a fullword integer; $n \geq 0$.

c
is the vector ***c***, containing the lower subdiagonal of matrix ***A*** in positions 2 through *n* in an array, referred to as `C`. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 106.

d
is the vector ***d***, containing the main diagonal of matrix ***A***, in positions 1 through *n* in an array, referred to as `D`. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 106.

e
is the vector ***e***, containing the upper subdiagonal of matrix ***A***, in positions 1 through $n-1$ in an array, referred to as `E`. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 106.

f
See On Return.

ipvt
See On Return.

On Return

c
is the vector ***c***, containing part of the factorization of matrix ***A*** in positions 1 through *n* in an array, referred to as `C`. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 106.

d
is the vector ***d***, containing part of the factorization of matrix ***A*** in an array, referred to as D. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 106.

e
is the vector ***e***, containing part of the factorization of the matrix ***A*** in positions 1 through *n* in an array, referred to as E. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 106 on page 562.

f
is the vector ***f***, containing part of the factorization of matrix ***A*** in the first *n* positions in an array, referred to as F. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 106 on page 562.

ipvt
is the integer vector ***ipvt*** of length *n*, containing the pivot information. Returned as: a one-dimensional array of (at least) length *n*, containing fullword integers.

Notes

1. For a description of how tridiagonal matrices are stored, see "General Tridiagonal Matrix" on page 90.
2. ***ipvt*** is not a permutation vector in the strict sense. It is used to record column interchanges in the tridiagonal matrix due to partial pivoting.
3. The factorization matrix ***A*** is stored in nonstandard format.

Function: The standard Gaussian elimination with partial pivoting of tridiagonal matrix ***A*** is computed. The factorization is returned by overwriting input arrays C, D, and E, and by writing into output array F, along with pivot information in vector ***ipvt***. This factorization can then be used by SGTS or DGTS, respectively, to solve tridiagonal systems of linear equations. See references [43], [51], [52], and [84]. If *n* is 0, no computation is performed.

Error Conditions

Computational Errors: Matrix ***A*** is singular or nearly singular.

- A pivot element has a value that cannot be reciprocated or is equal to 0. The index *i* of the element is identified in the computational error message.
- The return code is set to 1.
- *i* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2105 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see "What Can You Do about ESSL Computational Errors?" on page 48.

Input-Argument Errors: $n < 0$

Example: This example shows how to factor the following tridiagonal matrix ***A*** of order 4:

$$\begin{bmatrix} 2.0 & 2.0 & 0.0 & 0.0 \\ 1.0 & 3.0 & 2.0 & 0.0 \\ 0.0 & 1.0 & 3.0 & 2.0 \\ 0.0 & 0.0 & 1.0 & 3.0 \end{bmatrix}$$

Call Statement and Input

```
           N   C   D   E   F   IPVT
           |   |   |   |   |    |
CALL DGTF( 4 , C , D , E , F , IPVT )

C        =  ( . , 1.0, 1.0, 1.0)
D        =  (2.0, 3.0, 3.0, 3.0)
E        =  (2.0, 2.0, 2.0, . )
```

Output

```
C        =  ( . , -0.5, -0.5, -0.5)
D        =  (-0.5, -0.5, -0.5, -0.5)
E        =  (2.0, 2.0, 2.0, . )
IPVT     =  (X'00', X'00', X'00', X'00')
```

Notes

1. F is stored in an internal format and is passed unchanged to the solve subroutine.
2. A "." means you do not have to store a value in that position in the array. However, these storage positions are required and may be overwritten during the computation.

SGTS and DGTS—General Tridiagonal Matrix Solve

These subroutines solve a tridiagonal system of linear equations using the factorization of tridiagonal matrix *A*, stored in tridiagonal storage mode, produced by SGTF or DGTF, respectively.

Table 107. Data Types

c, ***d***, ***e***, ***f***, ***b***, ***x***	**Subroutine**
Short-precision real	SGTS
Long-precision real	DGTS

Note: The input to these solve subroutines must be the output from the factorization subroutines SGTF and DGTF, respectively.

Syntax

Fortran	CALL SGTS | DGTS (*n*, *c*, *d*, *e*, *f*, *ipvt*, *bx*)
C and C++	sgts | dgts (*n*, *c*, *d*, *e*, *f*, *ipvt*, *bx*);
PL/I	CALL SGTS | DGTS (*n*, *c*, *d*, *e*, *f*, *ipvt*, *bx*);

On Entry

n
is the order *n* of tridiagonal matrix ***A***. Specified as: a fullword integer; $n \geq 0$.

c
is the vector ***c***, containing part of the factorization of matrix ***A*** from SGTF or DGTF, respectively, in an array, referred to as `C`. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 107.

d
is the vector ***d***, containing part of the factorization of matrix ***A*** from SGTF or DGTF, respectively, in an array, referred to as `D`. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 107.

e
is the vector ***e***, containing part of the factorization of matrix ***A*** from SGTF or DGTF, respectively, in an array, referred to as `E`. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 107.

f
is the vector ***f***, containing part of the factorization of matrix ***A*** from SGTF or DGTF, respectively, in an array, referred to as `F`. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 107.

ipvt
is the integer vector ***ipvt*** of length *n*, containing the pivot information, produced by a preceding call to SGTF and DGTF, respectively. Specified as: a one-dimensional array of (at least) length *n*, containing fullword integers.

bx
is the vector ***b*** of length *n*, containing the right-hand side of the system in the first *n* positions in an array, referred to as `BX`. Specified as: a one-dimensional array of (at least) length *n*+1, containing numbers of the data type indicated in Table 107. For details on specifying the length, see "Notes" on page 566.

On Return

bx
is the solution vector ***x*** (at least) of length *n*, containing the solution of the tridiagonal system in the first *n* positions in an array, referred to as BX. Returned as: a one-dimensional array, of (at least) length (*n*+1), containing numbers of the data type indicated in Table 107 on page 565. For details about the length, see "Notes."

Notes

1. For a description of how tridiagonal matrices are stored, see "General Tridiagonal Matrix" on page 90.
2. Array BX can have a length of *n* if memory location BX(*n*+1) is addressable—that is, not in read-protected storage. If it is in read-protected storage, array BX must have a length of *n*+1. In both cases, the vector ***b*** (on input) and vector ***x*** (on output) reside in positions 1 through *n* in array BX. Array location BX(*n*+1) is not altered by these subroutines.

Function: Given the factorization produced by SGTF or DGTF, respectively, these subroutines use the standard forward elimination and back substitution to solve the tridiagonal system $\boldsymbol{Ax} = \boldsymbol{b}$, where ***A*** is a general tridiagonal matrix. See references [43], [51], [52], and [84].

Error Conditions

Computational Errors: None

Input-Argument Errors: $n < 0$

Example: This example solves the tridiagonal system $\boldsymbol{Ax} = \boldsymbol{b}$, where matrix ***A*** is the same matrix factored in "Example" on page 564 for SGTF and DGTF, and where:

```
b = (4.0, 6.0, 6.0, 4.0)
x = (1.0, 1.0, 1.0, 1.0)
```

Call Statement and Input

```
           N   C   D   E   F   IPVT   BX
           |   |   |   |   |    |     |
CALL DGTS( 4 , C , D , E , F , IPVT , BX )
```

```
C       =(same as output C in "Example" on page 564)
D       =(same as output D in "Example" on page 564)
E       =(same as output E in "Example" on page 564)
F       =(same as output F in "Example" on page 564)
IPVT    =(same as output IPVT in "Example" on page 564)
BX      =(4.0, 6.0, 6.0, 4.0, . )
```

Output

```
BX      =  (1.0, 1.0, 1.0, 1.0, . )
```

SGTNP, DGTNP, CGTNP, and ZGTNP—General Tridiagonal Matrix Combined Factorization and Solve with No Pivoting

These subroutines solve the tridiagonal system $\boldsymbol{Ax} = \boldsymbol{b}$ using Gaussian elimination, where tridiagonal matrix $\boldsymbol{A}$ is stored in tridiagonal storage mode.

Table 108. Data Types

c, ***d***, ***e***, ***b***, ***x***	**Subroutine**
Short-precision real	SGTNP
Long-precision real	DGTNP
Short-precision complex	CGTNP
Long-precision complex	ZGTNP

Note: In general, these subroutines provide better performance than the _GTNPF and _GTNPS subroutines; however, in the following instances, you get better performance by using _GTNPF and _GTNPS:

- For small *n*
- When performing a single factorization followed by multiple solves

Syntax

Fortran	CALL SGTNP | DGTNP | CGTNP | ZGTNP (*n*, *c*, *d*, *e*, *bx*)
C and C++	sgtnp | dgtnp | cgtnp | zgtnp (*n*, *c*, *d*, *e*, *bx*);
PL/I	CALL SGTNP | DGTNP | CGTNP | ZGTNP (*n*, *c*, *d*, *e*, *bx*);

On Entry

n
is the order *n* of tridiagonal matrix ***A***. Specified as: a fullword integer; $n \geq 0$.

c
is the vector ***c***, containing the lower subdiagonal of matrix ***A*** in positions 2 through *n* in an array, referred to as C. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 108. On output, C is overwritten; that is, the original input is not preserved.

d
is the vector ***d***, containing the main diagonal of matrix ***A*** in positions 1 through *n* in an array, referred to as D. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 108. On output, D is overwritten; that is, the original input is not preserved.

e
is the vector ***e***, containing the upper subdiagonal of matrix ***A*** in positions 1 through $n-1$ in an array, referred to as E. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 108. On output, E is overwritten; that is, the original input is not preserved.

bx
is the vector ***b***, containing the right-hand side of the system in the first *n* positions in an array, referred to as BX. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 108.

On Return

bx
is the solution vector ***x*** of length *n*, containing the solution of the tridiagonal system in the first *n* positions in an array, referred to as BX. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 108 on page 567.

Note: For a description of how tridiagonal matrices are stored, see "General Tridiagonal Matrix" on page 90.

Function: The solution of the tridiagonal system $\boldsymbol{Ax} = \boldsymbol{b}$ is computed by Gaussian elimination.

No pivoting is done. Therefore, these subroutines should not be used when pivoting is necessary to maintain the numerical accuracy of the solution. Overflow may occur if small main diagonal elements are generated. Underflow or accuracy loss may occur if large main diagonal elements are generated.

For performance reasons, complex divides are done without scaling. Computing the inverse in this way restricts the range of numbers for which the ZGTNP subroutine works properly.

For performance reasons, divides are done in a way that reduces the effective exponent range for which DGTNP and ZGTNP work properly; therefore, you may want to scale your problem, such that the diagonal elements are close to 1.0 for DGTNP and (1.0, 0.0) for ZGTNP.

Error Conditions

Computational Errors: None

Input-Argument Errors: $n < 0$

Example 1: This example shows a factorization of the real tridiagonal matrix ***A,*** of order 4:

$$\begin{bmatrix} 7.0 & 4.0 & 0.0 & 0.0 \\ 1.0 & 8.0 & 5.0 & 0.0 \\ 0.0 & 2.0 & 9.0 & 6.0 \\ 0.0 & 0.0 & 3.0 & 10.0 \end{bmatrix}$$

It then finds the solution of the tridiagonal system $\boldsymbol{Ax} = \boldsymbol{b}$, where $\boldsymbol{b}$ is:

```
(11.0, 14.0, 17.0, 13.0)
```

and $\boldsymbol{x}$ is:

```
(1.0, 1.0, 1.0, 1.0)
```

On output, arrays C, D, and E are overwritten.

Call Statement and Input

```
            N   C   D   E   BX
            |   |   |   |   |
CALL DGTNP( 4 , C , D , E , BX )

C        =  ( . , 1.0, 2.0, 3.0)
D        =  (7.0, 8.0, 9.0, 10.0)
E        =  (4.0, 5.0, 6.0, . )
BX       =  (11.0, 14.0, 17.0, 13.0)
```

Output

```
BX       =  (1.0, 1.0, 1.0, 1.0)
```

Example 2: This example shows a factorization of the complex tridiagonal matrix ***A***, of order 4:

$$\begin{bmatrix} (7.0, 7.0) & (4.0, 4.0) & (0.0, 0.0) & (0.0, 0.0) \\ (1.0, 1.0) & (8.0, 8.0) & (5.0, 5.0) & (0.0, 0.0) \\ (0.0, 0.0) & (2.0, 2.0) & (9.0, 9.0) & (6.0, 6.0) \\ (0.0, 0.0) & (0.0, 0.0) & (3.0, 3.0) & (10.0, 10.0) \end{bmatrix}$$

It then finds the solution of the tridiagonal system $\boldsymbol{Ax} = \boldsymbol{b}$, where $\boldsymbol{b}$ is:

```
((-11.0,19.0), (-14.0,50.0), (-17.0,93.0), (-13.0,85.0))
```

and $\boldsymbol{x}$ is:

```
((1.0,-1.0), (2.0,-2.0), (3.0,-3.0), (4.0,-4.0))
```

On output, arrays `C`, `D`, and `E` are overwritten.

Call Statement and Input

```
            N   C   D   E   BX
            |   |   |   |   |
CALL ZGTNP( 4 , C , D , E , BX )

C        =  ( . , (1.0, 1.0), (2.0, 2.0), (3.0, 3.0))
D        =  ((7.0, 7.0), (8.0, 8.0), (9.0, 9.0), (10.0, 10.0))
E        =  ((4.0, 4.0), (5.0, 5.0), (6.0, 6.0), . )
BX       =  ((-11.0, 19.0), (-14.0, 50.0), (-17.0, 93.0), (-13.0, 85.0))
```

Output

```
BX       =  ((0.0, 1.0), (1.0, 2.0), (2.0, 3.0), (3.0, 4.0))
```

SGTNPF, DGTNPF, CGTNPF, and ZGTNPF—General Tridiagonal Matrix Factorization with No Pivoting

These subroutines factor tridiagonal matrix ***A***, stored in tridiagonal storage mode, using Gaussian elimination. To solve a tridiagonal system of linear equations with one or more right-hand sides, follow the call to these subroutines with one or more calls to SGTNPS, DGTNPS, CGTNPS, or ZGTNPS, respectively.

Table 109. Data Types

c, ***d***, ***e***	**Subroutine**
Short-precision real	SGTNPF
Long-precision real	DGTNPF
Short-precision complex	CGTNPF
Long-precision complex	ZGTNPF

Notes:

1. The output from these factorization subroutines should be used only as input to the solve subroutines SGTNPS, DGTNPS, CGTNPS, and ZGTNPS, respectively.
2. In general, the _GTNP subroutines provide better performance than the _GTNPF and _GTNPS subroutines; however, in the following instances, you get better performance by using _GTNPF and _GTNPS:
   - For small *n*
   - When performing a single factorization followed by multiple solves

Syntax

Fortran	CALL SGTNPF | DGTNPF | CGTNPF | ZGTNPF (*n*, *c*, *d*, *e*, *iopt*)
C and C++	sgtnpf | dgtnpf | cgtnpf | zgtnpf (*n*, *c*, *d*, *e*, *iopt*);
PL/I	CALL SGTNPF | DGTNPF | CGTNPF | ZGTNPF (*n*, *c*, *d*, *e*, *iopt*);

On Entry

n
is the order *n* of tridiagonal matrix ***A***. Specified as: a fullword integer; $n \geq 0$.

c
is the vector ***c***, containing the lower subdiagonal of matrix ***A*** in positions 2 through *n* in an array, referred to as `C`. Specified as: a one-dimensional array, of (at least) length `n`, containing numbers of the data type indicated in Table 109.

d
is the vector ***d***, containing the main diagonal of matrix ***A*** in positions 1 through *n* in an array, referred to as `D`. Specified as: a one-dimensional array, of (at least) length *n*, containing numbers of the data type indicated in Table 109.

e
is the vector ***e***, containing the upper subdiagonal of matrix ***A*** in positions 1 through *n*−1 in an array, referred to as `E`. Specified as: a one-dimensional array, of (at least) length *n*, containing numbers of the data type indicated in Table 109.

iopt
indicates the type of computation to be performed, where:

If *iopt* = 0 or 1, Gaussian elimination is used to factor the matrix.

Specified as: a fullword integer; *iopt* = 0 or 1.

On Return

c
is the vector ***c***, containing part of the factorization of matrix ***A*** in positions 1 through *n* in an array, referred to as C. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 109 on page 570.

d
is the vector ***d***, containing part of the factorization of matrix ***A*** in an array, referred to as D. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 109 on page 570.

e
is the vector ***e***, containing part of the factorization of matrix ***A*** in positions 1 through *n* in an array, referred to as E. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 109 on page 570. It has the same length as E on entry.

Note: For a description of how tridiagonal matrices are stored, see "General Tridiagonal Matrix" on page 90.

Function: The factorization of a diagonally-dominant tridiagonal matrix ***A*** is computed using Gaussian elimination, This factorization can then be used by SGTNPS, DGTNPS, CGTNPS, or ZGTNPS respectively, to solve the tridiagonal systems of linear equations. See reference [71].

No pivoting is done by these subroutines. Therefore, these subroutines should not be used when pivoting is necessary to maintain the numerical accuracy of the solution. Overflow may occur if small main diagonal elements are generated. Underflow or accuracy loss may occur if large main diagonal elements are generated.

For performance reasons, complex divides are done without scaling. Computing the inverse in this way restricts the range of numbers for which ZGTNPF works properly.

For performance reasons, divides are done in a way that reduces the effective exponent range for which DGTNPF and ZGTNPF work properly; therefore, you may want to scale your problem, such that the diagonal elements are close to 1.0 for DGTNPF and (1.0, 0.0) for ZGTNPF.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. $n < 0$
2. *iopt* $\neq$ 0 or 1

Example 1: This example shows a factorization of the tridiagonal matrix ***A***, of order 4:

$$\begin{bmatrix} 1.0 & 1.0 & 0.0 & 0.0 \\ 1.0 & 2.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 3.0 & 1.0 \\ 0.0 & 0.0 & 1.0 & 1.0 \end{bmatrix}$$

Call Statement and Input

```
            N   C   D   E   IOPT
            |   |   |   |    |
CALL DGTNPF( 4 , C , D , E ,  0   )

C       =  ( . , 1.0, 1.0, 1.0)
D       =  (1.0, 2.0, 3.0, 1.0)
E       =  (1.0, 1.0, 1.0, .  )
```

Output

```
C       =  ( . , -1.0, -1.0, 1.0)
D       =  (-1.0, -1.0, -1.0, -1.0)
E       =  (1.0, 1.0, -1.0, . )
```

Example 2: This example shows a factorization of the tridiagonal matrix ***A***, of order 4:

$$\begin{bmatrix} (7.0, 7.0) & (4.0, 4.0) & (0.0, 0.0) & (0.0, 0.0) \\ (1.0, 1.0) & (8.0, 8.0) & (5.0, 5.0) & (0.0, 0.0) \\ (0.0, 0.0) & (2.0, 2.0) & (9.0, 9.0) & (6.0, 6.0) \\ (0.0, 0.0) & (0.0, 0.0) & (3.0, 3.0) & (10.0, 10.0) \end{bmatrix}$$

Call Statement and Input

```
            N   C   D   E   IOPT
            |   |   |   |    |
CALL ZGTNPF( 4 , C , D , E ,  0   )

C       =  ( . , (1.0, 1.0), (2.0, 2.0), (3.0, 3.0))
D       =  ((7.0, 7.0), (8.0, 8.0), (9.0, 9.0), (10.0, 10.0))
E       =  ((4.0, 4.0), (5.0, 5.0), (6.0, 6.0), . )
```

Output

```
C       =  ( . , (-0.142, 0.0), (-0.269, 0.0), (3.0, 3.0))
D       =  ((-0.0714, 0.0714), (-0.0673, 0.0673), (-0.0854, 0.0854),
            (-0.05, 0.05))
E       =  ((4.0, 4.0), (5.0, 5.0), (-0.6, 0.0), . )
```

Notes

1. A "**.**" means you do not have to store a value in that position in the array. However, these storage positions are required and may be overwritten during the computation.

SGTNPS, DGTNPS, CGTNPS, and ZGTNPS—General Tridiagonal Matrix Solve with No Pivoting

These subroutines solve a tridiagonal system of equations using the factorization of matrix ***A***, stored in tridiagonal storage mode, produced by SGTNPF, DGTNPF, CGTNPF, or ZGTNPF, respectively.

Table 110. Data Types

c, ***d***, ***e***, ***b***, ***x***	**Subroutine**
Short-precision real	SGTNPS
Long-precision real	DGTNPS
Short-precision complex	CGTNPS
Long-precision complex	ZGTNPS

Note: The input to these solve subroutines must be the output from the factorization subroutines SGTNPF, DGTNPF, CGTNPF, and ZGTNPF, respectively.

Syntax

Fortran	CALL SGTNPS | DGTNPS | CGTNPS | ZGTNPS (*n*, *c*, *d*, *e*, *bx*)
C and C++	sgtnps | dgtnps | cgtnps | zgtnps (*n*, *c*, *d*, *e*, *bx*);
PL/I	CALL SGTNPS | DGTNPS | CGTNPS | ZGTNPS (*n*, *c*, *d*, *e*, *bx*);

On Entry

n
is the order *n* of tridiagonal matrix ***A***. Specified as: a fullword integer; $n \geq 0$.

c
is the vector ***c***, containing part of the factorization of matrix ***A*** from SGTNPF, DGTNPF, CGTNPF, and ZGTNPF, respectively, in an array, referred to as `C`. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 110.

d
is the vector ***d***, containing part of the factorization of matrix ***A*** from SGTNPF, DGTNPF, CGTNPF, and ZGTNPF, respectively, in an array, referred to as `D`. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 110.

e
is the vector ***e***, containing part of the factorization of matrix ***A*** from SGTNPF, DGTNPF, CGTNPF, and ZGTNPF, respectively, in an array, referred to as `E`. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 110.

bx
is the vector ***b***, containing the right-hand side of the system in the first *n* positions in an array, referred to as `BX`. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 110.

On Return

bx
is the solution vector ***x*** of length *n*, containing the solution of the tridiagonal system in the first *n* positions in an array, referred to as BX. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 110 on page 573.

Note: For a description of how tridiagonal matrices are stored, see "General Tridiagonal Matrix" on page 90.

Function: The solution of tridiagonal system $\boldsymbol{Ax} = \boldsymbol{b}$ is computed using the factorization produced by SGTNPF, DGTNPF, CGTNPF, or ZGTNPF, respectively. The factorization is based on Gaussian elimination. See reference [71].

Error Conditions

Computational Errors: None

Input-Argument Errors: $n < 0$

Example 1: This example finds the solution of tridiagonal system $\boldsymbol{Ax} = \boldsymbol{b}$, where matrix ***A*** is the same matrix factored in "Example 1" on page 572 for SGTNPF and DGTNPF. ***b*** is:

```
(2.0, 4.0, 5.0, 2.0)
```

and ***x*** is:

```
(1.0, 1.0, 1.0, 1.0)
```

Call Statement and Input

```
             N   C   D   E   BX
             |   |   |   |   |
CALL DGTNPS( 4 , C , D , E , BX )
```

```
C       =(same as output C in "Example 1" on page 572)
D       =(same as output D in "Example 1" on page 572)
E       =(same as output E in "Example 1" on page 572)
BX      =(2.0, 4.0, 5.0, 2.0)
```

Output

```
BX      =  (1.0, 1.0, 1.0, 1.0)
```

Example 2: This example finds the solution of tridiagonal system $\boldsymbol{Ax} = \boldsymbol{b}$, where matrix ***A*** is the same matrix factored in "Example 2" on page 572 for CGTNPF and ZGTNPF. ***b*** is:

```
((-11.0,19.0), (-14.0,50.0), (-17.0,93.0), (-13.0,85.0))
```

and ***x*** is:

```
((0.0,1.0), (1.0,2.0), (2.0,3.0), (3.0,4.0))
```

Call Statement and Input

```
             N   C   D   E   BX
             |   |   |   |   |
CALL ZGTNPS( 4 , C , D , E , BX )
```

C =(same as output C in "Example 2" on page 572)
D =(same as output D in "Example 2" on page 572)
E =(same as output E in "Example 2" on page 572)
BX =((-11.0, 19.0), (-14.0, 50.0), (-17.0, 93.0), (-13.0, 85.))

Output

```
BX       =  ((0.0, 1.0), (1.0, 2.0), (2.0, 3.0), (3.0, 4.0))
```

SPTF and DPTF—Positive Definite Symmetric Tridiagonal Matrix Factorization

These subroutines factor symmetric tridiagonal matrix *A*, stored in symmetric-tridiagonal storage mode, using Gaussian elimination. To solve a tridiagonal system of linear equations with one or more right-hand sides, follow the call to these subroutines with one or more calls to SPTS or DPTS, respectively.

Table 111. Data Types

c, ***d***	**Subroutine**
Short-precision real	SPTF
Long-precision real	DPTF

Note: The output from these factorization subroutines should be used only as input to the solve subroutines SPTS and DPTS, respectively.

Syntax

Fortran	CALL SPTF | DPTF (*n*, *c*, *d*, *iopt*)
C and C++	sptf | dptf (*n*, *c*, *d*, *iopt*);
PL/I	CALL SPTF | DPTF (*n*, *c*, *d*, *iopt*);

On Entry

n
is the order *n* of tridiagonal matrix *A*. Specified as: a fullword integer; $n \geq 0$.

c
is the vector ***c***, containing the off-diagonal of matrix ***A*** in positions 2 through *n* in an array, referred to as `C`. Specified as: a one-dimensional array, of (at least) length *n*, containing numbers of the data type indicated in Table 111.

d
is the vector ***d***, containing the main diagonal of matrix ***A*** in positions 1 through *n* in an array referred to as `D`. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 111.

iopt
indicates the type of computation to be performed, where:

If *iopt* = 0 or 1, Gaussian elimination is used to factor the matrix.

Specified as: a fullword integer; *iopt* = 0 or 1.

On Return

c
is the vector ***c***, containing part of the factorization of matrix ***A*** in an array, referred to as `C`. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 111.

d
is the vector ***d***, containing part of the factorization of matrix ***A*** in positions 1 through *n* in an array, referred to as `D`. Returned as: a one-dimensional array of (at least)

length *n*, containing numbers of the data type indicated in Table 111. It has the same length as D on entry.

Note: For a description of how positive definite symmetric tridiagonal matrices are stored, see "Positive Definite Symmetric Tridiagonal Matrix" on page 92.

Function: The factorization of positive definite symmetric tridiagonal matrix ***A*** is computed using Gaussian elimination. This factorization can then be used by SPTS or DPTS, respectively, to solve the tridiagonal systems of linear equations. See reference [71].

No pivoting is done. Therefore, these subroutines should not be used when pivoting is necessary to maintain the numerical accuracy of the solution. Overflow may occur if small pivots are generated.

For performance reasons, divides are done in a way that reduces the effective exponent range for which DPTF works properly; therefore, you may want to scale your problem, such that the diagonal elements are close to 1.0 for DPTF.

Error Conditions

Computational Errors: None

Note: There is no test for positive definiteness in these subroutines.

Input-Argument Errors

1. $n < 0$
2. *iopt* $\neq$ 0 or 1

Example: This example shows a factorization of the tridiagonal matrix ***A,*** of order 4:

$$\begin{bmatrix} 1.0 & 1.0 & 0.0 & 0.0 \\ 1.0 & 2.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 3.0 & 1.0 \\ 0.0 & 0.0 & 1.0 & 1.0 \end{bmatrix}$$

Call Statement and Input

```
          N   C   D  IOPT
          |   |   |   |
CALL DPTF( 4 , C , D , 0  )

C        =  ( . , 1.0, 1.0, 1.0)
D        =  (1.0, 2.0, 3.0, 1.0)
```

Output

```
C        =  ( . , -1.0, -1.0, -1.0)
D        =  (-1.0, -1.0, -1.0, -1.0)
```

Notes

1. A "." means you do not have to store a value in that position in the array. However, these storage positions are required and may be overwritten during the computation.

SPTS and DPTS—Positive Definite Symmetric Tridiagonal Matrix Solve

These subroutines solve a positive definite symmetric tridiagonal system of equations using the factorization of matrix ***A***, stored in symmetric-tridiagonal storage mode, produced by SPTF and DPTF, respectively.

Table 112. Data Types

c, ***d***, ***b***, ***x***	**Subroutine**
Short-precision real	SPTS
Long-precision real	DPTS

Note: The input to these solve subroutines must be the output from the factorization subroutines SPTF and DPTF, respectively.

Syntax

Fortran	CALL SPTS | DPTS (*n*, *c*, *d*, *bx*)
C and C++	spts | dpts (*n*, *c*, *d*, *bx*);
PL/I	CALL SPTS | DPTS (*n*, *c*, *d*, *bx*);

On Entry

n
is the order *n* of tridiagonal matrix ***A***. Specified as: a fullword integer; $n \geq 0$.

c
is the vector ***c***, containing part of the factorization of matrix ***A*** from SPTF or DPTF, respectively, in an array, referred to as `C`. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 112.

d
is the vector ***d***, containing part of the factorization of matrix ***A*** from SPTF or DPTF, respectively, in an array, referred to as `D`. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 112.

bx
is the vector ***b***, containing the right-hand side of the system in the first *n* positions in an array, referred to as `BX`. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 112.

On Return

bx
is the solution vector ***x*** of length *n*, containing the solution of the tridiagonal system in the first *n* positions in an array, referred to as `BX`. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 112.

Note: For a description of how tridiagonal matrices are stored, see "Positive Definite or Negative Definite Symmetric Matrix" on page 69.

Function: The solution of positive definite symmetric tridiagonal system $\boldsymbol{Ax} = \boldsymbol{b}$ is computed using the factorization produced by SPTF or DPTF, respectively. The factorization is based on Gaussian elimination. See reference [71].

Error Conditions

Computational Errors: None

Input-Argument Errors: $n < 0$

Example: This example finds the solution of tridiagonal system $\boldsymbol{Ax} = \boldsymbol{b}$, where matrix $\boldsymbol{A}$ is the same matrix factored in "Example" on page 577 for SPTF and DPTF. $\boldsymbol{b}$ is:

```
(2.0, 4.0, 5.0, 2.0)
```

and $\boldsymbol{x}$ is:

```
(1.0, 1.0, 1.0, 1.0)
```

Call Statement and Input

```
           N   C   D   BX
           |   |   |   |
CALL DPTS( 4 , C , D , BX )

C        =  ( . , -1.0, -1.0, -1.0)
D        =  (-1.0, -1.0, -1.0, -1.0)
BX       =  (2.0, 4.0, 5.0, 2.0)
```

Output

```
BX       =  (1.0, 1.0, 1.0, 1.0)
```

STBSV, DTBSV, CTBSV, and ZTBSV—Triangular Band Equation Solve

STBSV and DTBSV solve one of the following triangular banded systems of equations with a single right-hand side, using the vector $\boldsymbol{x}$ and triangular band matrix $\boldsymbol{A}$ or its transpose:

Solution	Equation
1. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-1}\boldsymbol{x}$	$\boldsymbol{Ax} = \boldsymbol{b}$
2. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-T}\boldsymbol{x}$	$\boldsymbol{A}^{T}\boldsymbol{x} = \boldsymbol{b}$

CTBSV and ZTBSV solve one of the following triangular banded systems of equations with a single right-hand side, using the vector $\boldsymbol{x}$ and triangular band matrix $\boldsymbol{A}$, its transpose, or its conjugate transpose:

Solution	Equation
1. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-1}\boldsymbol{x}$	$\boldsymbol{Ax} = \boldsymbol{b}$
2. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-T}\boldsymbol{x}$	$\boldsymbol{A}^{T}\boldsymbol{x} = \boldsymbol{b}$
3. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-H}\boldsymbol{x}$	$\boldsymbol{A}^{H}\boldsymbol{x} = \boldsymbol{b}$

Matrix $\boldsymbol{A}$ can be either upper or lower triangular and is stored in upper- or lower-triangular-band-packed storage mode, respectively.

Table 113. Data Types

$\boldsymbol{A}$, $\boldsymbol{x}$	Subprogram
Short-precision real	STBSV
Long-precision real	DTBSV
Short-precision complex	CTBSV
Long-precision complex	ZTBSV

Syntax

Fortran	CALL STBSV | DTBSV | CTBSV | ZTBSV (*uplo*, *trans*, *diag*, *n*, *k*, *a*, *lda*, *x*, *incx*)
C and C++	stbsv | dtbsv | ctbsv | ztbsv (*uplo*, *trans*, *diag*, *n*, *k*, *a*, *lda*, *x*, *incx*);
PL/I	CALL STBSV | DTBSV | CTBSV | ZTBSV (*uplo*, *trans*, *diag*, *n*, *k*, *a*, *lda*, *x*, *incx*);

On Entry

uplo
indicates whether matrix $\boldsymbol{A}$ is an upper or lower triangular band matrix, where:

If *uplo* = 'U', $\boldsymbol{A}$ is an upper triangular matrix.

If *uplo* = 'L', $\boldsymbol{A}$ is a lower triangular matrix.

Specified as: a single character. It must be 'U' or 'L'.

trans
indicates the form of matrix $\boldsymbol{A}$ used in the system of equations, where:

If *trans* = 'N', $\boldsymbol{A}$ is used, resulting in solution 1.

If *trans* = 'T', A^T is used, resulting in solution 2.

If *trans* = 'C', A^H is used, resulting in solution 3.

Specified as: a single character. It must be 'N', 'T', or 'C'.

diag
indicates the characteristics of the diagonal of matrix ***A***, where:

If *diag* = 'U', ***A*** is a unit triangular matrix.

If *diag* = 'N', ***A*** is not a unit triangular matrix.

Specified as: a single character. It must be 'U' or 'N'.

n
is the order of triangular band matrix ***A***. Specified as: a fullword integer; $n \geq 0$.

k
is the upper or lower band width *k* of the matrix ***A***. Specified as: a fullword integer; $k \geq 0$.

a
is the upper or lower triangular band matrix ***A*** of order *n*, stored in upper- or lower-triangular-band-packed storage mode, respectively. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 113 on page 580.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; $lda > 0$ and $lda \geq k+1$.

x
is the vector ***x*** of length *n*, containing the right-hand side of the triangular system to be solved. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 113 on page 580.

incx
is the stride for vector ***x***. Specified as: a fullword integer; $incx > 0$ or $incx < 0$.

On Return

x
is the solution vector ***x*** of length *n*, containing the results of the computation. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 113 on page 580.

Notes

1. These subroutines accept lowercase letters for the *uplo*, *trans*, and *diag* arguments.
2. For STBSV and DTBSV, if you specify 'C' for the *trans* argument, it is interpreted as though you specified 'T'.
3. Matrix ***A*** and vector ***x*** must have no common elements; otherwise, results are unpredictable.
4. For unit triangular matrices, the elements of the diagonal are assumed to be 1.0 for real matrices and (1.0, 0.0) for complex matrices, and you do not need to set these values in the array.
5. For both upper and lower triangular band matrices, if you specify $k \geq n$, ESSL assumes, **for purposes of the computation only**, that the upper or lower band width of matrix ***A*** is $n-1$; that is, it processes matrix ***A*** of order *n*, as though it is a (nonbanded) triangular matrix. However, ESSL uses the original value for *k* **for the purposes of finding the locations** of element a_{11} and all other elements in the array

specified for ***A***, as described in "Triangular Band Matrix" on page 86. For an illustration of this technique, see "Example 3" on page 584.

6. For a description of triangular band matrices and how they are stored in upper- and lower-triangular-band-packed storage mode, see "Triangular Band Matrix" on page 86.
7. If you are using a lower triangular band matrix, it may save your program some time if you use this alternate approach instead of using lower-triangular-band-packed storage mode. Leave matrix ***A*** in full-matrix storage mode when you pass it to ESSL and specify the *lda* argument to be *lda*+1, which is the leading dimension of matrix ***A*** plus 1. ESSL then processes the matrix elements in the same way as though you had set them up in lower-triangular-band-packed storage mode.

Function: These subroutines solve a triangular banded system of equations with a single right-hand side. The solution, ***x***, may be any of the following, where triangular band matrix ***A***, its transpose, or its conjugate transpose is used, and where ***A*** can be either upper- or lower-triangular:

1. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-1}\boldsymbol{x}$
2. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-T}\boldsymbol{x}$
3. $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-H}\boldsymbol{x}$ (for CTBSV and ZTBSV only)

where:

x is a vector of length *n*.
A is an upper or lower triangular band matrix of order *n*, stored in upper- or lower-triangular-band-packed storage mode, respectively.

See references [34], [46], and [38]. If *n* is 0, no computation is performed.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. $n < 0$
2. $k < 0$
3. $lda \leq 0$
4. $lda < k+1$
5. $incx = 0$
6. *uplo* ≠ 'L' or 'U'
7. *trans* ≠ 'T', 'N', or 'C'
8. *diag* ≠ 'N' or 'U'

Example 1: This example shows the solution $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-1}\boldsymbol{x}$. Matrix ***A*** is a real 9 by 9 upper triangular band matrix with an upper band width of 2 that is not unit triangular, stored in upper-triangular-band-packed storage mode. Vector ***x*** is a vector of length 9, where matrix ***A*** is:

```
┌                                                   ┐
│ 1.0  1.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0 │
│ 0.0  4.0  2.0  3.0  0.0  0.0  0.0  0.0  0.0 │
│ 0.0  0.0  4.0  1.0  1.0  0.0  0.0  0.0  0.0 │
│ 0.0  0.0  0.0  4.0  2.0  2.0  0.0  0.0  0.0 │
│ 0.0  0.0  0.0  0.0  3.0  1.0  1.0  0.0  0.0 │
│ 0.0  0.0  0.0  0.0  0.0  3.0  2.0  2.0  0.0 │
│ 0.0  0.0  0.0  0.0  0.0  0.0  3.0  1.0  0.0 │
│ 0.0  0.0  0.0  0.0  0.0  0.0  0.0  2.0  2.0 │
│ 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0 │
└                                                   ┘
```

Call Statement and Input

```
            UPLO TRANS  DIAG  N   K   A  LDA  X  INCX
             |    |     |    |   |   |   |   |   |
CALL STBSV( 'U' , 'N' , 'N' , 9 , 2 , A , 3 , X , 1  )

        ┌                                         ┐
        │  .    .   1.0  3.0  1.0  2.0  1.0  2.0  0.0 │
A    =  │  .   1.0  2.0  1.0  2.0  1.0  2.0  1.0  2.0 │
        │ 1.0  4.0  4.0  4.0  3.0  3.0  3.0  2.0  1.0 │
        └                                         ┘

X       =  (2.0, 7.0, 1.0, 8.0, 2.0, 8.0, 1.0, 8.0, 3.0)
```

Output

```
X       =  (1.0, 1.0, 0.0, 1.0, 0.0, 2.0, 0.0, 1.0, 3.0)
```

Example 2: This example shows the solution $x \leftarrow A^{-T}x$, solving the same system as in Example 1. Matrix A is a real 9 by 9 lower triangular band matrix with a lower band width of 2 that is not unit triangular, stored in lower-triangular-band-packed storage mode. Vector x is a vector of length 9 where matrix A is:

```
┌                                                   ┐
│ 1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 │
│ 1.0  4.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 │
│ 1.0  2.0  4.0  0.0  0.0  0.0  0.0  0.0  0.0 │
│ 0.0  3.0  1.0  4.0  0.0  0.0  0.0  0.0  0.0 │
│ 0.0  0.0  1.0  2.0  3.0  0.0  0.0  0.0  0.0 │
│ 0.0  0.0  0.0  2.0  1.0  3.0  0.0  0.0  0.0 │
│ 0.0  0.0  0.0  0.0  1.0  2.0  3.0  0.0  0.0 │
│ 0.0  0.0  0.0  0.0  0.0  2.0  1.0  2.0  0.0 │
│ 0.0  0.0  0.0  0.0  0.0  0.0  0.0  2.0  1.0 │
└                                                   ┘
```

Call Statement and Input

```
            UPLO TRANS  DIAG  N   K   A  LDA  X  INCX
             |    |     |    |   |   |   |   |   |
CALL STBSV( 'L' , 'T' , 'N' , 9 , 2 , A , 3 , X , 1  )

        ┌                                         ┐
        │ 1.0  4.0  4.0  4.0  3.0  3.0  3.0  2.0  1.0 │
A    =  │ 1.0  2.0  1.0  2.0  1.0  2.0  1.0  2.0   .  │
        │ 1.0  3.0  1.0  2.0  1.0  2.0  0.0   .    .  │
        └                                         ┘
```

```
X          =(same as input X in Example 1)
```

Output

```
X          =(same as output X in Example 1)
```

Example 3: This example shows the solution $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-T}\boldsymbol{x}$, where $k > n$. Matrix $\boldsymbol{A}$ is a real 4 by 4 upper triangular band matrix with an upper band width of 3, even though k is specified as 5. It is not unit triangular and is stored in upper-triangular-band-packed storage mode. Vector $\boldsymbol{x}$ is a vector of length 4 where matrix $\boldsymbol{A}$ is:

```
┌                        ┐
│ 1.0  2.0  3.0  2.0 │
│ 0.0  2.0  2.0  5.0 │
│ 0.0  0.0  3.0  3.0 │
│ 0.0  0.0  0.0  1.0 │
└                        ┘
```

Call Statement and Input

```
            UPLO TRANS  DIAG  N   K   A  LDA  X  INCX
             |     |     |    |   |   |   |   |   |
CALL STBSV( 'U' , 'T' , 'N' , 4 , 5 , A , 6 , X , 1  )

        ┌                    ┐
        │  .    .    .    .  │
        │  .    .    .    .  │
A    =  │  .    .    .   2.0 │
        │  .    .   3.0  5.0 │
        │  .   2.0  2.0  3.0 │
        │ 1.0  2.0  3.0  1.0 │
        └                    ┘

X        =  (5.0, 18.0, 32.0, 41.0)
```

Output

```
X        =  (5.0, 4.0, 3.0, 2.0)
```

Example 4: This example shows the solution $\boldsymbol{x} \leftarrow \boldsymbol{A}^{-T}\boldsymbol{x}$. Matrix $\boldsymbol{A}$ is a complex 7 by 7 lower triangular band matrix with a lower band width of 3 that is not unit triangular, stored in lower-triangular-band-packed storage mode. Vector $\boldsymbol{x}$ is a vector of length 7. Matrix $\boldsymbol{A}$ is:

```
┌                                                                                ┐
│ (1.0, 0.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) │
│ (1.0, 2.0) (2.0, 1.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) │
│ (1.0, 3.0) (2.0, 2.0) (3.0, 1.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) │
│ (1.0, 4.0) (2.0, 3.0) (3.0, 3.0) (4.0, 1.0) (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) │
│ (0.0, 0.0) (2.0, 4.0) (3.0, 3.0) (4.0, 2.0) (2.0, 1.0) (0.0, 0.0) (0.0, 0.0) │
│ (0.0, 0.0) (0.0, 0.0) (3.0, 3.0) (4.0, 3.0) (5.0, 1.0) (3.0, 1.0) (0.0, 0.0) │
│ (0.0, 0.0) (0.0, 0.0) (0.0, 0.0) (4.0, 4.0) (5.0, 2.0) (6.0, 1.0) (2.0, 1.0) │
└                                                                                ┘
```

Call Statement and Input

```
                UPLO TRANS  DIAG  N   K   A  LDA  X  INCX
                 |    |     |    |   |   |   |   |   |
CALL CTBSV( 'L' , 'T' , 'N' , 7 , 3 , A , 4 , X , 1  )
```

$$A = \begin{bmatrix} (1.0, 0.0) & (2.0, 1.0) & (3.0, 1.0) & (4.0, 1.0) & (2.0, 1.0) & (3.0, 1.0) & (2.0, 1.0) \\ (1.0, 2.0) & (2.0, 2.0) & (3.0, 3.0) & (4.0, 2.0) & (5.0, 1.0) & (6.0, 1.0) & . \\ (1.0, 3.0) & (2.0, 3.0) & (3.0, 3.0) & (4.0, 3.0) & (5.0, 2.0) & . & . \\ (1.0, 4.0) & (2.0, 4.0) & (3.0, 3.0) & (4.0, 4.0) & . & . & . \end{bmatrix}$$

```
X       =  ((2.0, 2.0), (7.0, 1.0), (1.0, 1.0), (8.0, 1.0),
            (2.0, 0.0), (8.0, 1.0), (1.0, 2.0))
```

Output

```
X       =  ((-12.048, -13.136), (6.304, -1.472), (-1.880, 1.040),
            (2.600, -1.800), (-2.160, 1.880), (0.800, -1.400),
            (0.800, 0.600))
```

Sparse Linear Algebraic Equation Subroutines

This section contains the sparse linear algebraic equation subroutine descriptions.

DGSF—General Sparse Matrix Factorization Using Storage by Indices, Rows, or Columns

This subroutine factors sparse matrix ***A*** by Gaussian elimination, using a modified Markowitz count with threshold pivoting. The sparse matrix can be stored by indices, rows, or columns. To solve the system of equations, follow the call to this subroutine with a call to DGSS.

Syntax

Fortran	CALL DGSF (*iopt*, *n*, *nz*, *a*, *ia*, *ja*, *lna*, *iparm*, *rparm*, *oparm*, *aux*, *naux*)
C and C++	dgsf (*iopt*, *n*, *nz*, *a*, *ia*, *ja*, *lna*, *iparm*, *rparm*, *oparm*, *aux*, *naux*);
PL/I	CALL DGSF (*iopt*, *n*, *nz*, *a*, *ia*, *ja*, *lna*, *iparm*, *rparm*, *oparm*, *aux*, *naux*);

On Entry

iopt
indicates the storage technique used for sparse matrix ***A***, where:

If *iopt* = 0, it is stored by indices.

If *iopt* = 1, it is stored by rows.

If *iopt* = 2, it is stored by columns.

Specified as: a fullword integer; *iopt* = 0, 1, or 2.

n
is the order *n* of sparse matrix ***A***. Specified as: a fullword integer; $n \geq 0$.

nz
is the number of elements in sparse matrix ***A***, stored in an array, referred to as A. Specified as: a fullword integer; $nz > 0$.

a
is the sparse matrix ***A***, to be factored, stored in an array, referred to as A. Specified as: an array of length *lna*, containing long-precision real numbers.

ia
is the array, referred to as IA, where:

If *iopt* = 0, it contains the row numbers that correspond to the elements in array A.

If *iopt* = 1, it contains the row pointers.

If *iopt* = 2, it contains the row numbers that correspond to the elements in array A.

Specified as: an array of length *lna*, containing fullword integers; IA(*i*) ≥ 1. See "Sparse Matrix" on page 92 for more information on storage techniques.

ja
is the array, referred to as JA, where:

If *iopt* = 0, it contains the column numbers that correspond to the elements in array A.

If *iopt* = 1, it contains the column numbers that correspond to the elements in array A.

If *iopt* = 2, it contains the column pointers.

Specified as: an array of length *lna*, containing fullword integers; JA(*i*) ≥ 1. See "Sparse Matrix" on page 92 for more information on storage techniques.

lna
is the length of the arrays specified for *a*, *ia*, and *ja*. Specified as: a fullword integer; *lna* > 2*nz*. If you do not specify a sufficient amount, it results in an error. See "Error Conditions" on page 590.

The size of *lna* depends on the structure of the input matrix. The requirement that *lna* > 2*nz* does not guarantee a successful run of the program. If the input matrix is expected to have many fill-ins, *lna* should be set larger. Larger *lna* may result in a performance improvement.

For details on how *lna* relates to storage compressions, see "Performance and Accuracy Considerations" on page 463.

iparm
is an array of parameters, IPARM(*i*), where:

- IPARM(1) determines whether the default values for *iparm* and *rparm* are used by this subroutine.

  If IPARM(1) = 0, the following default values are used:

  IPARM(2) = 10
  IPARM(3) = 1
  IPARM(4) = 0
  RPARM(1) = 10^{-12}
  RPARM(2) = 0.1

  If IPARM(1) = 1, the default values are not used.

- IPARM(2) determines the number of minimal Markowitz counts that are examined to determine a pivot. (See reference [95].)

- IPARM(3) has the following meaning, where:

  If IPARM(3) = 0, this subroutine checks the values in arrays IA and JA.

  If IPARM(3) = 1, this subroutine assumes that the input values are correct in arrays IA and JA.

- IPARM(4) has the following meaning, where:

  If IPARM(4) = 0, this computation is not performed.

  If IPARM(4) = 1, this subroutine computes:

  The absolute value of the smallest pivot element
  The absolute value of the largest element in ***U***.

  These values are stored in OPARM(2) and OPARM(3), respectively.

- IPARM(5) is reserved.

Specified as: an array of (at least) length 5, containing fullword integers, where the *iparm* values must be:
IPARM(1) = 0 or 1
IPARM(2) ≥ 1
IPARM(3) = 0 or 1
IPARM(4) = 0 or 1

rparm
is an array of parameters, RPARM(*i*), where:

- RPARM(1) contains the lower bound of the absolute value of all elements in the

matrix. If a pivot element is less than this number, the matrix is reported as singular. Any computed element whose absolute value is less than this number is set to 0.

- RPARM(2) is the threshold pivot tolerance used to control the choice of pivots.
- RPARM(3) is reserved.
- RPARM(4) is reserved.
- RPARM(5) is reserved.

Specified as: a one-dimensional array of (at least) length 5, containing long-precision real numbers, where the *rparm* values must be:
RPARM(1) ≥ 0.0
0.0 ≤ RPARM(2) ≤ 1.0

For additional information about *rparm*, see "Performance and Accuracy Considerations" on page 463.

oparm
See On Return.

aux
is the storage work area used by this subroutine. Its size is specified by *naux*.
Specified as: an area of storage, containing long-precision real numbers.

naux
is the size of the work area specified by *aux*—that is, the number of elements in *aux*.
Specified as: a fullword integer; *naux* ≥ 10*n*+100.

On Return

a
is the transformed array, referred to as A, containing the factored matrix ***A***, required as input to DGSS. Returned as: a one-dimensional array of length *lna*, containing long-precision real numbers.

ia
is the transformed array, referred to as IA, required as input to DGSS. Returned as: a one-dimensional array of length *lna*, containing fullword integers.

ja
is the transformed array, referred to as JA, required as input to DGSS. Returned as: a one-dimensional array of length *lna*, containing fullword integers.

oparm
is an array of parameters, OPARM(*i*), where:

- OPARM(1) is the amount of fill-ins for the sparse processing portion of the algorithm.
- OPARM(2) contains the absolute value of the smallest pivot element of the matrix. This value is computed and set only if IPARM(4) = 1.
- OPARM(3) contains the absolute value of the largest element encountered in ***U*** after the factorization. This value is computed and set only if IPARM(4) = 1.
- OPARM(4) is reserved.
- OPARM(5) is reserved.

Returned as: a one-dimensional array of length 5, containing long-precision real numbers.

aux
is the storage work area used by this subroutine. It contains the information required as input for DGSS. Specified as: an area of storage, containing long-precision real numbers.

Notes

1. For a description of the three storage techniques used by this subroutine for sparse matrices, see "Sparse Matrix" on page 92.
2. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: The matrix ***A*** is factored by Gaussian elimination, using a modified Markowitz count with threshold pivoting to compute the sparse LU factorization of ***A***:

$$\boldsymbol{LU} = \boldsymbol{PAQ}$$

where:

A is a general sparse matrix of order *n*, stored by indices, columns, or rows in arrays A, IA, and JA.
L is a unit lower triangular matrix.
U is an upper triangular matrix.
P is a permutation matrix.
Q is a permutation matrix.

To solve the system of equations, follow the call to this subroutine with a call to DGSS. If *n* is 0, no computation is performed. See references [10], [47], and [87].

Error Conditions

Computational Errors

1. If this subroutine has to perform storage compressions, an attention message is issued. When this occurs, the performance of this subroutine is affected. The performance can be improved by increasing the value specified for *lna*.
2. The following errors with their corresponding return codes can occur in this subroutine. Where a value of *i* is indicated, it can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for that particular error code in the ESSL error option table; otherwise, the default value causes your program to terminate when the error occurs. For details, see "What Can You Do about ESSL Computational Errors?" on page 48.
   - For error 2117, return code 2 indicates that the pivot element in a column, *i*, is smaller than the value specified in RPARM(1).
   - For error 2118, return code 3 indicates that pivot element in a row, *i*, is smaller than the value specified in RPARM(1).
   - For error 2120, return code 4 indicates that a row, *i*, is found empty on factorization. The matrix is singular.
   - For error 2121, return code 5 indicates that a column is found empty on factorization. The matrix is singular.
   - For error 2119, return code 6 indicates that the storage space indicated by *lna* is insufficient.

- For error 2122, return code 7 indicates that no pivot element was found in the active submatrix.

Input-Argument Errors

1. *iopt* ≠ 0, 1, or 2
2. *n* < 0
3. *nz* ≤ 0
4. *lna* ≤ 2*nz*
5. IPARM(1) ≠ 0 or 1
6. IPARM(2) ≤ 0
7. IPARM(3) ≠ 0 or 1
8. IPARM(4) ≠ 0 or 1
9. RPARM(1) < 0.0
10. RPARM(2) < 0.0 or RPARM(2) > 1.0
11. *iopt* = 1 and *ia*(*i*) ≥ *ia* (*i*+1), *i* = 1, *n*
12. *iopt* = 2 and *ja*(*i*) ≥ *ja*(*i*+1), *i* = 1, *n*
13. *iopt* = 0 or 1 and *ja*(*i*) < 1 or *ja*(*i*) > *n*, *i* = 1, *nz*
14. *iopt* = 0 or 1 and *ia*(*i*) < 1 or *ia*(*i*) > *n*, *i* = 1, *nz*
15. There are duplicate indices in a row or column of the input matrix.
16. The matrix is singular if a row or column of the input matrix is empty.
17. *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example: This example factors 5 by 5 sparse matrix ***A***, which is stored by indices in arrays A, IA, and JA. The three storage techniques are shown in this example, and the output is the same regardless of the storage technique used. The matrix is factored using Gaussian elimination with threshold pivoting. Matrix ***A*** is:

$$\begin{bmatrix} 2.0 & 0.0 & 4.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 0.0 & 0.0 & 3.0 \\ 0.0 & 0.0 & 3.0 & 4.0 & 0.0 \\ 2.0 & 2.0 & 0.0 & 1.0 & 5.0 \\ 0.0 & 0.0 & 1.0 & 1.0 & 0.0 \end{bmatrix}$$

Note: In this example, only nonzero elements are used as input to the matrix.

Call Statement and Input (Storage-By-Indices)

```
          IOPT  N  NZ  A  IA  JA  LNA  IPARM  RPARM  OPARM  AUX  NAUX
           |    |   |  |   |   |   |     |      |      |     |    |
CALL DGSF( 0  , 5, 13, A, IA, JA, 27 , IPARM, RPARM, OPARM, AUX, 150 )

A       =  (2.0, 1.0, 1.0, 3.0, 4.0, 1.0, 5.0, 2.0, 2.0, 1.0, 1.0,
            4.0, 3.0, . , . , . , . , . , . , . , . , . , . , . , . , . ,
            . , . )
IA      =  (1, 2, 2, 3, 3, 4, 4, 4, 4, 5, 5, 1, 2, . , . , . , . ,
            . , . , . , . , . , . , . , . , . , . , . , . )
JA      =  (1, 1, 2, 3, 4, 4, 5, 1, 2, 3, 4, 3, 5, . , . , . , . ,
            . , . , . , . , . , . , . , . , . , . , . , . )
IPARM   =  (1, 3, 1, 1)
RPARM   =  (1.D-12, 0.1D0)
```

Call Statement and Input (Storage-By-Rows)

```
          IOPT  N  NZ  A  IA  JA  LNA  IPARM  RPARM  OPARM  AUX  NAUX
           |    |   | |   |   |   |     |      |      |     |    |
CALL DGSF( 1  , 5, 13, A, IA, JA, 27 , IPARM, RPARM, OPARM, AUX, 150 )

A       =  (2.0, 4.0, 1.0, 1.0, 3.0, 3.0, 4.0, 2.0, 2.0, 1.0, 5.0,
            1.0, 1.0, . , . , . , . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
            . , . )
IA      =  (1, 3, 6, 8, 12, 14, . , . , . , . , . , . , . , . , . , . , . , . ,
            . , . , . , . , . , . )
JA      =  (1, 3, 1, 2, 5, 3, 4, 1, 2, 4, 5, 3, 4, . , . , . , . , . ,
            . , . , . , . , . , . , . , . , . , . , . , . , . , . , . , . )
IPARM   =  (1, 3, 1, 1)
RPARM   =  (1.D-12, 0.1D0)
```

Call Statement and Input (Storage-By-Columns)

```
          IOPT  N  NZ  A  IA  JA  LNA  IPARM  RPARM  OPARM  AUX  NAUX
           |    |   | |   |   |   |     |      |      |     |    |
CALL DGSF( 2  , 5, 13, A, IA, JA, 27 , IPARM, RPARM, OPARM, AUX, 150 )

A       =  (2.0, 1.0, 2.0, 1.0, 2.0, 4.0, 3.0, 1.0, 4.0, 1.0, 1.0,
            3.0, 5.0, . , . , . , . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
            . , . )
IA      =  (1, 2, 4, 2, 4, 1, 3, 5, 3, 4, 5, 2, 4, . , . , . , . , . ,
            . , . , . , . , . , . , . , . , . , . , . , . , . , . , . , . )
JA      =  (1, 4, 6, 9, 12, 14, . , . , . , . , . , . , . , . , . , . , . , . ,
            . , . , . , . , . , . )
IPARM   =  (1, 3, 0, 1)
RPARM   =  (1.D-12, 0.1D0)
```

Output

```
A       =  (0.5, . , 0.3, 1.0, . , 1.0, . , 3.0, . , . , . , 1.0,
            1.0, . , . , . , . , . , . , . , . , -1.7, -0.5, -1.0, -1.0,
            4.0, -3.0, -4.0)
IA      =  (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, . , . , . , . , . , . ,
            . , . , . , . , 2, 1, 1, 3, 3, 5, 5)
JA      =  (1, 0, 5, 2, 0, 4, 0, 2, 0, 0, 0, 3, 4, . , . , . , . , . , . ,
            . , . , . , . , 4, 2, 4, 4, 1, 3, 1)
OPARM   =  (1.000000, 0.333333, 3.000000)
```

DGSS—General Sparse Matrix or Its Transpose Solve Using Storage by Indices, Rows, or Columns

This subroutine solves either of the following systems:

$$\boldsymbol{Ax} = \boldsymbol{b}$$
$$\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x} = \boldsymbol{b}$$

where $\boldsymbol{A}$ is a sparse matrix, $\boldsymbol{A}^{\mathrm{T}}$ is the transpose of sparse matrix $\boldsymbol{A}$, and $\boldsymbol{x}$ and $\boldsymbol{b}$ are vectors. DGSS uses the results of the factorization of matrix $\boldsymbol{A}$, produced by a preceding call to DGSF.

Note: The input to this solve subroutine must be the output from the factorization subroutine, DGSF.

Syntax

Fortran	CALL DGSS (*jopt*, *n*, *a*, *ia*, *ja*, *lna*, *bx*, *aux*, *naux*)
C and C++	dgss (*jopt*, *n*, *a*, *ia*, *ja*, *lna*, *bx*, *aux*, *naux*);
PL/I	CALL DGSS (*jopt*, *n*, *a*, *ia*, *ja*, *lna*, *bx*, *aux*, *naux*);

On Entry

jopt
indicates the type of computation to be performed, where:

If *jopt* = 0, $\boldsymbol{Ax} = \boldsymbol{b}$ is solved, where the right-hand side is not sparse.

If *jopt* = 1, $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x} = \boldsymbol{b}$ is solved, where the right-hand side is not sparse.

If *jopt* = 10, $\boldsymbol{Ax} = \boldsymbol{b}$ is solved, where the right-hand side is sparse.

If *jopt* = 11, $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x} = \boldsymbol{b}$ is solved, where the right-hand side is sparse.

Specified as: a fullword integer; *jopt* = 0, 1, 10, or 11.

n
is the order *n* of sparse matrix $\boldsymbol{A}$. Specified as: a fullword integer; $n \geq 0$.

a
is the factorization of sparse matrix $\boldsymbol{A}$, stored in array A, produced by a preceding call to DGSF. Specified as: an array of length *lna*, containing long-precision real numbers.

ia
is the array, referred to as IA, produced by a preceding call to DGSF. Specified as: an array of length *lna*, containing fullword integers.

ja
is the array, referred to as JA, produced by a preceding call to DGSF. Specified as: an array of length *lna*, containing fullword integers.

lna
is the length of the arrays A, IA, and JA. In DGSS, *lna* must be identical to the value specified in DGSF; otherwise, results are unpredictable. Specified as: a fullword integer; *lna* > 0.

bx
is the vector $\boldsymbol{b}$ of length *n*, containing the right-hand side of the system. Specified as: a one-dimensional array of (at least) length *n*, containing long-precision real numbers.

aux
is the storage work area passed to this subroutine by a preceding call to DGSF. Its size is specified by *naux*. Specified as: an area of storage, containing long-precision real numbers.

naux
is the size of the work area specified by *aux*—that is, the number of elements in *aux*. Specified as: a fullword integer; *naux* ≥ 10*n*+100.

On Return

ia
is the transformed array, referred to as IA, which can be used as input in subsequent calls to this subroutine. This may result in a performance increase. Specified as: an array of length *lna*, containing fullword integers.

bx
is the solution vector $\boldsymbol{x}$ of length *n*, containing the results of the computation. Specified as: a one-dimensional array, containing long-precision real numbers.

Notes

1. The input arguments *n*, *lna*, and *naux*, must be the same as those specified for DGSF. Whereas, the input arguments *a, ia, ja*, and *aux* must be those produced on output by DGSF. Otherwise, results are unpredictable.
2. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: The system $\boldsymbol{Ax} = \boldsymbol{b}$ is solved for $\boldsymbol{x}$, where $\boldsymbol{A}$ is a sparse matrix and $\boldsymbol{x}$ and $\boldsymbol{b}$ are vectors. Depending on the value specified for the *jopt* argument, DGSS can also solve the system $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x} = \boldsymbol{b}$, where $\boldsymbol{A}^{\mathrm{T}}$ is the transpose of sparse matrix $\boldsymbol{A}$.

If the value specified for the *jopt* argument is 0 or 10, the following equation is solved:

$$\boldsymbol{Ax} = \boldsymbol{b}$$

If the value specified for the *jopt* argument is 1 or 11, the following equation is solved:

$$\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x} = \boldsymbol{b}$$

DGSS uses the results of the factorization of matrix $\boldsymbol{A}$, produced by a preceding call to DGSF. The transformed matrix $\boldsymbol{A}$ consists of the upper triangular matrix $\boldsymbol{U}$ and the lower triangular matrix $\boldsymbol{L}$.

See references [10], [47], and [87].

Error Conditions

Computational Errors: None

Input-Argument Errors

1. *jopt* ≠ 0, 1, 10, or 11
2. *n* < 0
3. *lna* ≤ 0
4. *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example 1: This example shows how to solve the system $\boldsymbol{Ax} = \boldsymbol{b}$, where matrix $\boldsymbol{A}$ is a 5 by 5 sparse matrix. The right-hand side is not sparse.

Note: The input for this subroutine is the same as the output from DGSF, except for BX.

Matrix $\boldsymbol{A}$ is:

$$\begin{bmatrix} 2.0 & 0.0 & 4.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 0.0 & 0.0 & 3.0 \\ 0.0 & 0.0 & 3.0 & 4.0 & 0.0 \\ 2.0 & 2.0 & 0.0 & 1.0 & 5.0 \\ 0.0 & 0.0 & 1.0 & 1.0 & 0.0 \end{bmatrix}$$

Call Statement and Input

```
          JOPT  N   A   IA   JA   LNA   BX   AUX   NAUX
           |    |   |    |    |    |     |    |     |
CALL DGSS( 0  , 5 , A , IA , JA , 27  , BX , AUX , 150 )

A       =  (0.5, . , 0.3, 1.0, . , 1.0, . , 3.0, . , . , . , 1.0,
            1.0, . , . , . , . , . , . , . , . , -1.7, -0.5, -1.0, -1.0,
            4.0, -3.0, -4.0)
IA      =  (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, . , . , . , . , . ,
            . , . , . , 2, 1, 1, 3, 3, 5, 5)
JA      =  (1, 0, 5, 2, 0, 4, 0, 2, 0, 0, 0, 3, 4, . , . , . , . , . ,
            . , . , . , 4, 2, 4, 4, 1, 3, 1)
BX      =  (1.0, 1.0, 1.0, 1.0, 1.0)
```

Output

```
IA      =  (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, . , . , . , . , . ,
            . , . , . , 2, 1, 1, 3, 3, 5, 5)
BX      =  (-5.500000, 9.500000, 3.000000, -2.000000, -1.000000)
```

Example 2: This example shows how to solve the system $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x} = \boldsymbol{b}$, using the same matrix $\boldsymbol{A}$ used in Example 1. The input is also the same as in Example 1, except for the *jopt* argument. The right-hand side is not sparse.

Call Statement and Input

```
          JOPT  N   A   IA   JA   LNA   BX   AUX   NAUX
           |    |   |    |    |    |     |    |     |
CALL DGSS( 1  , 5 , A , IA , JA , 27  , BX , AUX , 150 )

BX      =  (1.0, 1.0, 1.0, 1.0, 1.0)
```

Output

```
IA      =  (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, . , . , . , . , . ,
            . , . , . , 2, 1, 1, 3, 3, 5, 5)
BX      =  (0.000000, -3.000000, -2.000000, 2.000000, 7.000000)
```

Example 3: This example shows how to solve the system $\boldsymbol{Ax} = \boldsymbol{b}$, using the same matrix $\boldsymbol{A}$ as in Examples 1 and 2. The input is also the same as in Examples 1 and 2, except for the *jopt* and *bx* arguments. The right-hand side is sparse.

Call Statement and Input

```
          JOPT  N   A   IA   JA   LNA   BX   AUX   NAUX
           |    |   |    |    |    |     |    |     |
CALL DGSS( 10 , 5 , A , IA , JA , 27  , BX , AUX , 150 )

BX       =  (0.0, 0.0, 0.0, 1.0, 0.0)
```

Output

```
IA       =  (1, 4, 2, 5, 3, 5, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
             0, 2, 1, 1, 3, 3, 5, 5)
BX       =  (0.000000, 3.000000, 0.000000, 0.000000, -1.000000)
```

Example 4: This example shows how to solve the system $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x} = \boldsymbol{b}$, using the same matrix $\boldsymbol{A}$ as in Examples 1, 2, and 3. The input is also the same as in Examples 1, 2, and 3, except for the *jopt* argument. The right-hand side is sparse.

Call Statement and Input

```
          JOPT  N   A   IA   JA   LNA   BX   AUX   NAUX
           |    |   |    |    |    |     |    |     |
CALL DGSS( 11 , 5 , A , IA , JA , 27  , BX , AUX , 150 )

BX       =  (0.0, 0.0, 0.0, 1.0, 0.0)
```

Output

```
IA       =  (1, 4, 2, 5, 3, 5, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
             0, 2, 1, 1, 3, 3, 5, 5)
BX       =   (0.000000, 0.000000, 1.000000, 0.000000, -3.000000 )
```

DGKFS—General Sparse Matrix or Its Transpose Factorization, Determinant, and Solve Using Skyline Storage Mode

This subroutine can perform either or both of the following functions for general sparse matrix ***A***, stored in skyline storage mode, and for vectors ***x*** and ***b***:

- Factor ***A*** and, optionally, compute the determinant of ***A***.
- Solve the system $\boldsymbol{Ax} = \boldsymbol{b}$ or $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x} = \boldsymbol{b}$ using the results of the factorization of matrix ***A***, produced on this call or a preceding call to this subroutine.

You also have the choice of using profile-in or diagonal-out skyline storage mode for ***A*** on input or output.

Note: The input to the solve performed by this subroutine must be the output from the factorization performed by this subroutine.

Syntax

Fortran	CALL DGKFS (*n*, *au*, *nu*, *idu*, *al*, *nl*, *idl*, *iparm*, *rparm*, *aux*, *naux*, *bx*, *ldbx*, *mbx*)
C and C++	dgkfs (*n*, *au*, *nu*, *idu*, *al*, *nl*, *idl*, *iparm*, *rparm*, *aux*, *naux*, *bx*, *ldbx*, *mbx*);
PL/I	CALL DGKFS (*n*, *au*, *nu*, *idu*, *al*, *nl*, *idl*, *iparm*, *rparm*, *aux*, *naux*, *bx*, *ldbx*, *mbx*);

On Entry

n
is the order of general sparse matrix ***A***. Specified as: a fullword integer; $n \geq 0$.

au
is the array, referred to as AU, containing one of three forms of the upper triangular part of general sparse matrix ***A***, depending on the type of computation performed, where:

- If you are doing a **factor and solve** or a **factor only**, and if IPARM(3) = 0, then AU contains the unfactored upper triangle of general sparse matrix ***A***.
- If you are doing a **factor only**, and if IPARM(3) > 0, then AU contains the partially factored upper triangle of general sparse matrix ***A***. The first IPARM(3) columns in the upper triangle of ***A*** are already factored. The remaining columns are factored in this computation.
- If you are doing a **solve only**, then AU contains the factored upper triangle of general sparse matrix ***A***, produced by a preceding call to this subroutine.

In each case:

If IPARM(4) = 0, diagonal-out skyline storage mode is used for ***A***.

If IPARM(4) = 1, profile-in skyline storage mode is used for ***A***.

Specified as: a one-dimensional array of (at least) length *nu*, containing long-precision real numbers.

nu
is the length of array AU. Specified as: a fullword integer; $nu \geq 0$ and $nu \geq (IDU(n+1)-1)$.

idu
is the array, referred to as IDU, containing the relative positions of the diagonal elements of matrix ***A*** (in one of its three forms) in array AU. Specified as: a one-dimensional array of (at least) length *n*+1, containing fullword integers.

al
is the array, referred to as AL, containing one of three forms of the lower triangular part of general sparse matrix ***A***, depending on the type of computation performed, where:

- If you are doing a **factor and solve** or a **factor only**, and if IPARM(3) = 0, then AL contains the unfactored lower triangle of general sparse matrix ***A***.
- If you are doing a **factor only**, and if IPARM(3) > 0, then AL contains the partially factored lower triangle of general sparse matrix ***A***. The first IPARM(3) rows in the lower triangle of ***A*** are already factored. The remaining rows are factored in this computation.
- If you are doing a **solve only**, then AL contains the factored lower triangle of general sparse matrix ***A***, produced by a preceding call to this subroutine.

Note: In all these cases, entries in AL for diagonal elements of ***A*** are not assumed to have meaningful values.

In each case:

If IPARM(4) = 0, diagonal-out skyline storage mode is used for ***A***.

If IPARM(4) = 1, profile-in skyline storage mode is used for ***A***.

Specified as: a one-dimensional array of (at least) length *nl*, containing long-precision real numbers.

nl
is the length of array AL. Specified as: a fullword integer; $nl \geq 0$ and $nl \geq (IDL(n+1)-1)$.

idl
is the array, referred to as IDL, containing the relative positions of the diagonal elements of matrix ***A*** (in one of its three forms) in array AL. Specified as: a one-dimensional array of (at least) length *n*+1, containing fullword integers.

iparm
is an array of parameters, IPARM(*i*), where:

- IPARM(1) indicates whether certain default values for *iparm* and *rparm* are used by this subroutine, where:

  If IPARM(1) = 0, the following default values are used. For restrictions, see "Notes" on page 604.

  ```
  IPARM(2) = 0
  IPARM(3) = 0
  IPARM(4) = 0
  IPARM(5) = 0
  IPARM(10) = 0
  IPARM(11) = −1
  IPARM(12) = −1
  IPARM(13) = −1
  IPARM(14) = −1
  IPARM(15) = 0
  RPARM(10) = 10−12
  ```

If IPARM(1) = 1, the default values are not used.

- IPARM(2) indicates the type of computation performed by this subroutine. The following table gives the IPARM(2) values for each variation:

Type of Computation	$Ax = b$	$Ax = b$ and Determinant(A)	$A^Tx = b$	$A^Tx = b$ and Determinant(A)
Factor and Solve	0	10	100	110
Factor Only	1	11	N/A	N/A
Solve Only	2	N/A	102	N/A

- IPARM(3) indicates whether a full or partial factorization is performed on matrix ***A***, where:

  If IPARM(3) = 0, and:

  If you are doing a **factor and solve** or a **factor only**, then a full factorization is performed for matrix ***A*** on rows and columns 1 through *n*. If you are doing a **solve only**, this argument has no effect on the computation, but must be set to 0.

  If IPARM(3) > 0, and you are doing a **factor only**, then a partial factorization is performed on matrix ***A***. Rows 1 through IPARM(3) of columns 1 through IPARM(3) in matrix ***A*** must be in factored form from a preceding call to this subroutine. The factorization is performed on rows IPARM(3)+1 through *n* and columns IPARM(3)+1 through *n*. For an illustration, see "Notes" on page 604.

- IPARM(4) indicates the input storage mode used for matrix ***A***. This determines the arrangement of data in arrays AU, IDU, AL, and IDL on input, where:

  If IPARM(4) = 0, diagonal-out skyline storage mode is used.

  If IPARM(4) = 1, profile-in skyline storage mode is used.

- IPARM(5) indicates the output storage mode used for matrix ***A***. This determines the arrangement of data in arrays AU, IDU, AL, and IDL on output, where:

  If IPARM(5) = 0, diagonal-out skyline storage mode is used.

  If IPARM(5) = 1, profile-in skyline storage mode is used.

- IPARM(6) through IPARM(9) are reserved.

- IPARM(10) has the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, then IPARM(10) indicates whether certain default values for *iparm* and *rparm* are used by this subroutine, where:

  If IPARM(10) = 0, the following default values are used. For restrictions, see "Notes" on page 604.

  IPARM(11) = −1
  IPARM(12) = −1
  IPARM(13) = −1
  IPARM(14) = −1
  IPARM(15) = 0
  RPARM(10) = 10^{-12}

If IPARM(10) = 1, the default values are not used.

If you are doing a **solve only**, this argument is not used.

- IPARM(11) through IPARM(15) have the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, then IPARM(11) through IPARM(15) control the type of processing to apply to pivot elements occurring in regions 1 through 5, respectively. The pivot elements are u_{kk} for $k = 1, n$ when doing a full factorization, and they are $k =$ IPARM(3)+1, n when doing a partial factorization. The region in which a pivot element falls depends on the sign and magnitude of the pivot element. The regions are determined by RPARM(10). For a description of the regions and associated pivot values, see "Notes" on page 604. For each region i for $i = 1,5$, where the pivot occurs in region i, the processing applied to the pivot element is determined by IPARM(10+i), where:

  If IPARM(10+i) = −1, the pivot element is trapped and computational error 2126 is generated. See "Error Conditions" on page 606.
  If IPARM(10+i) = 0, for $i = 1, 2, 4$, and 5, processing continues normally.

  Note: A value of 0 is not permitted for region 3, because if processing continues, a divide-by-zero exception occurs.

  If IPARM(10+i) = 1, the pivot element is replaced with the value in RPARM(10+i), and processing continues normally.

If you are doing a **solve only**, these arguments are not used.

- IPARM(16) through IPARM(25), see On Return.

Specified as: a one-dimensional array of (at least) length 25, containing fullword integers, where:

IPARM(1) = 0 or 1
IPARM(2) = 0, 1, 2, 10, 11, 100, 102, or 110
If IPARM(2) = 0, 2, 10, 100, 102, or 110, then IPARM(3) = 0
If IPARM(2) = 1 or 11, then 0 ≤ IPARM(3) ≤ n
IPARM(4), IPARM(5) = 0 or 1
If IPARM(2) = 0, 1, 10, 11, 100, or 110, then:
IPARM(10) = 0 or 1
IPARM(11), IPARM(12) = −1, 0, or 1
IPARM(13) = −1 or 1
IPARM(14), IPARM(15) = −1, 0, or 1

rparm
is an array of parameters, RPARM(i), where:

- RPARM(1) through RPARM(9) are reserved.
- RPARM(10) has the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, RPARM(10) is the tolerance value for small pivots. This sets the bounds for the pivot regions, where pivots are processed according to the options you specify for the five regions in IPARM(11) through IPARM(15), respectively. The suggested value is $10^{-15} \le$ IPARM(10) ≤ 1.

  If you are doing a **solve only**, this argument is not used.

- RPARM(11) through RPARM(15) have the following meaning, where:

If you are doing a **factor and solve** or a **factor only**, RPARM(11) through RPARM(15) are the fix-up values to use for the pivots in regions 1 through 5, respectively. For each RPARM(10+*i*) for *i* = 1,5, where the pivot occurs in region *i*:

If IPARM(10+*i*) = 1, the pivot is replaced with RPARM(10+*i*), where |RPARM(10+*i*)| should be a sufficiently large nonzero value to avoid overflow when calculating the reciprocal of the pivot. The suggested value is $10^{-15} \le |RPARM(10+i)| \le 1$.
If IPARM(10+*i*) ≠ 1, RPARM(10+*i*) is not used.

If you are doing a **solve only**, these arguments are not used.

- RPARM(16) through RPARM(25), see On Return.

Specified as: a one-dimensional array of (at least) length 25, containing long-precision real numbers, where if IPARM(2) = 0, 1, 10, 11, 100, or 110, then:
RPARM(10) ≥ 0.0
RPARM(11) through RPARM(15) ≠ 0.0

aux

has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is the storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing long-precision real numbers.

naux

is the size of the work area specified by *aux*—that is, the number of elements in *aux*.
Specified as: a fullword integer, where:

If *naux* = 0 and error 2015 is unrecoverable, DGKFS dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise,

If you are doing a **factor only**, use *naux* ≥ 5*n*.

If you are doing a **factor and solve** or a **solve only**, use *naux* ≥ 5*n*+4*mbx*.

bx

has the following meaning, where:

If you are doing a **factor and solve** or a **solve only**, *bx* is the array, containing the *mbx* right-hand side vectors ***b*** of the system $\boldsymbol{Ax} = \boldsymbol{b}$ or $\boldsymbol{A}^T\boldsymbol{x} = \boldsymbol{b}$. Each vector ***b*** is length *n* and is stored in the corresponding column of the array.

If you are doing a **factor only**, this argument is not used in the computation.

Specified as: an *ldbx* by (at least) *mbx* array, containing long-precision real numbers.

ldbx

has the following meaning, where:

If you are doing a **factor and solve** or a **solve only**, *ldbx* is the leading dimension of the array specified for *bx*.

If you are doing a **factor only**, this argument is not used in the computation.

Specified as: a fullword integer; *ldbx* ≥ *n* and:

If *mbx* ≠ 0, then *ldbx* > 0.

If *mbx* = 0, then *ldbx* ≥ 0.

mbx
has the following meaning, where:

If you are doing a **factor and solve** or a **solve only**, *mbx* is the number of right-hand side vectors, ***b***, in the array specified for *bx*.

If you are doing a **factor only**, this argument is not used in the computation.

Specified as: a fullword integer; *mbx* ≥ 0.

On Return

au
is the array, referred to as AU, containing the upper triangular part of the ***LU*** factored form of general sparse matrix ***A***, where:

If IPARM(5) = 0, diagonal-out skyline storage mode is used for ***A***.

If IPARM(5) = 1, profile-in skyline storage mode is used for ***A***.

(If *mbx* = 0 and you are doing a solve only, then *au* is unchanged on output.)
Returned as: a one-dimensional array of (at least) length *nu*, containing long-precision real numbers.

idu
is the array, referred to as IDU, containing the relative positions of the diagonal elements of the factored output matrix ***A*** in array AU. (If *mbx* = 0 and you are doing a solve only, then *idu* is unchanged on output.) Returned as: a one-dimensional array of (at least) length *n*+1, containing fullword integers.

al
is the array, referred to as AL, containing the lower triangular part of the ***LU*** factored form of general sparse matrix ***A***, where:

If IPARM(5) = 0, diagonal-out skyline storage mode is used for ***A***.

If IPARM(5) = 1, profile-in skyline storage mode is used for ***A***.

Note: You should assume that entries in AL for diagonal elements of ***A*** do not have meaningful values.

(If *mbx* = 0 and you are doing a solve only, then *al* is unchanged on output.)
Returned as: a one-dimensional array of (at least) length *nl*, containing long-precision real numbers.

idl
is the array, referred to as IDL, containing the relative positions of the diagonal elements of the factored output matrix ***A*** in array AL. (If *mbx* = 0 and you are doing a solve only, then *idl* is unchanged on output.) Returned as: a one-dimensional array of (at least) length *n*+1, containing fullword integers.

iparm
is an array of parameters, IPARM(*i*), where:

- IPARM(1) through IPARM(15) are unchanged.
- IPARM(16) has the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, and:

  If IPARM(16) = −1, your factorization did not complete successfully, resulting in computational error 2126.

If IPARM(16) > 0, it is the row number k, in which the maximum absolute value of the ratio a_{kk}/u_{kk} occurred, where:

If IPARM(3) = 0, k can be any of the rows, 1 through n, in the full factorization.

If IPARM(3) > 0, k can be any of the rows, IPARM(3)+1 through n, in the partial factorization.

If you are doing a **solve only**, this argument is not used in the computation and is unchanged.

- IPARM(17) through IPARM(20) are reserved.
- IPARM(21) through IPARM(25) have the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, IPARM(21) through IPARM(25) have the following meanings for each region i for i = 1,5, respectively:

  If IPARM(20+i) = −1, your factorization did not complete successfully, resulting in computational error 2126.

  If IPARM(20+i) ≥ 0, it is the number of pivots in region i for the columns that were factored in matrix $\boldsymbol{A}$, where:

  If IPARM(3) = 0, columns 1 through n were factored in the full factorization.

  If IPARM(3) > 0, columns IPARM(3)+1 through n were factored in the partial factorization.

  If you are doing a **solve only**, these arguments are not used in the computation and are unchanged.

Returned as: a one-dimensional array of (at least) length 25, containing fullword integers.

rparm

is an array of parameters, RPARM(i), where:

- RPARM(1) through RPARM(15) are unchanged.
- RPARM(16) has the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, and:

  If RPARM(16) = 0.0, your factorization did not complete successfully, resulting in computational error 2126.

  If |RPARM(16)| > 0.0, it is the ratio for row k, a_{kk}/u_{kk}, having the maximum absolute value. Row k is indicated in IPARM(16), and:

  If IPARM(3) = 0, the ratio corresponds to one of the rows, 1 through n, in the full factorization.

  If IPARM(3) > 0, the ratio corresponds to one of the rows, IPARM(3)+1 through n, in the partial factorization.

  If you are doing a **solve only**, this argument is not used in the computation and is unchanged.

- RPARM(17) and RPARM(18) have the following meaning, where:

If you are **computing the determinant** of matrix ***A***, then RPARM(17) is the mantissa, *detbas*, and RPARM(18) is the power of 10, *detpwr*, used to express the value of the determinant: $detbas(10^{detpwr})$, where $1 \le detbas < 10$. Also:

If IPARM(3) = 0, the determinant is computed for columns 1 through *n* in the full factorization.
If IPARM(3) > 0, the determinant is computed for columns IPARM(3)+1 through *n* in the partial factorization.

If you are **not computing the determinant** of matrix ***A***, these arguments are not used in the computation and are unchanged.

- RPARM(19) through RPARM(25) are reserved.

Returned as: a one-dimensional array of (at least) length 25, containing long-precision real numbers.

bx

has the following meaning, where:

If you are doing a **factor and solve** or a **solve only**, *bx* is the array, containing the *mbx* solution vectors ***x*** of the system $\boldsymbol{Ax} = \boldsymbol{b}$ or $\boldsymbol{A}^T\boldsymbol{x} = \boldsymbol{b}$. Each vector ***x*** is length *n* and is stored in the corresponding column of the array. (If *mbx* = 0, then *bx* is unchanged on output.)

If you are doing a **factor only**, this argument is not used in the computation and is unchanged.

Returned as: an *ldbx* by (at least) *mbx* array, containing long-precision real numbers.

Notes

1. If you set either IPARM(1) = 0 or IPARM(10) = 0, indicating you want to use the default values for IPARM(11) through IPARM(15) and RPARM(10), then:
   - Matrix ***A*** must be positive definite.
   - No pivots are fixed, using RPARM(11) through RPARM(15) values.
   - No small pivots are tolerated; that is, the value should be |*pivot*| > RPARM(10).
2. Many of the input and output parameters for *iparm* and *rparm* are defined for the five pivot regions handled by this subroutine. The limits of the regions are based on RPARM(10), as shown in Figure 11. The pivot values in each region are:

   Region 1: *pivot* < −RPARM(10)
   Region 2: −RPARM(10) ≤ *pivot* < 0
   Region 3: *pivot* = 0
   Region 4: 0 < *pivot* ≤ RPARM(10)
   Region 5: *pivot* > RPARM(10)

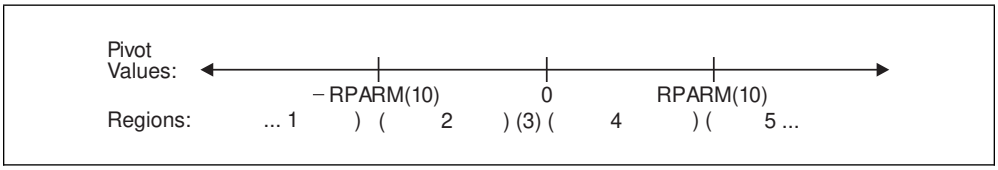


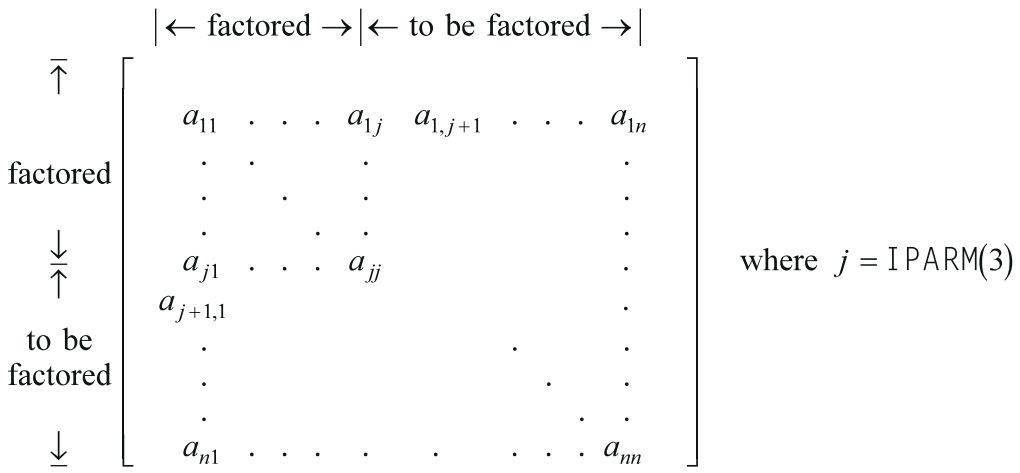
Figure 11. Five Pivot Regions

3. The IPARM(4) and IPARM(5) arguments allow you to specify the same or different skyline storage modes for your input and output arrays for matrix ***A***. This allows you

to change storage modes as needed. However, if you are concerned with performance, you should use diagonal-out skyline storage mode for both input and output, if possible, because there is less overhead.

For a description of how sparse matrices are stored in skyline storage mode, see "Profile-In Skyline Storage Mode" on page 103 and "Diagonal-Out Skyline Storage Mode" on page 100.

4. Following is an illustration of the portion of matrix ***A*** factored in the partial factorization when `IPARM(3) > 0`. In this case, the subroutine assumes that rows and columns 1 through `IPARM(3)` are already factored and that rows and columns `IPARM(3)+1` through *n* are to be factored in this computation.



You use the partial factorization function when, for design or storage reasons, you must factor the matrix ***A*** in stages. When doing a partial factorization, you must use the same skyline storage mode for all parts of the matrix as it is progressively factored.

5. Your various arrays must have no common elements; otherwise, results are unpredictable.
6. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: This subroutine can factor, compute the determinant of, and solve general sparse matrix ***A***, stored in skyline storage mode. For all computations, input matrix ***A*** can be stored in either diagonal-out or profile-in skyline storage mode. Output matrix ***A*** can also be stored in either of these modes and can be different from the mode used for input.

Matrix ***A*** is factored into the following form using specified pivot processing:

$$A = LU$$

where:

U is an upper triangular matrix.
L is a lower triangular matrix.

The transformed matrix ***A***, factored into its ***LU*** form, is stored in packed format in arrays AU and AL. The inverse of the diagonal of matrix ***U*** is stored in the corresponding elements of array AU. The off-diagonal elements of the upper triangular matrix ***U*** are stored in the corresponding off-diagonal elements of array AU. The off-diagonal elements of the lower triangular matrix ***L*** are stored in the corresponding off-diagonal elements of array AL. (The diagonal elements stored in array AL do not have meaningful values.)

The partial factorization of matrix ***A***, which you can do when you specify the factor-only option, assumes that the first IPARM(3) rows and columns are already factored in the input matrix. It factors the remaining n–IPARM(3) rows and columns in matrix ***A***. (See "Notes" on page 604 for an illustration.) It updates only the elements in arrays AU and AL corresponding to the part of matrix ***A*** that is factored.

The determinant can be computed with any of the factorization computations. With a full factorization, you get the determinant for the whole matrix. With a partial factorization, you get the determinant for only that part of the matrix factored in this computation.

The system $\boldsymbol{Ax} = \boldsymbol{b}$ or $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x} = \boldsymbol{b}$, having multiple right-hand sides, is solved for $\boldsymbol{x}$, using the transformed matrix ***A*** produced by this call or a subsequent call to this subroutine.

See references [9], [12], [25], [47], and [65]. If n is 0, no computation is performed. If *mbx* is 0, no solve is performed.

Error Conditions

Resource Errors

- Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.
- Unable to allocate internal work area.

Computational Errors

1. If a pivot occurs in region i for i = 1,5 and IPARM(10+i) = 1, the pivot value is replaced with RPARM(10+i), an attention message is issued, and processing continues.
2. Unacceptable pivot values occurred in the factorization of matrix ***A***.
   - One or more diagonal elements of ***U*** contains unacceptable pivots and no valid fixup is applicable. The row number i of the first unacceptable pivot element is identified in the computational error message.
   - The return code is set to 2.
   - i can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2126 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see "What Can You Do about ESSL Computational Errors?" on page 48.

Input-Argument Errors

1. $n < 0$
2. $nu < 0$
3. IDU(n+1) > nu+1
4. IDU(i+1) ≤ IDU(i) for i = 1, n
5. IDU(i+1) > IDU(i)+i and IPARM(4) = 0 for i = 1, n
6. IDU(i) > IDU(i–1)+i and IPARM(4) = 1 for i = 2, n
7. $nl < 0$
8. IDL(n+1) > nl+1

9. IDL(i+1) ≤ IDL(i) for i = 1, n
10. IDL(i+1) > IDL(i)+i and IPARM(4) = 0 for i = 1, n
11. IDL(i) > IDL(i−1)+i and IPARM(4) = 1 for i = 2, n
12. IPARM(1) ≠ 0 or 1
13. IPARM(2) ≠ 0, 1, 2, 10, 11, 100, 102, or 110
14. IPARM(3) < 0
15. IPARM(3) > n
16. IPARM(3) > 0 and IPARM(2) ≠ 1 or 11
17. IPARM(4), IPARM(5) ≠ 0 or 1
18. IPARM(2) = 0, 1, 10, 11, 100, or 110 and:
    IPARM(10) ≠ 0 or 1
    IPARM(11), IPARM(12) ≠ −1, 0, or 1
    IPARM(13) ≠ −1 or 1
    IPARM(14), IPARM(15) ≠ −1, 0, or 1
    RPARM(10) < 0.0
    RPARM(10+i) = 0.0 and IPARM(10+i) = 1 for i = 1,5
19. IPARM(2) = 0, 2, 10, 100, 102, or 110 and:
    $ldbx$ ≤ 0 and mbx ≠ 0 and n ≠ 0
    $ldbx$ < 0 and mbx = 0
    $ldbx$ < n and mbx ≠ 0
    mbx < 0
20. Error 2015 is recoverable or $naux$≠0, and $naux$ is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example 1: This example shows how to factor a 9 by 9 general sparse matrix $\boldsymbol{A}$ and solve the system $\boldsymbol{Ax} = \boldsymbol{b}$ with three right-hand sides. The default values are used for IPARM and RPARM. Input matrix $\boldsymbol{A}$, shown here, is stored in diagonal-out skyline storage mode. Matrix $\boldsymbol{A}$ is:

```
┌                                                   ┐
│ 2.0  2.0  2.0  0.0  0.0  0.0   0.0  0.0   0.0 │
│ 2.0  4.0  4.0  2.0  2.0  0.0   0.0  0.0   2.0 │
│ 2.0  4.0  6.0  4.0  4.0  0.0   2.0  0.0   4.0 │
│ 2.0  4.0  6.0  6.0  6.0  2.0   4.0  0.0   6.0 │
│ 0.0  0.0  0.0  2.0  4.0  4.0   4.0  2.0   4.0 │
│ 0.0  2.0  4.0  6.0  8.0  6.0   8.0  4.0  10.0 │
│ 0.0  0.0  0.0  2.0  4.0  6.0   8.0  6.0   8.0 │
│ 0.0  0.0  0.0  2.0  4.0  6.0   8.0  8.0  10.0 │
│ 2.0  4.0  6.0  6.0  8.0  6.0  10.0  8.0  16.0 │
└                                                   ┘
```

Output matrix $\boldsymbol{A}$, shown here, is in $\boldsymbol{LU}$ factored form with $\boldsymbol{U}^{-1}$ on the diagonal, and is stored in diagonal-out skyline storage mode. Matrix $\boldsymbol{B}$ is:

```
┌                                                   ┐
│ 0.5  2.0  2.0  0.0  0.0  0.0  0.0  0.0  0.0 │
│ 1.0  0.5  2.0  2.0  2.0  0.0  0.0  0.0  2.0 │
│ 1.0  1.0  0.5  2.0  2.0  0.0  2.0  0.0  2.0 │
│ 1.0  1.0  1.0  0.5  2.0  2.0  2.0  0.0  2.0 │
│ 0.0  0.0  0.0  1.0  0.5  2.0  2.0  2.0  2.0 │
│ 0.0  1.0  1.0  1.0  1.0  0.5  2.0  2.0  2.0 │
│ 0.0  0.0  0.0  1.0  1.0  1.0  0.5  2.0  2.0 │
│ 0.0  0.0  0.0  1.0  1.0  1.0  1.0  0.5  2.0 │
│ 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  0.5 │
└                                                   ┘
```

Call Statement and Input

```
            N   AU  NU  IDU  AL  NL  IDL  IPARM  RPARM  AUX  NAUX  BX  LDBX MBX
            |   |   |    |   |   |    |     |      |     |    |    |    |    |
CALL DGKFS( 9 , AU, 33, IDU, AL, 35, IDL, IPARM, RPARM, AUX,  57 , BX , 12 , 3 )
```

```
AU      =  (2.0, 4.0, 2.0, 6.0, 4.0, 2.0, 6.0, 4.0, 2.0, 4.0, 6.0,
            4.0, 2.0, 6.0, 4.0, 2.0, 8.0, 8.0, 4.0, 4.0, 2.0, 8.0,
            6.0, 4.0, 2.0, 16.0, 10.0, 8.0, 10.0, 4.0, 6.0, 4.0, 2.0)
IDU     =  (1, 2, 4, 7, 10, 14, 17, 22, 26, 34)
AL      =  (0.0, 0.0, 2.0, 0.0, 4.0, 2.0, 0.0, 6.0, 4.0, 2.0, 0.0,
            2.0, 0.0, 8.0, 6.0, 4.0, 2.0, 0.0, 6.0, 4.0, 2.0, 0.0,
            8.0, 6.0, 4.0, 2.0, 0.0, 8.0, 10.0, 6.0, 8.0, 6.0, 6.0,
            4.0, 2.0)
IDL     =  (1, 2, 4, 7, 11, 13, 18, 22, 27, 36)
IPARM   =  (0, . , . , . , . , . , . , . , . , . , . , . , . , . ,
            . , . , . , . , . , . , . , . , . , . , . )

RPARM   =(not relevant)
```

```
        ⎡                        ⎤
        ⎢  6.00   12.00   18.00  ⎥
        ⎢ 16.00   32.00   48.00  ⎥
        ⎢ 26.00   52.00   78.00  ⎥
        ⎢ 36.00   72.00  108.00  ⎥
        ⎢ 20.00   40.00   60.00  ⎥
BX   =  ⎢ 48.00   96.00  144.00  ⎥
        ⎢ 34.00   68.00  102.00  ⎥
        ⎢ 38.00   76.00  114.00  ⎥
        ⎢ 66.00  132.00  198.00  ⎥
        ⎢   .       .       .    ⎥
        ⎢   .       .       .    ⎥
        ⎣   .       .       .    ⎦
```

Output

```
AU      =  (0.5, 0.5, 2.0, 0.5, 2.0, 2.0, 0.5, 2.0, 2.0, 0.5, 2.0,
            2.0, 2.0, 0.5, 2.0, 2.0, 0.5, 2.0, 2.0, 2.0, 2.0, 0.5,
            2.0, 2.0, 2.0, 0.5, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0)
IDU     =(same as input)
AL      =  (0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0,
            1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0,
            1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
            1.0, 1.0)
IDL     =(same as input)
IPARM   =  (0, . , . , . , . , . , . , . , . , . , . , . , . , . ,
            . , 9, . , . , . , . , 0, 0, 0, 0, 9)
RPARM   =  ( . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
            . , 8.0, . , . , . , . , . , . , . , . , . )
```

```
                 ⎡                    ⎤
                 ⎢ 1.00  2.00  3.00 ⎥
                 ⎢ 1.00  2.00  3.00 ⎥
                 ⎢ 1.00  2.00  3.00 ⎥
                 ⎢ 1.00  2.00  3.00 ⎥
                 ⎢ 1.00  2.00  3.00 ⎥
BX    =          ⎢ 1.00  2.00  3.00 ⎥
                 ⎢ 1.00  2.00  3.00 ⎥
                 ⎢ 1.00  2.00  3.00 ⎥
                 ⎢ 1.00  2.00  3.00 ⎥
                 ⎢  .     .     .   ⎥
                 ⎢  .     .     .   ⎥
                 ⎢  .     .     .   ⎥
                 ⎣                    ⎦
```

Example 2: This example shows how to factor the 9 by 9 general sparse matrix $\boldsymbol{A}$ from Example 1, solve the system $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x} = \boldsymbol{b}$ with three right-hand sides, and compute the determinant of $\boldsymbol{A}$. The default values for pivot processing are used for IPARM. Input matrix $\boldsymbol{A}$ is stored in profile-in skyline storage mode. Output matrix $\boldsymbol{A}$ is in $\boldsymbol{LU}$ factored form with $\boldsymbol{U}^{-1}$ on the diagonal, and is stored in diagonal-out skyline storage mode. It is the same as output matrix $\boldsymbol{A}$ in Example 1.

Call Statement and Input

```
            N  AU  NU  IDU  AL  NL  IDL  IPARM  RPARM  AUX  NAUX  BX  LDBX MBX
            |  |   |    |   |   |    |     |       |     |   |    |    &darrow    |
CALL DGKFS( 9, AU, 33, IDU, AL, 35, IDL, IPARM, RPARM, AUX,  57 , BX , 12 , 3 )

                 AU       =  (2.0, 2.0, 4.0, 2.0, 4.0, 6.0, 2.0, 4.0, 6.0, 2.0, 4.0,
                              6.0, 4.0, 2.0, 4.0, 6.0, 2.0, 4.0, 4.0, 8.0, 8.0, 2.0,
                              4.0, 6.0, 8.0, 2.0, 4.0, 6.0, 4.0, 10.0, 8.0, 10.0, 16.0)
                 IDU      =  (1, 3, 6, 9, 13, 16, 21, 25, 33, 34)
                 AL       =  (0.0, 2.0, 0.0, 2.0, 4.0, 0.0, 2.0, 4.0, 6.0, 0.0, 2.0,
                              0.0, 2.0, 4.0, 6.0, 8.0, 0.0, 2.0, 4.0, 6.0, 0.0, 2.0,
                              4.0, 6.0, 8.0, 0.0, 2.0, 4.0, 6.0, 6.0, 8.0, 6.0, 10.0,
                              8.0, 0.0)
                 IDL      =  (1, 3, 6, 10, 12, 17, 21, 26, 35, 36)
                 IPARM    =  (1, 110, 0, 1, 0, . , . , . , . , 0, . , . , . , . , . ,
                              . , . , . , . , . , . , . , . , . , . )

                 RPARM    =(not relevant)

                 ⎡                      ⎤
                 ⎢ 10.00   20.00   30.00 ⎥
                 ⎢ 20.00   40.00   60.00 ⎥
                 ⎢ 28.00   56.00   84.00 ⎥
                 ⎢ 30.00   60.00   90.00 ⎥
                 ⎢ 40.00   80.00  120.00 ⎥
BX    =          ⎢ 30.00   60.00   90.00 ⎥
                 ⎢ 44.00   88.00  132.00 ⎥
                 ⎢ 28.00   56.00   84.00 ⎥
                 ⎢ 60.00  120.00  180.00 ⎥
                 ⎢   .       .       .   ⎥
                 ⎢   .       .       .   ⎥
                 ⎢   .       .       .   ⎥
                 ⎣                      ⎦
```

Output

```
AU      =(same as output AU in Example 1)
IDU     =(same as output IDU in Example 1)
AL      =(same as output AL in Example 1)
IDL     =(same as output IDL in Example 1)
IPARM   =  (1, 110, 0, 1, 0, . , . , . , . , . , 0, . , . , . , . , . , . ,
            9, . , . , . , . , . , 0, 0, 0, 0, 9)
RPARM   =  ( . , . , . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
             . , 8.0, 5.12, 2.0, . , . , . , . , . , . , . , . )
BX      =(same as output BX in Example 1)
```

Example 3: This example shows how to factor a 9 by 9 negative-definite general sparse matrix ***A***, solve the system ***Ax*** = ***b*** with three right-hand sides, and compute the determinant of ***A***. (Default values for pivot processing are not used for IPARM because ***A*** is negative-definite.) Input matrix ***A***, shown here, is stored in diagonal-out skyline storage mode:

```
⎡                                                          ⎤
| -2.0  -2.0  -2.0   0.0   0.0   0.0    0.0   0.0    0.0 |
| -2.0  -4.0  -4.0  -2.0  -2.0   0.0    0.0   0.0   -2.0 |
| -2.0  -4.0  -6.0  -4.0  -4.0   0.0   -2.0   0.0   -4.0 |
| -2.0  -4.0  -6.0  -6.0  -6.0  -2.0   -4.0   0.0   -6.0 |
|  0.0   0.0   0.0  -2.0  -4.0  -4.0   -4.0  -2.0   -4.0 |
|  0.0  -2.0  -4.0  -6.0  -8.0  -6.0   -8.0  -4.0  -10.0 |
|  0.0   0.0   0.0  -2.0  -4.0  -6.0   -8.0  -6.0   -8.0 |
|  0.0   0.0   0.0  -2.0  -4.0  -6.0   -8.0  -8.0  -10.0 |
| -2.0  -4.0  -6.0  -6.0  -8.0  -6.0  -10.0  -8.0  -16.0 |
⎣                                                          ⎦
```

Output matrix ***A***, shown here, is in ***LU*** factored form with $\boldsymbol{U}^{-1}$ on the diagonal, and is stored in diagonal-out skyline storage mode. Matrix ***A*** is:

```
⎡                                                          ⎤
| -0.5  -2.0  -2.0   0.0   0.0   0.0   0.0   0.0   0.0 |
|  1.0  -0.5  -2.0  -2.0  -2.0   0.0   0.0   0.0  -2.0 |
|  1.0   1.0  -0.5  -2.0  -2.0   0.0  -2.0   0.0  -2.0 |
|  1.0   1.0   1.0  -0.5  -2.0  -2.0  -2.0   0.0  -2.0 |
|  0.0   0.0   0.0   1.0  -0.5  -2.0  -2.0  -2.0  -2.0 |
|  0.0   1.0   1.0   1.0   1.0  -0.5  -2.0  -2.0  -2.0 |
|  0.0   0.0   0.0   1.0   1.0   1.0  -0.5  -2.0  -2.0 |
|  0.0   0.0   0.0   1.0   1.0   1.0   1.0  -0.5  -2.0 |
|  1.0   1.0   1.0   1.0   1.0   1.0   1.0   1.0  -0.5 |
⎣                                                          ⎦
```

Call Statement and Input

```
            N  AU  NU  IDU  AL  NL  IDL  IPARM  RPARM  AUX  NAUX  BX  LDBX MBX
            |  |   |   |    |   |   |    |      |      |    |     |   |    |
CALL DGKFS( 9, AU, 33, IDU, AL, 35, IDL, IPARM, RPARM, AUX,  57 , BX , 12 , 3 )
```

```
AU      =  (-2.0, -4.0, -2.0, -6.0, -4.0, -2.0, -6.0, -4.0, -2.0,
            -4.0, -6.0, -4.0, -2.0, -6.0, -4.0, -2.0, -8.0, -8.0,
            -4.0, -4.0, -2.0, -8.0, -6.0, -4.0, -2.0, -16.0, -10.0,
            -8.0, -10.0, -4.0, -6.0, -4.0, -2.0)
IDU     =  (1, 2, 4, 7, 10, 14, 17, 22, 26, 34)
AL      =  (0.0, 0.0, -2.0, 0.0, -4.0, -2.0, 0.0, -6.0, -4.0, -2.0,
            0.0, -2.0, 0.0, -8.0, -6.0, -4.0, -2.0, 0.0, -6.0, -4.0,
            -2.0, 0.0, -8.0, -6.0, -4.0, -2.0, 0.0, -8.0, -10.0,
            -6.0, -8.0, -6.0, -6.0, -4.0, -2.0)
IDL     =  (1, 2, 4, 7, 11, 13, 18, 22, 27, 36)
IPARM   =  (1, 10, 0, 0, 0, . , . , . , . , . , 1, 0, -1, -1, -1, -1, . ,
            . , . , . , . , . , . , . , . , . , . , . )

RPARM   =  ( . , . , . , . , . , . , . , . , . , . , . , . , . , 10-15, . , . ,
             . , . , . , . , . , . , . , . , . , . , . , . , . , . , . , . , . )
BX      =(same as input BX in Example 1)
```

Output

```
AU      =  (-0.5, -0.5, -2.0, -0.5, -2.0, -2.0, -0.5, -2.0, -2.0,
            -0.5, -2.0, -2.0, -2.0, -0.5, -2.0, -2.0, -0.5, -2.0,
            -2.0, -2.0, -2.0, -0.5, -2.0, -2.0, -2.0, -0.5, -2.0,
            -2.0, -2.0, -2.0, -2.0, -2.0, -2.0)
IDU     =(same as input)
AL      =  (0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0,
            1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0,
            1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
            1.0, 1.0)
IDL     =(same as input)
IPARM   =  (1, 10, 0, 0, 0, . , . , . , . , . , 1, 0, -1, -1, -1, -1, 9,
            . , . , . , . , . , 9, 0, 0, 0, 0)

RPARM   =  ( . , . , . , . , . , . , . , . , . , . , . , . , . , 10-15, . , . ,
             . , . , . , 8.0, -5.12, 2.0, . , . , . , . , . , . , . , . , . )
```

```
        ⎡                    ⎤
        ⎢ -1.00  -2.00  -3.00 ⎥
        ⎢ -1.00  -2.00  -3.00 ⎥
        ⎢ -1.00  -2.00  -3.00 ⎥
        ⎢ -1.00  -2.00  -3.00 ⎥
        ⎢ -1.00  -2.00  -3.00 ⎥
BX   =  ⎢ -1.00  -2.00  -3.00 ⎥
        ⎢ -1.00  -2.00  -3.00 ⎥
        ⎢ -1.00  -2.00  -3.00 ⎥
        ⎢ -1.00  -2.00  -3.00 ⎥
        ⎢   .      .      .   ⎥
        ⎢   .      .      .   ⎥
        ⎢   .      .      .   ⎥
        ⎣                    ⎦
```

Example 4: This example shows how to factor the first six rows and columns, referred to as matrix ***A1***, of the 9 by 9 general sparse matrix ***A*** from Example 1 and compute the determinant of ***A1***. Input matrix ***A1***, shown here, is stored in diagonal-out skyline storage mode. Input matrix ***A1*** is:

$$\begin{bmatrix} 2.0 & 2.0 & 2.0 & 0.0 & 0.0 & 0.0 \\ 2.0 & 4.0 & 4.0 & 2.0 & 2.0 & 0.0 \\ 2.0 & 4.0 & 6.0 & 4.0 & 4.0 & 0.0 \\ 2.0 & 4.0 & 6.0 & 6.0 & 6.0 & 2.0 \\ 0.0 & 0.0 & 0.0 & 2.0 & 4.0 & 4.0 \\ 0.0 & 2.0 & 4.0 & 6.0 & 8.0 & 6.0 \end{bmatrix}$$

Output matrix ***A1***, shown here, is in ***LU*** factored form with $\boldsymbol{U}^{-1}$ on the diagonal, and is stored in diagonal-out skyline storage mode. Output matrix ***A1*** is:

$$\begin{bmatrix} 0.5 & 2.0 & 2.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 0.5 & 2.0 & 2.0 & 2.0 & 0.0 \\ 1.0 & 1.0 & 0.5 & 2.0 & 2.0 & 0.0 \\ 1.0 & 1.0 & 1.0 & 0.5 & 2.0 & 2.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.5 & 2.0 \\ 0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.5 \end{bmatrix}$$

Call Statement and Input

```
            N  AU  NU  IDU  AL  NL  IDL  IPARM  RPARM  AUX  NAUX  BX   LDBX   MBX
            |  |   |   |    |   |   |    |      |      |    |     |    |      |
CALL DGKFS( 6, AU, 33, IDU, AL, 35, IDL, IPARM, RPARM, AUX,  45 , BX , LDBX , MBX )
```

```
AU       =(same as input AU in Example 1)
IDU      =  (1, 2, 4, 7, 10, 14, 17)
AL       =(same as input AL in Example 1)
IDL      =  (1, 2, 4, 7, 11, 13, 18)
IPARM    =  (1, 11, 0, 0, 0, . , . , . , . , . , 0, . , . , . , . , . , . ,
             . , . , . , . , . , . , . , . , . , . , . , . )
RPARM    =(not relevant)
BX       =(not relevant)
LDBX     =(not relevant)
MBX      =(not relevant)
```

Output

```
AU       =  (0.5, 0.5, 2.0, 0.5, 2.0, 2.0, 0.5, 2.0, 2.0, 0.5, 2.0,
             2.0, 2.0, 0.5, 2.0, 2.0, 8.0, 8.0, 4.0, 4.0, 2.0, 8.0,
             6.0, 4.0, 2.0, 16.0, 10.0, 8.0, 10.0, 4.0, 6.0, 4.0, 2.0)
IDU      =(same as input)
AL       =  (0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0,
             1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 6.0, 4.0, 2.0, 0.0,
             8.0, 6.0, 4.0, 2.0, 0.0, 8.0, 10.0, 6.0, 8.0, 6.0, 6.0,
             4.0, 2.0)
IDL      =(same as input)
IPARM    =  (1, 11, 0, 0, 0, . , . , . , . , . , 0, . , . , . , . , . , . , 3,
             . , . , . , . , . , 0, 0, 0, 0, 6)
RPARM    =  ( . , . , . , . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
             . , 3.0, 6.4, 1.0, . , . , . , . , . , . , . , . )
BX       =(same as input)
LDBX     =(same as input)
MBX      =(same as input)
```

Example 5: This example shows how to do a partial factorization of the 9 by 9 general sparse matrix ***A*** from Example 1, where the first six rows and columns were factored in Example 4. It factors the remaining three rows and columns and computes the determinant of that part of the matrix. The input matrix, referred to as ***A2***, shown here, is made up of the output factored matrix ***A1*** plus the three remaining unfactored rows and columns of matrix ***A***. Matrix ***A2*** is:

```
⎡                                                   ⎤
| 0.5  2.0  2.0  0.0  0.0  0.0   0.0  0.0   0.0 |
| 1.0  0.5  2.0  2.0  2.0  0.0   0.0  0.0   2.0 |
| 1.0  1.0  0.5  2.0  2.0  0.0   2.0  0.0   4.0 |
| 1.0  1.0  1.0  0.5  2.0  2.0   4.0  0.0   6.0 |
| 0.0  0.0  0.0  1.0  0.5  2.0   4.0  2.0   4.0 |
| 0.0  1.0  1.0  1.0  1.0  0.5   8.0  4.0  10.0 |
| 0.0  0.0  0.0  2.0  4.0  6.0   8.0  6.0   8.0 |
| 0.0  0.0  0.0  2.0  4.0  6.0   8.0  8.0  10.0 |
| 2.0  4.0  6.0  6.0  8.0  6.0  10.0  8.0  16.0 |
⎣                                                   ⎦
```

Both parts of input matrix ***A2*** are stored in diagonal-out skyline storage mode.

Output matrix ***A2*** is the same as output matrix ***A*** in Example 1 and is stored in diagonal-out skyline storage mode.

Call Statement and Input

```
            N  AU  NU  IDU  AL  NL  IDL  IPARM  RPARM  AUX  NAUX  BX   LDBX   MBX
            |  |   |   |    |   |   |    |      |      |    |     |    |      |
CALL DGKFS( 9, AU, 33, IDU, AL, 35, IDL, IPARM, RPARM, AUX,  45 , BX , LDBX , MBX )
```

```
AU       =(same as output AU in Example 4)
IDU      =(same as input IDU in Example 1)
AL       =(same as output AL in Example 4)
IDL      =(same as input IDL in Example 1)
IPARM    =  (1, 11, 6, 0, 0, . , . , . , . , . , 0, . , . , . , . , . , . ,
             . , . , . , . , . , . , . , . , . , . , . , . , . , . , . , . )
RPARM    =(not relevant)
BX       =(not relevant)
LDBX     =(not relevant)
MBX      =(not relevant)
```

Output

```
AU       =(same as output AU in Example 1)
IDU      =(same as output IDU in Example 1)
AL       =(same as output AL in Example 1)
IDL      =(same as output IDL in Example 1)
IPARM    =  (1, 11, 6, 0, 0, . , . , . , . , . , 0, . , . , . , . , . , . , 9,
             . , . , . , . , 0, 0, 0, 0, 3)
RPARM    =  ( . , . , . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
             . , 8.0, 8.0, 0.0, . , . , . , . , . , . , . , . , . , . , . )
BX       =(same as input)
LDBX     =(same as input)
MBX      =(same as input)
```

Example 6: This example shows how to solve the system $Ax = b$ with one right-hand side for a general sparse matrix A. Input matrix A, used here, is the same as factored output matrix A from Example 1, stored in profile-in skyline storage mode. Here, output matrix A is unchanged on output and is stored in profile-in skyline storage mode.

Call Statement and Input

```
            N  AU  NU  IDU  AL  NL  IDL  IPARM  RPARM  AUX NAUX  BX  LDBX MBX
            |  |   |   |    |   |   |    |      |      |   |    |   |    |
CALL DGKFS( 9, AU, 33, IDU, AL, 35, IDL, IPARM, RPARM, AUX, 49 , BX , 9 ,  1 )
```

```
AU       =  (0.5, 2.0, 0.5, 2.0, 2.0, 0.5, 2.0, 2.0, 0.5, 2.0, 2.0,
             2.0, 0.5, 2.0, 2.0, 0.5, 2.0, 2.0, 2.0, 2.0, 0.5, 2.0,
             2.0, 2.0, 0.5, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 0.5)
IDU      =  (1, 3, 6, 9, 13, 16, 21, 25, 33, 34)
AL       =  (0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0,
             0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0,
             1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
             1.0, 0.0)
IDL      =  (1, 3, 6, 10, 12, 17, 21, 26, 35, 36)
IPARM    =  (1, 2, 0, 1, 1, . , . , . , . , . , . , . , . , . , . , . ,
             . , . , . , . , . , . , . , . , . , . , . , . , . )
RPARM    =(not relevant)
BX       =  (12.0, 58.0, 114.0, 176.0, 132.0, 294.0, 240.0, 274.0,
             406.0)
```

Output

```
AU       =(same as input)
IDU      =(same as input)
AL       =(same as input)
IDL      =(same as input)
IPARM    =(same as input)
RPARM    =(not relevant)
BX       =  (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0)
```

DSKFS—Symmetric Sparse Matrix Factorization, Determinant, and Solve Using Skyline Storage Mode

This subroutine can perform either or both of the following functions for symmetric sparse matrix ***A***, stored in skyline storage mode, and for vectors ***x*** and ***b***:

- Factor ***A*** and, optionally, compute the determinant of ***A***.
- Solve the system ***Ax = b*** using the results of the factorization of matrix ***A***, produced on this call or a preceding call to this subroutine.

You have the choice of using either Gaussian elimination or Cholesky decomposition. You also have the choice of using profile-in or diagonal-out skyline storage mode for ***A*** on input or output.

Note: The input to the solve performed by this subroutine must be the output from the factorization performed by this subroutine.

Syntax

Fortran	CALL DSKFS (*n*, *a*, *na*, *idiag*, *iparm*, *rparm*, *aux*, *naux*, *bx*, *ldbx*, *mbx*)
C and C++	dskfs (*n*, *a*, *na*, *idiag*, *iparm*, *rparm*, *aux*, *naux*, *bx*, *ldbx*, *mbx*);
PL/I	CALL DSKFS (*n*, *a*, *na*, *idiag*, *iparm*, *rparm*, *aux*, *naux*, *bx*, *ldbx*, *mbx*);

On Entry

n

is the order of symmetric sparse matrix ***A***. Specified as: a fullword integer; $n \geq 0$.

a

is the array, referred to as A, containing one of three forms of the upper triangular part of symmetric sparse matrix ***A***, depending on the type of computation performed, where:

- If you are doing a **factor and solve** or a **factor only**, and if IPARM(3) = 0, then A contains the unfactored upper triangle of symmetric sparse matrix ***A***.
- If you are doing a **factor only**, and if IPARM(3) > 0, then A contains the partially factored upper triangle of symmetric sparse matrix ***A***. The first IPARM(3) columns in the upper triangle of ***A*** are already factored. The remaining columns are factored in this computation.
- If you are doing a **solve only**, then A contains the factored upper triangle of sparse matrix ***A***, produced by a preceding call to this subroutine.

In each case:

If IPARM(4) = 0, diagonal-out skyline storage mode is used for ***A***.

If IPARM(4) = 1, profile-in skyline storage mode is used for ***A***.

Specified as: a one-dimensional array of (at least) length *na*, containing long-precision real numbers.

na

is the length of array A. Specified as: a fullword integer; $na \geq 0$ and $na \geq (IDIAG(n+1)-1)$.

idiag
is the array, referred to as IDIAG, containing the relative positions of the diagonal elements of matrix ***A*** (in one of its three forms) in array A. Specified as: a one-dimensional array of (at least) length *n*+1, containing fullword integers.

iparm
is an array of parameters, IPARM(*i*), where:

- IPARM(1) indicates whether certain default values for *iparm* and *rparm* are used by this subroutine, where:

  If IPARM(1) = 0, the following default values are used. For restrictions, see "Notes" on page 622.

  IPARM(2) = 0
  IPARM(3) = 0
  IPARM(4) = 0
  IPARM(5) = 0
  IPARM(10) = 0
  IPARM(11) = −1
  IPARM(12) = −1
  IPARM(13) = −1
  IPARM(14) = −1
  IPARM(15) = 0
  RPARM(10) = 10^{-12}

  If IPARM(1) = 1, the default values are not used.

- IPARM(2) indicates the type of computation performed by this subroutine. The following table gives the IPARM(2) values for each variation:

Type of Computation	Gaussian Elimination ***Ax = b***	Gaussian Elimination ***Ax = b*** and Determinant(***A***)	Cholesky Decomposition ***Ax = b***	Cholesky Decomposition ***Ax = b*** and Determinant(***A***)
Factor and Solve	0	10	100	110
Factor Only	1	11	101	111
Solve Only	2	N/A	102	N/A

- IPARM(3) indicates whether a full or partial factorization is performed on matrix ***A***, where:

  If IPARM(3) = 0, and:

  If you are doing a **factor and solve** or a **factor only**, then a full factorization is performed for matrix ***A*** on rows and columns 1 through *n*. If you are doing a **solve only**, this argument has no effect on the computation, but must be set to 0.

  If IPARM(3) > 0, and you are doing a **factor only**, then a partial factorization is performed on matrix ***A***. Rows 1 through IPARM(3) of columns 1 through IPARM(3) in matrix ***A*** must be in factored form from a preceding call to this subroutine. The factorization is performed on rows IPARM(3)+1 through *n* and columns IPARM(3)+1 through *n*. For an illustration, see "Notes" on page 622.

- IPARM(4) indicates the input storage mode used for matrix ***A***. This determines the arrangement of data in arrays A and IDIAG on input, where:

  If IPARM(4) = 0, diagonal-out skyline storage mode is used.

  If IPARM(4) = 1, profile-in skyline storage mode is used.

- IPARM(5) indicates the output storage mode used for matrix ***A***. This determines the arrangement of data in arrays A and IDAIG on output, where:

  If IPARM(5) = 0, diagonal-out skyline storage mode is used.

  If IPARM(5) = 1, profile-in skyline storage mode is used.

- IPARM(6) through IPARM(9) are reserved.

- IPARM(10) has the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, then IPARM(10) indicates whether certain default values for *iparm* and *rparm* are used by this subroutine, where:

  If IPARM(10) = 0, the following default values are used. For restrictions, see "Notes" on page 622.
  IPARM(11) = −1
  IPARM(12) = −1
  IPARM(13) = −1
  IPARM(14) = −1
  IPARM(15) = 0
  RPARM(10) = 10^{-12}
  If IPARM(10) = 1, the default values are not used.

  If you are doing a **solve only**, this argument is not used.

- IPARM(11) through IPARM(15) have the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, then IPARM(11) through IPARM(15) control the type of processing to apply to pivot elements occurring in regions 1 through 5, respectively. The pivot elements are d_{kk} for Gaussian elimination and r_{kk} for Cholesky decomposition for k = 1, n when doing a full factorization, and they are k = IPARM(3)+1, n when doing a partial factorization. The region in which a pivot element falls depends on the sign and magnitude of the pivot element. The regions are determined by RPARM(10). For a description of the regions and associated pivot values, see "Notes" on page 622. For each region i for i = 1,5, where the pivot occurs in region i, the processing applied to the pivot element is determined by IPARM(10+i), where:

  If IPARM(10+i) = −1, the pivot element is trapped and computational error 2126 is generated. See "Error Conditions" on page 624.
  If IPARM(10+i) = 0, processing continues normally.

  Note: A value of 0 is not permitted for region 3, because if processing continues, a divide-by-zero exception occurs. In addition, if you are doing a Cholesky decomposition, a value of 0 is not permitted in regions 1 and 2, because a square root exception occurs.

  If IPARM(10+i) = 1, the pivot element is replaced with the value in RPARM(10+i), and processing continues normally.

  If you are doing a **solve only**, these arguments are not used.

- IPARM(16) through IPARM(25), see On Return.

Specified as: a one-dimensional array of (at least) length 25, containing fullword integers, where:

IPARM(1) = 0 or 1
IPARM(2) = 0, 1, 2, 10, 11, 100, 101, 102, 110, or 111
If IPARM(2) = 0, 2, 10, 100, 102, or 110, then IPARM(3) = 0
If IPARM(2) = 1, 11, 101, or 111, then $0 \le$ IPARM(3) $\le n$
IPARM(4), IPARM(5) = 0 or 1
If IPARM(2) = 0, 1, 10, or 11, then:
IPARM(10) = 0 or 1
IPARM(11), IPARM(12) = −1, 0, or 1
IPARM(13) = −1 or 1
IPARM(14), IPARM(15) = −1, 0, or 1

If IPARM(2) = 100, 101, 110, or 111, then:
IPARM(10) = 0 or 1
IPARM(11), IPARM(12), IPARM(13) = −1 or 1
IPARM(14), IPARM(15) = −1, 0, or 1

rparm
is an array of parameters, RPARM(*i*), where:

- RPARM(1) through RPARM(9) are reserved.
- RPARM(10) has the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, RPARM(10) is the tolerance value for small pivots. This sets the bounds for the pivot regions, where pivots are processed according to the options you specify for the five regions in IPARM(11) through IPARM(15), respectively. The suggested value is $10^{-15} \le$ IPARM(10) ≤ 1.

  If you are doing a **solve only**, this argument is not used.

- RPARM(11) through RPARM(15) have the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, RPARM(11) through RPARM(15) are the fix-up values to use for the pivots in regions 1 through 5, respectively. For each RPARM(10+*i*) for $i = 1,5$, where the pivot occurs in region *i*:

  If IPARM(10+*i*) = 1, the pivot is replaced with RPARM(10+*i*), where |RPARM(10+*i*)| should be a sufficiently large nonzero value to avoid overflow when calculating the reciprocal of the pivot. For Gaussian elimination, the suggested value is $10^{-15} \le$ |RPARM(10+*i*)| ≤ 1. For Cholesky decomposition, the value must be RPARM(10+*i*) > 0.
  If IPARM(10+*i*) ≠ 1, RPARM(10+*i*) is not used.

  If you are doing a **solve only**, these arguments are not used.

- RPARM(16) through RPARM(25), see On Return.

Specified as: a one-dimensional array of (at least) length 25, containing long-precision real numbers, where if IPARM(2) = 0, 1, 10, 11, 100, 101, 110, or 111, then:

RPARM(10) ≥ 0.0

If IPARM(2) = 0, 1, 10, or 11, then RPARM(11) through RPARM(15) ≠ 0.0
If IPARM(2) = 100, 101, 110, or 111, then RPARM(11) through RPARM(15) > 0.0

aux
has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is the storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing long-precision real numbers.

naux
is the size of the work area specified by *aux*—that is, the number of elements in *aux*.
Specified as: a fullword integer, where:

If *naux* = 0 and error 2015 is unrecoverable, DSKFS dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, If you are doing a **factor only**, you can use *naux* ≥ *n*; however, for optimal performance, use *naux* ≥ 3*n*.

If you are doing a **factor and solve** or a **solve only**, use *naux* ≥ 3*n*+4*mbx*.

For further details on error handling and the special factor-only case, see "Notes" on page 622.

bx
has the following meaning, where:

If you are doing a **factor and solve** or a **solve only**, *bx* is the array, containing the *mbx* right-hand side vectors ***b*** of the system ***Ax*** = ***b***. Each vector ***b*** is length *n* and is stored in the corresponding column of the array.

If you are doing a **factor only**, this argument is not used in the computation.

Specified as: an *ldbx* by (at least) *mbx* array, containing long-precision real numbers.

ldbx
has the following meaning, where:

If you are doing a **factor and solve** or a **solve only**, *ldbx* is the leading dimension of the array specified for *bx*.

If you are doing a **factor only**, this argument is not used in the computation.

Specified as: a fullword integer; *ldbx* ≥ *n* and:

If *mbx* ≠ 0, then *ldbx* > 0.

If *mbx* = 0, then *ldbx* ≥ 0.

mbx
has the following meaning, where:

If you are doing a **factor and solve** or a **solve only**, *mbx* is the number of right-hand side vectors, ***b***, in the array specified for *bx*.

If you are doing a **factor only**, this argument is not used in the computation.

Specified as: a fullword integer; *mbx* ≥ 0.

On Return

a
is the array, referred to as A, containing the upper triangular part of symmetric sparse matrix ***A*** in $\boldsymbol{LDL}^{T}$ or $\boldsymbol{R}^{T}\boldsymbol{R}$ factored form, where:

If IPARM(5) = 0, diagonal-out skyline storage mode is used for ***A***.

If IPARM(5) = 1, profile-in skyline storage mode is used for ***A***.

(If *mbx* = 0 and you are doing a solve only, then *a* is unchanged on output.)
Returned as: a one-dimensional array of (at least) length *na*, containing long-precision real numbers.

idiag
is the array, referred to as IDIAG, containing the relative positions of the diagonal elements of the factored output matrix ***A*** in array A. (If *mbx* = 0 and you are doing a solve only, then *idiag* is unchanged on output.)

Returned as: a one-dimensional array of (at least) length *n*+1, containing fullword integers.

iparm
is an array of parameters, IPARM(*i*), where:

- IPARM(1) through IPARM(15) are unchanged.
- IPARM(16) has the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, and:

  If IPARM(16) = −1, your factorization did not complete successfully, resulting in computational error 2126.
  If IPARM(16) > 0, it is the row number *k*, in which the maximum absolute value of the ratio a_{kk}/d_{kk} for Gaussian elimination and a_{kk}/r_{kk} for Cholesky decomposition occurred, where:
  If IPARM(3) = 0, *k* can be any of the rows, 1 through *n*, in the full factorization.
  If IPARM(3) > 0, *k* can be any of the rows, IPARM(3)+1 through *n*, in the partial factorization.

  If you are doing a **solve only**, this argument is not used in the computation and is unchanged.
- IPARM(17) through IPARM(20) are reserved.
- IPARM(21) through IPARM(25) have the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, IPARM(21) through IPARM(25) have the following meanings for each region *i* for *i* = 1,5, respectively:

  If IPARM(20+*i*) = −1, your factorization did not complete successfully, resulting in computational error 2126.
  If IPARM(20+*i*) ≥ 0, it is the number of pivots in region *i* for the columns that were factored in matrix ***A***, where:
  If IPARM(3) = 0, columns 1 through *n* were factored in the full factorization.
  If IPARM(3) > 0, columns IPARM(3)+1 through *n* were factored in the partial factorization.

If you are doing a **solve only**, these arguments are not used in the computation and are unchanged.

Returned as: a one-dimensional array of (at least) length 25, containing fullword integers.

rparm
is an array of parameters, RPARM(*i*), where:

- RPARM(1) through RPARM(15) are unchanged.
- RPARM(16) has the following meaning, where:

  If you are doing a **factor and solve** or a **factor only**, and:

  If RPARM(16) = 0.0, your factorization did not complete successfully, resulting in computational error 2126.
  If |RPARM(16)| > 0.0, it is the ratio for row k, a_{kk}/d_{kk} for Gaussian elimination and a_{kk}/r_{kk} for Cholesky decomposition, having the maximum absolute value. Row k is indicated in IPARM(16), and:
  If IPARM(3) = 0, the ratio corresponds to one of the rows, 1 through n, in the full factorization.
  If IPARM(3) > 0, the ratio corresponds to one of the rows, IPARM(3)+1 through n, in the partial factorization.

  If you are doing a **solve only**, this argument is not used in the computation and is unchanged.

- RPARM(17) and RPARM(18) have the following meaning, where:

  If you are **computing the determinant** of matrix ***A***, then RPARM(17) is the mantissa, *detbas*, and RPARM(18) is the power of 10, *detpwr*, used to express the value of the determinant: $detbas(10^{detpwr})$, where $1 \le detbas < 10$. Also:

  If IPARM(3) = 0, the determinant is computed for columns 1 through n in the full factorization.
  If IPARM(3) > 0, the determinant is computed for columns IPARM(3)+1 through n in the partial factorization.

  If you are **not computing the determinant** of matrix ***A***, these arguments are not used in the computation and are unchanged.

- RPARM(19) through RPARM(25) are reserved.

Returned as: a one-dimensional array of (at least) length 25, containing long-precision real numbers.

bx
has the following meaning, where:

If you are doing a **factor and solve** or a **solve only**, *bx* is the array, containing the *mbx* solution vectors $\boldsymbol{x}$ of the system $\boldsymbol{Ax} = \boldsymbol{b}$. Each vector $\boldsymbol{x}$ is length n and is stored in the corresponding column of the array. (If *mbx* = 0, then *bx* is unchanged on output.)

If you are doing a **factor only**, this argument is not used in the computation and is unchanged.

Returned as: an *ldbx* by (at least) *mbx* array, containing long-precision real numbers.

Notes

1. When doing a **solve only**, you should specify the same factorization method in IPARM(2), Gaussian elimination or Cholesky decomposition, that you specified for your factorization on a previous call to this subroutine.
2. If you set either IPARM(1) = 0 or IPARM(10) = 0, indicating you want to use the default values for IPARM(11) through IPARM(15) and RPARM(10), then:
   - Matrix ***A*** must be positive definite.
   - No pivots are fixed, using RPARM(11) through RPARM(15) values.
   - No small pivots are tolerated; that is, the value should be $|pivot| >$ RPARM(10).
3. Many of the input and output parameters for *iparm* and *rparm* are defined for the five pivot regions handled by this subroutine. The limits of the regions are based on RPARM(10), as shown in Figure 12. The pivot values in each region are:

   Region 1: $pivot < -$RPARM(10)
   Region 2: $-$RPARM(10) $\leq pivot < 0$
   Region 3: $pivot = 0$
   Region 4: $0 < pivot \leq$ RPARM(10)
   Region 5: $pivot >$ RPARM(10)

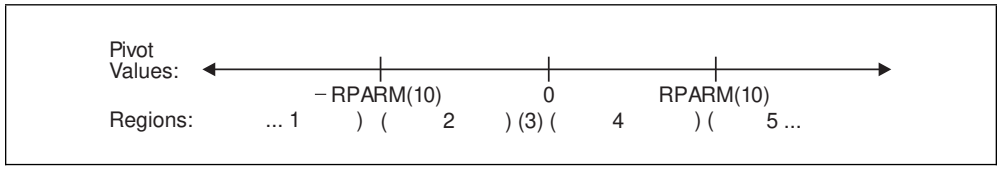


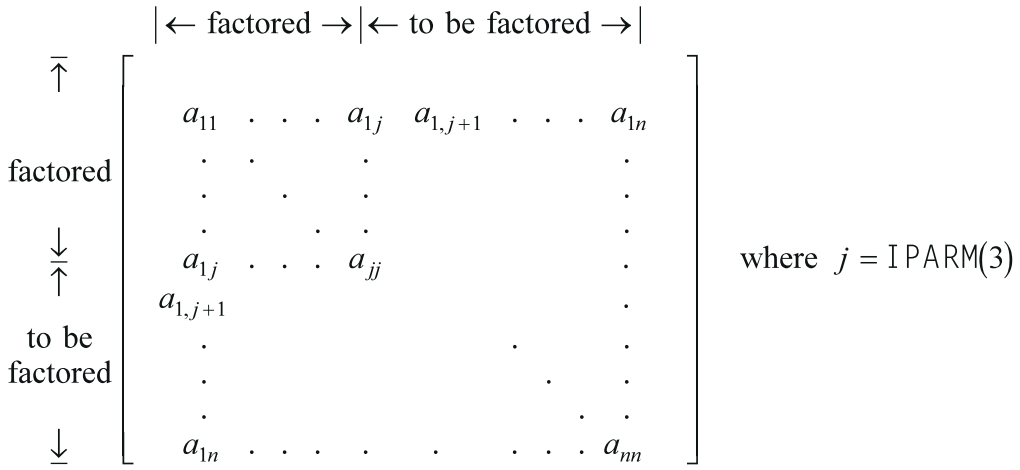
Figure 12. Five Pivot Regions

4. The IPARM(4) and IPARM(5) arguments allow you to specify the same or different skyline storage modes for your input and output arrays for matrix ***A***. This allows you to change storage modes as needed. However, if you are concerned with performance, you should use diagonal-out skyline storage mode for both input and output, if possible, because there is less overhead.

   For a description of how sparse matrices are stored in skyline storage mode, see "Profile-In Skyline Storage Mode" on page 103 and "Diagonal-Out Skyline Storage Mode" on page 100. Those descriptions use different array and variable names from the ones used here. To relate the two sets, use the following table:

Name Here	Name in the Storage Description
A	AU
na	*nu*
IDIAG	IDU

5. Following is an illustration of the portion of matrix ***A*** factored in the partial factorization when IPARM(3) > 0. In this case, the subroutine assumes that rows and columns 1 through IPARM(3) are already factored and that rows and columns IPARM(3)+1 through *n* are to be factored in this computation.



You use the partial factorization function when, for design or storage reasons, you must factor the matrix ***A*** in stages. When doing a partial factorization, you must use the same skyline storage mode for all parts of the matrix as it is progressively factored.

6. Your various arrays must have no common elements; otherwise, results are unpredictable.
7. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: This subroutine can factor, compute the determinant of, and solve symmetric sparse matrix ***A***, stored in skyline storage mode. It can use either Gaussian elimination or Cholesky decomposition. For all computations, input matrix ***A*** can be stored in either diagonal-out or profile-in skyline storage mode. Output matrix ***A*** can also be stored in either of these modes and can be different from the mode used for input.

For Gaussian elimination, matrix ***A*** is factored into the following form using specified pivot processing:

$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^{\mathrm{T}}$$

where:

D is a diagonal matrix.
L is a lower triangular matrix.

The transformed matrix ***A***, factored into its $\boldsymbol{LDL}^{\mathrm{T}}$ form, is stored in packed format in array A, such that the inverse of the diagonal matrix ***D*** is stored in the corresponding elements of array A. The off-diagonal elements of the unit upper triangular matrix $\boldsymbol{L}^{\mathrm{T}}$ are stored in the corresponding off-diagonal elements of array A.

For Cholesky decomposition, matrix ***A*** is factored into the following form using specified pivot processing:

$$\boldsymbol{A} = \boldsymbol{R}^{\mathrm{T}}\boldsymbol{R}$$

where ***R*** is an upper triangular matrix

The transformed matrix ***A***, factored into its $\boldsymbol{R}^{T}\boldsymbol{R}$ form, is stored in packed format in array A, such that the inverse of the diagonal elements of the upper triangular matrix ***R*** is stored in the corresponding elements of array A. The off-diagonal elements of matrix ***R*** are stored in the corresponding off-diagonal elements of array A.

The partial factorization of matrix ***A***, which you can do when you specify the factor-only option, assumes that the first IPARM(3) rows and columns are already factored in the input matrix. It factors the remaining n–IPARM(3) rows and columns in matrix ***A***. (See "Notes" on page 622 for an illustration.) It updates only the elements in array A corresponding to the part of matrix ***A*** that is factored.

The determinant can be computed with any of the factorization computations. With a full factorization, you get the determinant for the whole matrix. With a partial factorization, you get the determinant for only that part of the matrix factored in this computation.

The system $\boldsymbol{Ax} = \boldsymbol{b}$, having multiple right-hand sides, is solved for ***x*** using the transformed matrix ***A*** produced by this call or a subsequent call to this subroutine.

See references [9], [12], [25], [47], [65]. If n is 0, no computation is performed. If *mbx* is 0, no solve is performed.

Error Conditions

Resource Errors

- Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.
- Unable to allocate internal work area.

Computational Errors

1. If a pivot occurs in region i for i = 1,5 and IPARM(10+i) = 1, the pivot value is replaced with RPARM(10+i), an attention message is issued, and processing continues.
2. Unacceptable pivot values occurred in the factorization of matrix ***A***.
   - One or more diagonal elements of ***D*** or ***R*** contains unacceptable pivots and no valid fixup is applicable. The row number i of the first unacceptable pivot element is identified in the computational error message.
   - The return code is set to 2.
   - i can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2126 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see "What Can You Do about ESSL Computational Errors?" on page 48.

Input-Argument Errors

1. $n < 0$
2. $na < 0$
3. IDIAG(n+1) > na+1
4. IDIAG(i+1) ≤ IDIAG(i) for i = 1, n
5. IDIAG(i+1) > IDIAG(i)+i and IPARM(4) = 0 for i = 1, n
6. IDIAG(i) > IDIAG(i–1)+i and IPARM(4) = 1 for i = 2, n
7. IPARM(1) ≠ 0 or 1
8. IPARM(2) ≠ 0, 1, 2, 10, 11, 100, 101, 102, 110, or 111
9. IPARM(3) < 0
10. IPARM(3) > n

11. IPARM(3) > 0 and IPARM(2) ≠ 1, 11, 101, or 111
12. IPARM(4), IPARM(5) ≠ 0 or 1
13. IPARM(2) = 0, 1, 10, or 11 and:
    IPARM(10) ≠ 0 or 1
    IPARM(11), IPARM(12) ≠ −1, 0, or 1
    IPARM(13) ≠ −1 or 1
    IPARM(14), IPARM(15) ≠ −1, 0, or 1
    RPARM(10) < 0.0
    RPARM(10+*i*) = 0.0 and IPARM(10+*i*) = 1 for *i* = 1,5
14. IPARM(2) = 100, 101, 110, or 111 and:
    IPARM(10) ≠ 0 or 1
    IPARM(11), IPARM(12), IPARM(13) ≠ −1 or 1
    IPARM(14), IPARM(15) ≠ −1, 0, or 1
    RPARM(10) < 0.0
    RPARM(10+*i*) ≤ 0.0 and IPARM(10+*i*) = 1 for *i* = 1,5
15. IPARM(2) = 0, 2, 10, 100, 102, or 110 and:
    ldbx ≤ 0 and *mbx* ≠ 0 and *n* ≠ 0
    ldbx < 0 and *mbx* = 0
    ldbx < *n* and *mbx* ≠ 0
    mbx < 0
16. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example 1: This example shows how to factor a 9 by 9 symmetric sparse matrix $\boldsymbol{A}$ and solve the system $\boldsymbol{Ax} = \boldsymbol{b}$ with three right-hand sides. It uses Gaussian elimination. The default values are used for IPARM and RPARM. Input matrix $\boldsymbol{A}$, shown here, is stored in diagonal-out skyline storage mode. Matrix $\boldsymbol{A}$ is:

```
[                                                  ]
| 1.0  1.0  1.0  1.0  0.0  0.0  0.0  0.0  0.0 |
| 1.0  2.0  2.0  2.0  1.0  1.0  0.0  1.0  0.0 |
| 1.0  2.0  3.0  3.0  2.0  2.0  0.0  2.0  0.0 |
| 1.0  2.0  3.0  4.0  3.0  3.0  0.0  3.0  0.0 |
| 0.0  1.0  2.0  3.0  4.0  4.0  1.0  4.0  0.0 |
| 0.0  1.0  2.0  3.0  4.0  5.0  2.0  5.0  1.0 |
| 0.0  0.0  0.0  0.0  1.0  2.0  3.0  3.0  2.0 |
| 0.0  1.0  2.0  3.0  4.0  5.0  3.0  7.0  3.0 |
| 0.0  0.0  0.0  0.0  0.0  1.0  2.0  3.0  4.0 |
[                                                  ]
```

Output matrix $\boldsymbol{A}$, shown here, is in $\boldsymbol{LDL}^{\mathrm{T}}$ factored form with $\boldsymbol{D}^{-1}$ on the diagonal, and is stored in diagonal-out skyline storage mode. Matrix $\boldsymbol{A}$ is:

```
[                                                  ]
| 1.0  1.0  1.0  1.0  0.0  0.0  0.0  0.0  0.0 |
| 1.0  1.0  1.0  1.0  1.0  1.0  0.0  1.0  0.0 |
| 1.0  1.0  1.0  1.0  1.0  1.0  0.0  1.0  0.0 |
| 1.0  1.0  1.0  1.0  1.0  1.0  0.0  1.0  0.0 |
| 0.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  0.0 |
| 0.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
| 0.0  0.0  0.0  0.0  1.0  1.0  1.0  1.0  1.0 |
| 0.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
| 0.0  0.0  0.0  0.0  0.0  1.0  1.0  1.0  1.0 |
[                                                  ]
```

Call Statement and Input

```
            N  A  NA  IDIAG  IPARM  RPARM  AUX NAUX  BX  LDBX MBX
            |  |  |     |      |      |     |   |    |    |    |
CALL DSKFS( 9, A, 33, IDIAG, IPARM, RPARM, AUX, 39 , BX , 12 , 3 )

A        =  (1.0, 2.0, 1.0, 3.0, 2.0, 1.0, 4.0, 3.0, 2.0, 1.0, 4.0,
             3.0, 2.0, 1.0, 5.0, 4.0, 3.0, 2.0, 1.0, 3.0, 2.0, 1.0,
             7.0, 3.0, 5.0, 4.0, 3.0, 2.0, 1.0, 4.0, 3.0, 2.0, 1.0)
IDIAG    =  (1, 2, 4, 7, 11, 15, 20, 23, 30, 34)
IPARM    =  (0, . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
             . , . , . , . , . , . , . , . , . , . , . )

RPARM    =(not relevant)

         ⎡                    ⎤
         ⎢  4.00   8.00  12.00 ⎥
         ⎢ 10.00  20.00  30.00 ⎥
         ⎢ 15.00  30.00  45.00 ⎥
         ⎢ 19.00  38.00  57.00 ⎥
         ⎢ 19.00  38.00  57.00 ⎥
BX   =   ⎢ 23.00  46.00  69.00 ⎥
         ⎢ 11.00  22.00  33.00 ⎥
         ⎢ 28.00  56.00  84.00 ⎥
         ⎢ 10.00  20.00  30.00 ⎥
         ⎢   .      .      .   ⎥
         ⎢   .      .      .   ⎥
         ⎣   .      .      .   ⎦
```

Output

```
A        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
             1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
             1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IDIAG    =(same as input)
IPARM    =  (0, . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
             . , 8, . , . , . , . , 0, 0, 0, 0, 9)
RPARM    =  ( . , . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
             . , 7.0, . , . , . , . , . , . , . , . , . )

         ⎡                    ⎤
         ⎢ 1.00  2.00  3.00 ⎥
         ⎢ 1.00  2.00  3.00 ⎥
         ⎢ 1.00  2.00  3.00 ⎥
         ⎢ 1.00  2.00  3.00 ⎥
         ⎢ 1.00  2.00  3.00 ⎥
BX   =   ⎢ 1.00  2.00  3.00 ⎥
         ⎢ 1.00  2.00  3.00 ⎥
         ⎢ 1.00  2.00  3.00 ⎥
         ⎢ 1.00  2.00  3.00 ⎥
         ⎢  .     .     .   ⎥
         ⎢  .     .     .   ⎥
         ⎣  .     .     .   ⎦
```

Example 2: This example shows how to factor the 9 by 9 symmetric sparse matrix $\boldsymbol{A}$ from Example 1, solve the system $\boldsymbol{Ax} = \boldsymbol{b}$ with three right-hand sides, and compute the determinant of $\boldsymbol{A}$. It uses Gaussian elimination. The default values for pivot processing are used for IPARM. Input matrix $\boldsymbol{A}$ is stored in profile-in skyline storage mode. Output matrix $\boldsymbol{A}$ is in $\boldsymbol{LDL}^{\mathrm{T}}$ factored form with $\boldsymbol{D}^{-1}$ on the diagonal, and is stored in diagonal-out skyline storage mode. It is the same as output matrix $\boldsymbol{A}$ in Example 1.

Call Statement and Input

```
            N  A  NA  IDIAG  IPARM  RPARM  AUX NAUX  BX  LDBX MBX
            |  |  |     |      |      |     |   |    |    |    |
CALL DSKFS( 9, A, 33, IDIAG, IPARM, RPARM, AUX, 39 , BX , 12 , 3 )

A        =  (1.0, 1.0, 2.0, 1.0, 2.0, 3.0, 1.0, 2.0, 3.0, 4.0, 1.0,
             2.0, 3.0, 4.0, 1.0, 2.0, 3.0, 4.0, 5.0, 1.0, 2.0, 3.0,
             1.0, 2.0, 3.0, 4.0, 5.0, 3.0, 7.0, 1.0, 2.0, 3.0, 4.0)
IDIAG    =  (1, 3, 6, 10, 14, 19, 22, 29, 33, 34)
IPARM    =  (1, 10, 0, 1, 0, . , . , . , . , 0, . , . , . , . , . ,
             . , . , . , . , . , . , . , . , . , . )

RPARM    =(not relevant)

         ⌈                    ⌉
         |  4.00   8.00  12.00 |
         | 10.00  20.00  30.00 |
         | 15.00  30.00  45.00 |
         | 19.00  38.00  57.00 |
         | 19.00  38.00  57.00 |
BX    =  | 23.00  46.00  69.00 |
         | 11.00  22.00  33.00 |
         | 28.00  56.00  84.00 |
         | 10.00  20.00  30.00 |
         |   .      .      .   |
         |   .      .      .   |
         |   .      .      .   |
         ⌊                    ⌋
```

Output

```
A        =(same as output A in Example 1)
IDIAG    =(same as input IDIAG in Example 1)
IPARM    =  (1, 10, 0, 1, 0, . , . , . , . , 0, . , . , . , . , . , 8,
             . , . , . , . , 0, 0, 0, 0, 9)
RPARM    =  ( . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
             . , 7.0, 1.0, 0.0, . , . , . , . , . , . , . )
BX       =(same as output BX in Example 1)
```

Example 3: This example shows how to factor a 9 by 9 negative-definite symmetric sparse matrix $\boldsymbol{A}$, solve the system $\boldsymbol{Ax} = \boldsymbol{b}$ with three right-hand sides, and compute the determinant of $\boldsymbol{A}$. It uses Gaussian elimination. (Default values for pivot processing are not used for IPARM because $\boldsymbol{A}$ is negative-definite.) Input matrix $\boldsymbol{A}$, shown here, is stored in diagonal-out skyline storage mode. Matrix $\boldsymbol{A}$ is:

$$\begin{bmatrix} -1.0 & -1.0 & -1.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ -1.0 & -2.0 & -2.0 & -2.0 & -1.0 & -1.0 & 0.0 & -1.0 & 0.0 \\ -1.0 & -2.0 & -3.0 & -3.0 & -2.0 & -2.0 & 0.0 & -2.0 & 0.0 \\ -1.0 & -2.0 & -3.0 & -4.0 & -3.0 & -3.0 & 0.0 & -3.0 & 0.0 \\ 0.0 & -1.0 & -2.0 & -3.0 & -4.0 & -4.0 & -1.0 & -4.0 & 0.0 \\ 0.0 & -1.0 & -2.0 & -3.0 & -4.0 & -5.0 & -2.0 & -5.0 & -1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & -1.0 & -2.0 & -3.0 & -3.0 & -2.0 \\ 0.0 & -1.0 & -2.0 & -3.0 & -4.0 & -5.0 & -3.0 & -7.0 & -3.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -1.0 & -2.0 & -3.0 & -4.0 \end{bmatrix}$$

Output matrix ***A***, shown here, is in $\boldsymbol{LDL}^{T}$ factored form with $\boldsymbol{D}^{-1}$ on the diagonal, and is stored in diagonal-out skyline storage mode. Matrix ***A*** is:

$$\begin{bmatrix} -1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & -1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 1.0 & -1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 1.0 & 0.0 \\ 1.0 & 1.0 & 1.0 & -1.0 & 1.0 & 1.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 1.0 & 1.0 & -1.0 & 1.0 & 1.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 1.0 & 1.0 & 1.0 & -1.0 & 1.0 & 1.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & -1.0 & 1.0 & 1.0 \\ 0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & -1.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 1.0 & -1.0 \end{bmatrix}$$

Call Statement and Input

```
          N  A  NA  IDIAG  IPARM  RPARM  AUX  NAUX  BX  LDBX MBX
          |  |  |     |      |      |     |    |    |    |    |
CALL DSKFS(9, A, 33, IDIAG, IPARM, RPARM, AUX,  39 , BX , 12 , 3 )

A       =  (-1.0, -2.0, -1.0, -3.0, -2.0, -1.0, -4.0, -3.0, -2.0,
            -1.0, -4.0, -3.0, -2.0, -1.0, -5.0, -4.0, -3.0, -2.0,
            -1.0, -3.0, -2.0, -1.0, -7.0, -3.0, -5.0, -4.0, -3.0,
            -2.0, -1.0, -4.0, -3.0, -2.0, -1.0)
IDIAG   =  (1, 2, 4, 7, 11, 15, 20, 23, 30, 34)
IPARM   =  (1, 10, 0, 0, 0, . , . , . , . , . , 1, 0, -1, -1, -1, -1, . ,
            . , . , . , . , . , . , . , . , . , . )

RPARM   =  ( . , . , . , . , . , . , . , . , . , . , . , 10**-15, . , . ,
             . , . , . , . , . , . , . , . , . , . , . , . , . , . , . )
BX      =(same as input BX in Example 1)
```

Output:

```
A       =  (-1.0, -1.0, 1.0, -1.0, 1.0, 1.0, -1.0, 1.0, 1.0, 1.0,
            -1.0, 1.0, 1.0, 1.0, -1.0, 1.0, 1.0, 1.0, 1.0, -1.0, 1.0,
            1.0, -1.0 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, 1.0, 1.0,
            1.0)
IDIAG   =(same as input)
IPARM   =  (1, 10, 0, 0, 0, . , . , . , . , . , 1, 0, -1, -1, -1, -1, 8,
            . , . , . , . , . , 9, 0, 0, 0, 0)

RPARM   =  ( . , . , . , . , . , . , . , . , . , . , . , . , . ,10**-15, . , . ,
             . , . , . , . , 7.0, -1.0, 0.0, . , . , . , . , . , . , . , . )
```

```
        ⎡                    ⎤
        ⎢ -1.00  -2.00  -3.00 ⎥
        ⎢ -1.00  -2.00  -3.00 ⎥
        ⎢ -1.00  -2.00  -3.00 ⎥
        ⎢ -1.00  -2.00  -3.00 ⎥
        ⎢ -1.00  -2.00  -3.00 ⎥
BX   =  ⎢ -1.00  -2.00  -3.00 ⎥
        ⎢ -1.00  -2.00  -3.00 ⎥
        ⎢ -1.00  -2.00  -3.00 ⎥
        ⎢ -1.00  -2.00  -3.00 ⎥
        ⎢   .      .      .   ⎥
        ⎢   .      .      .   ⎥
        ⎢   .      .      .   ⎥
        ⎣                    ⎦
```

Example 4: This example shows how to factor the first six rows and columns, referred to as matrix ***A1***, of the 9 by 9 symmetric sparse matrix ***A*** from Example 1 and compute the determinant of ***A1***. It uses Gaussian elimination. Input matrix ***A1***, shown here, is stored in diagonal-out skyline storage mode. Input matrix ***A1*** is:

```
⎡                                    ⎤
⎢ 1.0  1.0  1.0  1.0  0.0  0.0 ⎥
⎢ 1.0  2.0  2.0  2.0  1.0  1.0 ⎥
⎢ 1.0  2.0  3.0  3.0  2.0  2.0 ⎥
⎢ 1.0  2.0  3.0  4.0  3.0  3.0 ⎥
⎢ 0.0  1.0  2.0  3.0  4.0  4.0 ⎥
⎢ 0.0  1.0  2.0  3.0  4.0  5.0 ⎥
⎣                                    ⎦
```

Output matrix ***A1***, shown here, is in $\boldsymbol{LDL}^{\mathrm{T}}$ factored form with $\boldsymbol{D}^{-1}$ on the diagonal, and is stored in diagonal-out skyline storage mode. Output matrix ***A1*** is:

```
⎡                                    ⎤
⎢ 1.0  1.0  1.0  1.0  0.0  0.0 ⎥
⎢ 1.0  1.0  1.0  1.0  1.0  1.0 ⎥
⎢ 1.0  1.0  1.0  1.0  1.0  1.0 ⎥
⎢ 1.0  1.0  1.0  1.0  1.0  1.0 ⎥
⎢ 0.0  1.0  1.0  1.0  1.0  1.0 ⎥
⎢ 0.0  1.0  1.0  1.0  1.0  1.0 ⎥
⎣                                    ⎦
```

Call Statement and Input

```
            N   A   NA   IDIAG   IPARM   RPARM   AUX  NAUX  BX   LDBX   MBX
            |   |    |     |       |       |      |    |    |     |      |
CALL DSKFS (6 , A , 33 , IDIAG , IPARM , RPARM , AUX , 27 , BX , LDBX , MBX )

A        =(same as input A in Example 1)
IDIAG    =  (1, 2, 4, 7, 11, 15, 20)
IPARM    =  (1, 11, 0, 0, 0, . , . , . , . , 0, . , . , . , . , . ,
             . , . , . , . , . , . , . , . , . , . )
RPARM    =(not relevant)
BX       =(not relevant)
LDBX     =(not relevant)
MBX      =(not relevant)
```

Output

```
A        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
             1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 3.0, 2.0, 1.0,
             7.0, 3.0, 5.0, 4.0, 3.0, 2.0, 1.0, 4.0, 3.0, 2.0, 1.0)
IDIAG    =(same as input)
IPARM    =  (1, 11, 0, 0, 0, . , . , . , . , 0, . , . , . , . , . , 6,
             . , . , . , . , 0, 0, 0, 0, 6)
RPARM    =  ( . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
             . , 5.0, 1.0, 0.0, . , . , . , . , . , . , . )
BX       =(same as input)
LDBX     =(same as input)
MBX      =(same as input)
```

Example 5: This example shows how to do a partial factorization of the 9 by 9 symmetric sparse matrix ***A*** from Example 1, where the first six rows and columns were factored in Example 4. It factors the remaining three rows and columns and computes the determinant of that part of the matrix. It uses Gaussian elimination. The input matrix, referred to as ***A2***, shown here, is made up of the output factored matrix ***A1*** plus the three remaining unfactored rows and columns of matrix ***A*** Matrix ***A2*** is:

```
 ┌                                                   ┐
 │ 1.0  1.0  1.0  1.0  0.0  0.0  0.0  0.0  0.0 │
 │ 1.0  1.0  1.0  1.0  1.0  1.0  0.0  1.0  0.0 │
 │ 1.0  1.0  1.0  1.0  1.0  1.0  0.0  2.0  0.0 │
 │ 1.0  1.0  1.0  1.0  1.0  1.0  0.0  3.0  0.0 │
 │ 0.0  1.0  1.0  1.0  1.0  1.0  1.0  4.0  0.0 │
 │ 0.0  1.0  1.0  1.0  1.0  1.0  2.0  5.0  1.0 │
 │ 0.0  0.0  0.0  0.0  1.0  2.0  3.0  3.0  2.0 │
 │ 0.0  1.0  2.0  3.0  4.0  5.0  3.0  7.0  3.0 │
 │ 0.0  0.0  0.0  0.0  0.0  1.0  2.0  3.0  4.0 │
 └                                                   ┘
```

Both parts of input matrix ***A2*** are stored in diagonal-out skyline storage mode.

Output matrix ***A2*** is the same as output matrix ***A*** in Example 1 and is stored in diagonal-out skyline storage mode.

Call Statement and Input

```
            N   A   NA   IDIAG   IPARM   RPARM   AUX   NAUX  BX   LDBX   MBX
            |   |   |      |       |       |      |     |    |     |      |
CALL DSKFS (9 , A , 33 , IDIAG , IPARM , RPARM , AUX ,  27 , BX , LDBX , MBX )
```

```
A        =(same as output A in Example 4)
IDIAG    =(same as input IDIAG in Example 1)
IPARM    =  (1, 11, 6, 0, 0, . , . , . , . , 0, . , . , . , . , . ,
             . , . , . , . , . , . , . , . , . , . )
RPARM    =(not relevant)
BX       =(not relevant)
LDBX     =(not relevant)
MBX      =(not relevant)
```

Output

```
A        =(same as output A in Example 1)
```

```
IDIAG    =(same as output IDIAG in Example 1)
IPARM    =  (1, 11, 6, 0, 0, . , . , . , . , . , 0, . , . , . , . , . , . , 8,
             . , . , . , . , 0, 0, 0, 0, 3)
RPARM    =  ( . , . , . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
             . , 7.0, 1.0, 0.0, . , . , . , . , . , . , . , . )
BX       =(same as input)
LDBX     =(same as input)
MBX      =(same as input)
```

Example 6: This example shows how to solve the system $\boldsymbol{Ax} = \boldsymbol{b}$ with one right-hand side for a symmetric sparse matrix $\boldsymbol{A}$. Input matrix $\boldsymbol{A}$, used here, is the same as factored output matrix $\boldsymbol{A}$ from Example 1, stored in profile-in skyline storage mode. It specifies Gaussian elimination, as used in Example 1. Here, output matrix $\boldsymbol{A}$ is unchanged on output and is stored in profile-in skyline storage mode.

Call Statement and Input

```
            N  A  NA  IDIAG  IPARM  RPARM  AUX NAUX  BX  LDBX MBX
            |  |  |     |      |      |     |   |    |    |    |
CALL DSKFS (9, A, 33, IDIAG, IPARM, RPARM, AUX, 31 , BX , 9 ,  1 )

A        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
             1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
             1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IDIAG    =  (1, 3, 6, 10, 14, 19, 22, 29, 33, 34)
IPARM    =  (1, 2, 0, 1, 1, . , . , . , . , . , . , . , . , . , . , . , . ,
             . , . , . , . , . , . , . , . , . , . , . , . )
RPARM    =(not relevant)
BX       =  (10.0, 38.0, 64.0, 87.0, 103.0, 133.0, 80.0, 174.0, 80.0)
```

Output

```
A        =(same as input)
IDIAG    =(same as input)
IPARM    =(same as input)
APARM    =(same as input)
BX       =  (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0)
```

Example 7: This example shows how to factor a 9 by 9 symmetric sparse matrix $\boldsymbol{A}$ and solve the system $\boldsymbol{Ax} = \boldsymbol{b}$ with four right-hand sides. It uses Cholesky decomposition. Input matrix $\boldsymbol{A}$, shown here, is stored in profile-in skyline storage mode Matrix $\boldsymbol{A}$ is:

$$\begin{bmatrix} 1.0 & 1.0 & 1.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 1.0 \\ 1.0 & 5.0 & 3.0 & 0.0 & 3.0 & 0.0 & 0.0 & 0.0 & 3.0 \\ 1.0 & 3.0 & 11.0 & 3.0 & 5.0 & 3.0 & 3.0 & 0.0 & 5.0 \\ 0.0 & 0.0 & 3.0 & 17.0 & 5.0 & 5.0 & 5.0 & 0.0 & 5.0 \\ 1.0 & 3.0 & 5.0 & 5.0 & 29.0 & 7.0 & 7.0 & 0.0 & 9.0 \\ 0.0 & 0.0 & 3.0 & 5.0 & 7.0 & 39.0 & 9.0 & 6.0 & 9.0 \\ 0.0 & 0.0 & 3.0 & 5.0 & 7.0 & 9.0 & 53.0 & 8.0 & 11.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 6.0 & 8.0 & 66.0 & 10.0 \\ 1.0 & 3.0 & 5.0 & 5.0 & 9.0 & 9.0 & 11.0 & 10.0 & 89.0 \end{bmatrix}$$

Output matrix ***A***, shown here, is in $\boldsymbol{R}^{\mathrm{T}}\boldsymbol{R}$ factored form with the inverse of the diagonal of ***R*** on the diagonal, and is stored in profile-in skyline storage mode. Matrix ***A*** is:

```
 ⎡                                                                ⎤
 | 1.0  1.0   1.0  0.0  1.0   0.0   0.0   0.0   1.0 |
 | 1.0   .5   1.0  0.0  1.0   0.0   0.0   0.0   1.0 |
 | 1.0  1.0  .333  1.0  1.0   1.0   1.0   0.0   1.0 |
 | 0.0  0.0   1.0  .25  1.0   1.0   1.0   0.0   1.0 |
 | 1.0  1.0   1.0  1.0   .2   1.0   1.0   0.0   1.0 |
 | 0.0  0.0   1.0  1.0  1.0  .167   1.0   1.0   1.0 |
 | 0.0  0.0   1.0  1.0  1.0   1.0  .143   1.0   1.0 |
 | 0.0  0.0   0.0  0.0  0.0   1.0   1.0  .125   1.0 |
 | 1.0  1.0   1.0  1.0  1.0   1.0   1.0   1.0  .111 |
 ⎣                                                                ⎦
```

Call Statement and Input

```
            N  A  NA  IDIAG  IPARM  RPARM  AUX NAUX  BX  LDBX MBX
            |  |  |     |      |      |     |   |    |    |    |
CALL DSKFS( 9, A, 34, IDIAG, IPARM, RPARM, AUX, 43 , BX , 10 , 4 )

A        =  (1.0, 1.0, 5.0, 1.0, 3.0, 11.0, 3.0, 17.0, 1.0, 3.0, 5.0,
             5.0, 29.0, 3.0, 5.0, 7.0, 39.0, 3.0, 5.0, 7.0, 9.0, 53.0,
             6.0, 8.0, 66.0, 1.0, 3.0, 5.0, 5.0, 9.0, 9.0, 11.0, 10.0,
             89.0)
IDIAG    =  (1, 3, 6, 8, 13, 17, 22, 25, 34, 35)
IPARM    =  (1, 110, 0, 1, 1, . , . , . , . , 0, . , . , . , . , . ,
             . , . , . , . , . , . , . , . , . , . )

RPARM    =(not relevant)

         ⎡                                   ⎤
         |   5.00   10.00   15.00   20.00 |
         |  15.00   30.00   45.00   60.00 |
         |  34.00   68.00  102.00  136.00 |
         |  40.00   80.00  120.00  160.00 |
BX   =   |  66.00  132.00  198.00  264.00 |
         |  78.00  156.00  234.00  312.00 |
         |  96.00  192.00  288.00  384.00 |
         |  90.00  180.00  270.00  360.00 |
         | 142.00  284.00  426.00  568.00 |
         |    .       .       .       .   |
         ⎣                                   ⎦
```

Output

```
A        =  (1.0, 1.0, .5, 1.0, 1.0, .333, 1.0, .25, 1.0, 1.0, 1.0,
             1.0, .2, 1.0, 1.0, 1.0, .167, 1.0, 1.0, 1.0, 1.0, .143,
             1.0, 1.0, .125, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
             .111)
IDIAG    =(same as input)
IPARM    =  (1, 110, 0, 1, 1, . , . , . , . , 0, . , . , . , . , . ,
             9, . , . , . , . , 0, 0, 0, 0, 9)
RPARM    =  ( . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
             . , 9.89, 1.32, 11.0, . , . , . , . , . , . , . )
```

$$BX = \begin{bmatrix} 1.00 & 2.00 & 3.00 & 4.00 \\ 1.00 & 2.00 & 3.00 & 4.00 \\ 1.00 & 2.00 & 3.00 & 4.00 \\ 1.00 & 2.00 & 3.00 & 4.00 \\ 1.00 & 2.00 & 3.00 & 4.00 \\ 1.00 & 2.00 & 3.00 & 4.00 \\ 1.00 & 2.00 & 3.00 & 4.00 \\ 1.00 & 2.00 & 3.00 & 4.00 \\ 1.00 & 2.00 & 3.00 & 4.00 \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

DSRIS—Iterative Linear System Solver for a General or Symmetric Sparse Matrix Stored by Rows

This subroutine solves a general or symmetric sparse linear system of equations, using an iterative algorithm, with or without preconditioning. The methods include conjugate gradient (CG), conjugate gradient squared (CGS), generalized minimum residual (GMRES), more smoothly converging variant of the CGS method (Bi-CGSTAB), or transpose-free quasi-minimal residual method (TFQMR). The preconditioners include an incomplete LU factorization, an incomplete Cholesky factorization (for positive definite symmetric matrices), diagonal scaling, or symmetric successive over-relaxation (SSOR) with two possible choices for the diagonal matrix: one uses the absolute values sum of the input matrix, and the other uses the diagonal obtained from the LU factorization. The sparse matrix is stored using storage-by-rows for general matrices and upper- or lower-storage-by-rows for symmetric matrices. Matrix ***A*** and vectors ***x*** and ***b*** are used:

$$\boldsymbol{Ax} = \boldsymbol{b}$$

where ***A***, ***x***, and ***b*** contain long-precision real numbers.

Syntax

Fortran	CALL DSRIS (*stor*, *init*, *n*, *ar*, *ja*, *ia*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	dsris (*stor*, *init*, *n*, *ar*, *ja*, *ia*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*);
PL/I	CALL DSRIS (*stor*, *init*, *n*, *ar*, *ja*, *ia*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

stor

indicates the form of sparse matrix ***A*** and the storage mode used, where:

If *stor* = 'G', ***A*** is a general sparse matrix, stored using storage-by-rows.

If *stor* = 'U', ***A*** is a symmetric sparse matrix, stored using upper-storage-by-rows.

If *stor* = 'L', ***A*** is a symmetric sparse matrix, stored using lower-storage-by-rows.

Specified as: a single character. It must be 'G', 'U', or 'L'.

init

indicates the type of computation to be performed, where:

If *init* = 'I', the preconditioning matrix is computed, the internal representation of the sparse matrix is generated, and the iteration procedure is performed. The coefficient matrix and preconditioner in internal format are saved in *aux1*.

If *init* = 'S', the iteration procedure is performed using the coefficient matrix and the preconditioner in internal format, stored in *aux1*, created in a preceding call to this subroutine with *init* = 'I'. You use this option to solve the same matrix for different right-hand sides, ***b***, optimizing your performance. As long as you do not change the coefficient matrix and preconditioner in *aux1*, any number of calls can be made with *init* = 'S'.

Specified as: a single character. It must be 'I' or 'S'.

n

is the order of the linear system $\boldsymbol{Ax} = \boldsymbol{b}$ and the number of rows and columns in sparse matrix ***A***. Specified as: a fullword integer; $n \geq 0$.

ar
is the sparse matrix ***A*** of order *n*, stored by rows in an array, referred to as AR. The *stor* argument indicates the storage variation used for storing matrix ***A***. Specified as: a one-dimensional array, containing long-precision real numbers. The number of elements in this array can be determined by subtracting 1 from the value in IA(*n*+1).

ja
is the array, referred to as JA, containing the column numbers of each nonzero element in sparse matrix **A**. Specified as: a one-dimensional array, containing fullword integers; $1 \le$ (JA elements) $\le n$. The number of elements in this array can be determined by subtracting 1 from the value in IA(*n*+1).

ia
is the row pointer array, referred to as IA, containing the starting positions of each row of matrix ***A*** in array AR and one position past the end of array AR. Specified as: a one-dimensional array of (at least) length *n*+1, containing fullword integers; IA(*i*+1) $\ge$ IA(*i*) for $i = 1, n+1$.

b
is the vector ***b*** of length *n*, containing the right-hand side of the matrix problem. Specified as: a one-dimensional array of (at least) length *n*, containing long-precision real numbers.

x
is the vector ***x*** of length *n*, containing your initial guess of the solution of the linear system. Specified as: a one-dimensional array of (at least) length *n*, containing long-precision real numbers. The elements can have any value, and if no guess is available, the value can be zero.

iparm
is an array of parameters, IPARM(*i*), where:

- IPARM(1) controls the number of iterations.

  If IPARM(1) > 0, IPARM(1) is the maximum number of iterations allowed.

  If IPARM(1) = 0, the following default values are used:

  ```
  IPARM(1) = 300
  IPARM(2) = 4
  IPARM(4) = 4
  IPARM(5) = 1
  RPARM(1) = 10**-6
  RPARM(2) = 1
  ```

- IPARM(2) is the flag used to select the iterative procedure used in this subroutine.

  If IPARM(2) = 1, the conjugate gradient (CG) method is used. Note that this algorithm should only be used with positive definite symmetric matrices.

  If IPARM(2) = 2, the conjugate gradient squared (CGS) method is used.

  If IPARM(2) = 3, the generalized minimum residual (GMRES) method, restarted after *k* steps, is used.

  If IPARM(2) = 4, the more smoothly converging variant of the CGS method (Bi-CGSTAB) is used.

  If IPARM(2) = 5, the transpose-free quasi-minimal residual method (TFQMR) is used.

- IPARM(3) has the following meaning, where:

  If IPARM(2) $\ne$ 3, then IPARM(3) is not used.

If IPARM(2) = 3, then IPARM(3) = k, where k is the number of steps after which the generalized minimum residual method is restarted. A value for k in the range of 5 to 10 is suitable for most problems.

- IPARM(4) is the flag that determines the type of preconditioning.

  If IPARM(4) = 1, the system is not preconditioned.

  If IPARM(4) = 2, the system is preconditioned by a diagonal matrix.

  If IPARM(4) = 3, the system is preconditioned by SSOR splitting with the diagonal given by the absolute values sum of the input matrix.

  If IPARM(4) = 4, the system is preconditioned by an incomplete LU factorization.

  If IPARM(4) = 5, the system is preconditioned by SSOR splitting with the diagonal given by the incomplete LU factorization.

  Note: The multithreaded version of DSRIS only runs on multiple threads when IPARM(4) = 1 or 2.

- IPARM(5) is the flag used to select the stopping criterion used in the computation, where the following items are used in the definitions of the stopping criteria below:

  – ε is the desired relative accuracy and is stored in RPARM(1).

  – x_j is the solution found at the j-th iteration.

  – r_j and r_0 are the preconditioned residuals obtained at iterations j and 0, respectively. (The residual at iteration j is given by $\boldsymbol{b}-\boldsymbol{A}x_j$.)

  If IPARM(5) = 1, the iterative method is stopped when:

  $$\|r_j\|_2 \,/\, \|x_j\|_2 < \varepsilon$$

  Note: IPARM(5) = 1 is the default value assumed by ESSL if you do not specify one of the values described here; therefore, if you do not update your program to set an IPARM(5) value, you, by default, use the above stopping criterion.

  If IPARM(5) = 2, the iterative method is stopped when:

  $$\|r_j\|_2 \,/\, \|r_0\|_2 < \varepsilon$$

  If IPARM(5) = 3, the iterative method is stopped when:

  $$\|x_j - x_{j-1}\|_2 \,/\, \|x_j\|_2 < \varepsilon$$

  Note: Stopping criterion 3 performs poorly with the TFQMR method; therefore, if you specify TFQMR (IPARM(2) = 5), you should not specify stopping criterion 3.

- IPARM(6), see On Return.

Specified as: an array of (at least) length 6, containing fullword integers, where:
IPARM(1) ≥ 0
IPARM(2) = 1, 2, 3, 4, or 5
If IPARM(2) = 3, then IPARM(3) > 0
IPARM(4) = 1, 2, 3, 4, or 5
IPARM(5) = 1, 2, or 3 (Other values default to stopping criterion 1.)

rparm
is an array of parameters, RPARM(*i*), where:

RPARM(1) is the relative accuracy ϵ used in the stopping criterion. See "Notes" on page 639.

RPARM(2), see On Return.

RPARM(3) has the following meaning, where:

- If IPARM(4) $\neq$ 3, then RPARM(3) is not used.
- If IPARM(4) = 3, then RPARM(3) is the acceleration parameter used in SSOR. (A value in the range 0.5 to 2.0 is suitable for most problems.)

Specified as: a one-dimensional array of (at least) length 3, containing long-precision real numbers, where:
RPARM(1) $\geq$ 0
If IPARM(4) = 3, RPARM(3) > 0

aux1
is working storage for this subroutine, where:

If *init* = 'I', the working storage is computed. It can contain any values.

If *init* = 'S', the working storage is used in solving the linear system. It contains the coefficient matrix and preconditioner in internal format, computed in an earlier call to this subroutine.

Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1
is the number of doublewords in the working storage specified in *aux1*.

Specified as: a fullword integer, where:

In these formulas *nw* has the following value:
If *stor* = 'G', then *nw* = IA(*n*+1)−1+*n*.
If *stor* = 'U' or 'L', then *nw* = 2(IA(*n*+1)−1).

If IPARM(4) = 1, use *naux1* = (3/2)*nw*+(7/2)*n*+40.
If IPARM(4) = 2, use *naux1* = (3/2)*nw*+(9/2)*n*+40.
If IPARM(4) = 3, 4, or 5, then:
If IPARM(2) $\neq$ 1, use *naux1* = 3*nw*+10*n*+60.
If IPARM(2) = 1, use *naux1* = 3*nw*+(21/2)*n*+60.

Note: If you receive an attention message, you have not specified sufficient auxiliary storage to achieve optimal performance, but it is enough to perform the computation. To obtain optimal performance, you need to use the amount given by the attention message.

aux2
has the following meaning:

If *naux2* = 0 and error 2015 is unrecoverable, *aux2* is ignored.

Otherwise, it is working storage used by this subroutine that is available for use by the calling program between calls to this subroutine.

Specified as: an area of storage, containing *naux2* long-precision real numbers.

naux2
is the number of doublewords in the working storage specified in *aux2*. Specified as: a fullword integer, where:

If *naux2* = 0 and error 2015 is unrecoverable, DSRIS dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise,
If IPARM(2) = 1, use *naux*2 ≥ 4*n*.
If IPARM(2) = 2, use *naux*2 ≥ 7*n*.
If IPARM(2) = 3, use *naux*2 ≥ (*k*+2)*n*+*k*(*k*+4)+1, where *k* = IPARM(3).
If IPARM(2) = 4, use *naux*2 ≥ 7*n*.
If IPARM(2) = 5, use *naux*2 ≥ 9*n*.

On Return

ar
is the sparse matrix ***A*** of order *n*, stored by rows in an array, referred to as AR. The *stor* argument indicates the storage variation used for storing matrix ***A***. The order of the elements in each row of ***A*** in AR may be changed on output.

Returned as: a one-dimensional array, containing long-precision real numbers. The number of elements in this array can be determined by subtracting 1 from the value in IA(*n*+1).

ja
is the array, referred to as JA, containing the column numbers of each nonzero element in sparse matrix **A**. These elements correspond to the arrangement of the contents of AR on output.

Returned as: a one-dimensional array, containing fullword integers; 1 ≤ (JA elements) ≤ *n*. The number of elements in this array can be determined by subtracting 1 from the value in IA(*n*+1).

x
is the vector ***x*** of length *n*, containing the solution of the system ***Ax*** = ***b***. Returned as: a one-dimensional array of (at least) length *n*, containing long-precision real numbers.

iparm
is an array of parameters, IPARM(*i*), where:

IPARM(1) through IPARM(5) are unchanged.

IPARM(6) contains the number of iterations performed by this subroutine.

Returned as: a one-dimensional array of length 6, containing fullword integers.

rparm
is an array of parameters, RPARM(*i*), where:

RPARM(1) is unchanged.

RPARM(2) contains the estimate of the error of the solution. If the process converged, RPARM(2) ≤ ϵ.

RPARM(3) is unchanged.

Returned as: a one-dimensional array of length 3, containing long-precision real numbers.

aux1
is working storage for this subroutine, containing the coefficient matrix and preconditioner in internal format, ready to be passed in a subsequent invocation of this subroutine. Returned as: an area of storage, containing *naux1* long-precision real numbers.

Notes

1. If you want to solve the same sparse linear system of equations multiple times using a different algorithm with the same preconditioner and using a different right-hand side each time, you get the best performance by using the following technique. Call DSRIS the first time with *init* = 'I'. This solves the system, and then stores the coefficient matrix and preconditioner in internal format in *aux1*. On the subsequent invocations of DSRIS with different right-hand sides, specify *init* = 'S'. This indicates to DSRIS to use the contents of *aux1*, saving the time to convert your coefficient matrix and preconditioner to internal format. If you use this technique, you should not modify the contents of *aux1* between calls to DSRIS.

   In some cases, you can specify a different algorithm in IPARM(2) when making calls with *init* = 'S'. (See "Example 2" on page 642.) However, DSRIS sometimes needs different information in *aux1* for different algorithms. When this occurs, DSRIS issues an attention message, continues processing the computation, and then resets the contents of *aux1*. Your performance is not improved in this case, which is functionally equivalent to calling DSRIS with *init* = 'I'.

2. If you use the CG method with *init* = 'I', you must use the CG method when you specify *init* = 'S'. However, if you use a different method with *init* = 'I', you can use any other method, except CG, when you specify *init* = 'S'.

3. These subroutines accept lowercase letters for the *stor* and *init* arguments.

4. Matrix $\boldsymbol{A}$, vector $\boldsymbol{x}$, and vector $\boldsymbol{b}$ must have no common elements; otherwise, results are unpredictable.

5. In this subroutine, a value of RPARM(1) = 0 is permitted to force the solver to evaluate exactly IPARM(1) iterations. The algorithm computes a sequence of approximate solution vectors $\boldsymbol{x}$ that converge to the solution. The iterative procedure is stopped when the selected stopping criterion is satisfied or when more than the maximum number of iterations (in IPARM(1)) is reached.

   For the stopping criteria specified in IPARM(5), the relative accuracy ε (in RPARM(1)) must be specified reasonably (10^{-4} to 10^{-8}). If you specify a larger ε, the algorithm takes fewer iterations to converge to a solution. If you specify a smaller ε, the algorithm requires more iterations and computer time, but converges to a more precise solution. If the value you specify is unreasonably small, the algorithm may fail to converge within the number of iterations it is allowed to perform.

6. For a description of how sparse matrices are stored by rows, see "Storage-by-Rows" on page 99.

7. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: The linear system:

$$\boldsymbol{Ax} = \boldsymbol{b}$$

is solved using one of the following methods: conjugate gradient (CG), conjugate gradient squared (CGS), generalized minimum residual (GMRES), more smoothly converging variant of the CGS method (Bi-CGSTAB), or transpose-free quasi-minimal residual method (TFQMR), where:

A is a sparse matrix of order *n*. The matrix is stored in arrays AR, IA, and JA. If it is general, it is stored by rows. If it is symmetric, it can be stored using upper- or lower-storage-by-rows.
x is a vector of length *n*.
b is a vector of length *n*.

One of the following preconditioners is used:

- an incomplete LU factorization
- an incomplete Cholesky factorization (for positive definite symmetric matrices)
- diagonal scaling
- symmetric successive over-relaxation (SSOR) with two possible choices for the diagonal matrix:
  - the absolute values sum of the input matrix
  - the diagonal obtained from the LU factorization

See references [36], [53], [76], [80], [83], and [89].

When you call this subroutine to solve a system for the first time, you specify *init* = 'I'. After that, you can solve the same system any number of times by calling this subroutine each time with *init* = 'S'. These subsequent calls use the coefficient matrix and preconditioner, stored in internal format in *aux1*. You optimize performance by doing this, because certain portions of the computation have already been performed.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux2* = 0, and unable to allocate work area.

Computational Errors: The following errors, with their corresponding return codes, can occur in this subroutine. For details on error handling, see "What Can You Do about ESSL Computational Errors?" on page 48.

- For error 2110, return code 1 indicates that the subroutine exceeded IPARM(1) iterations without converging. Vector ***x*** contains the approximate solution computed at the last iteration.
- For error 2130, return code 2 indicates that the incomplete LU factorization of ***A*** could not be completed, because one pivot was 0.
- For error 2124, the subroutine has been called with *init* = 'S', but the data contained in *aux1* was computed for a different algorithm. An attention message is issued. Processing continues, and the contents of *aux1* are reset correctly.
- For error 2134, return code 3 indicates that the data contained in *aux1* is not consistent with the input sparse matrix. The subroutine has been called with *init* = 'S', and *aux1* contains an incomplete factorization and internal data storage for the input matrix ***A*** that was computed by a previous call to the subroutine when *init* = 'I'. This error indicates that *aux1* has been modified since the last call to the subroutine, or that the input matrix is not the same as the one that was factored. If the default action has been overridden, the subroutine can be called again with the same parameters, with the exception of IPARM(4) = 1 or 4.
- For error 2131, return code 4 indicates that the matrix is singular, because all elements in one row of the matrix contain zero.
- For error 2129, return code 5 indicates that the matrix is not positive definite.

- For error 2128, return code 8 indicates an internal ESSL error. Please contact your IBM Representative.

Input-Argument Errors

1. $n < 0$
2. *stor* ≠ 'G', 'U', or 'L'
3. *init* ≠ 'I' or 'S'
4. IA(n+1) < 1
5. IA(i+1)–IA(i) < 0, for any $i = 1, n$
6. IPARM(1) < 0
7. IPARM(2) ≠ 1, 2, 3, 4, or 5
8. IPARM(3) ≤ 0 and IPARM(2) = 3
9. IPARM(4) ≠ 1, 2, 3, 4, or 5
10. RPARM(1) < 0
11. RPARM(3) ≤ 0 and IPARM(4) = 3
12. *naux1* is too small—that is, less than the minimum required value. Return code 6 is returned if error 2015 is recoverable.
13. Error 2015 is recoverable or *naux2*≠0, and *naux2* is too small—that is, less than the minimum required value. Return code 7 is returned for *naux2* if error 2015 is recoverable.

Example 1: This example finds the solution of the linear system $\boldsymbol{Ax} = \boldsymbol{b}$ for the sparse matrix $\boldsymbol{A}$, which is stored by rows in arrays AR, IA, and JA. The system is solved using the Bi-CGSTAB algorithm. The iteration is stopped when the norm of the residual is less than the given threshold specified in RPARM(1). The algorithm is allowed to perform 20 iterations. The process converges after 9 iterations. Matrix $\boldsymbol{A}$ is:

```
⎡                                                    ⎤
| 2.0  0.0   0.0  0.0   0.0   0.0   0.0   0.0   0.0 |
| 0.0  2.0  -1.0  0.0   0.0   0.0   0.0   0.0   0.0 |
| 0.0  1.0   2.0  0.0   0.0   0.0   0.0   0.0   0.0 |
| 1.0  0.0   0.0  2.0  -1.0   0.0   0.0   0.0   0.0 |
| 0.0  0.0   0.0  1.0   2.0  -1.0   0.0   0.0   0.0 |
| 0.0  0.0   0.0  0.0   1.0   2.0  -1.0   0.0   0.0 |
| 0.0  0.0   0.0  0.0   0.0   1.0   2.0  -1.0   0.0 |
| 0.0  0.0   0.0  0.0   0.0   0.0   1.0   2.0  -1.0 |
| 0.0  0.0   0.0  0.0   0.0   0.0   0.0   1.0   2.0 |
⎣                                                    ⎦
```

Call Statement and Input

```
            STOR INIT   N   AR   JA   IA   B   X   IPARM   RPARM   AUX1   NAUX1   AUX2   NAUX2
             |    |    |   |    |    |    |   |     |       |       |      |       |      |
CALL DSRIS( 'G' , 'I' , 9 , AR , JA , IA , B , X , IPARM , RPARM , AUX1 ,  98  , AUX2 ,  63 )
```

```
AR       =  (2.0, 2.0, -1.0, 1.0, 2.0, 1.0, 2.0, -1.0, 1.0, 2.0, -1.0,
             1.0, 2.0, -1.0, 1.0, 2.0, -1.0, 1.0, 2.0, -1.0, 1.0, 2.0)
JA       =  (1, 2, 3, 2, 3, 1, 4, 5, 4, 5, 6, 5, 6, 7, 6, 7, 8, 7, 8,
             9, 8, 9)
IA       =  (1, 2, 4, 6, 9, 12, 15, 18, 21, 23)
B        =  (2.0, 1.0, 3.0, 2.0, 2.0, 2.0, 2.0, 2.0, 3.0)
X        =  (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
IPARM(1) =  20
IPARM(2) =  4
IPARM(3) =  0
IPARM(4) =  1
IPARM(5) =  10
RPARM(1) =  1.D-7
RPARM(3) =  1.0
```

Output

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(6) =  9
RPARM(2) =  0.29D-16
```

Example 2: This example finds the solution of the linear system $\boldsymbol{Ax} = \boldsymbol{b}$ for the same sparse matrix $\boldsymbol{A}$ used in Example 1. It also uses the same right-hand side in $\boldsymbol{b}$ and the same initial guesses in $\boldsymbol{x}$. However, the system is solved using a different algorithm, conjugate gradient squared (CGS). Because INIT is 'S', the best performance is achieved. The iteration is stopped when the norm of the residual is less than the given threshold specified in RPARM(1). The algorithm is allowed to perform 20 iterations. The process converges after 9 iterations.

Call Statement and Input

```
            STOR INIT   N   AR   JA   IA   B   X   IPARM   RPARM   AUX1   NAUX1   AUX2   NAUX2
             |     |    |   |    |    |    |   |     |       |      |       |      |       |
CALL DSRIS( 'G' , 'S' , 9 , AR , JA , IA , B , X , IPARM , RPARM , AUX1 ,   98  , AUX2 ,   63 )
```

```
AR       =(same as input AR in Example 1)
JA       =(same as input JA in Example 1)
IA       =(same as input IA in Example 1)
B        =(same as input B in Example 1)
X        =(same as input X in Example 1)
IPARM(1) =  20
IPARM(2) =  2
IPARM(3) =  0
IPARM(4) =  1
IPARM(5) =  10
RPARM(1) =  1.D-7
RPARM(3) =  1.0
```

Output

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(6) =  9
RPARM(2) =  0.42D-19
```

Example 3: This example finds the solution of the linear system $\boldsymbol{Ax} = \boldsymbol{b}$ for the sparse matrix $\boldsymbol{A}$, which is stored by rows in arrays AR, IA, and JA. The system is solved using the two-term conjugate gradient method (CG), preconditioned by incomplete LU factorization. The iteration is stopped when the norm of the residual is less than the given threshold specified in RPARM(1). The algorithm is allowed to perform 20 iterations. The process converges after 1 iteration. Matrix $\boldsymbol{A}$ is:

$$\begin{bmatrix} 2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ -1.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & -1.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & -1.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & -1.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & -1.0 & 0.0 & 2.0 & 0.0 & -1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -1.0 & 0.0 & 2.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -1.0 & 0.0 & 2.0 \end{bmatrix}$$

Call Statement Input

```
            STOR INIT   N   AR   JA   IA   B   X   IPARM   RPARM   AUX1   NAUX1   AUX2   NAUX2
             |     |    |   |    |    |    |   |     |       |      |       |      |       |
CALL DSRIS( 'G' , 'I' , 9 , AR , JA , IA , B , X , IPARM , RPARM , AUX1 ,  223  , AUX2 ,   36 )
```

```
AR       =  (2.0, -1.0, 2.0, -1.0, -1.0, 2.0, -1.0, -1.0, 2.0, -1.0,
             -1.0, 2.0, -1.0, -1.0, 2.0, -1.0, -1.0, 2.0, -1.0, -1.0,
             2.0, -1.0, 2.0)
JA       =  (1, 3, 2, 4, 1, 3, 5, 2, 4, 6, 3, 5, 7, 4, 6, 8, 5, 7, 9,
             6, 8, 7, 9)
IA       =  (1, 3, 5, 8, 11, 14, 17, 20, 22, 24)
B        =  (1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0)
X        =  (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
IPARM(1) =  20
IPARM(2) =  1
IPARM(3) =  0
IPARM(4) =  4
IPARM(5) =  1
RPARM(1) =  1.D-7
RPARM(3) =  1.0
```

Output

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(6) =  1
RPARM(2) =  0.16D-15
```

Example 4: This example finds the solution of the linear system $\boldsymbol{Ax} = \boldsymbol{b}$ for the same sparse matrix $\boldsymbol{A}$ used in Example 3. However, matrix $\boldsymbol{A}$ is stored using upper-storage-by-rows in arrays AR, IA, and JA. The system is solved using the generalized minimum residual (GMRES), restarted after 5 steps and preconditioned with SSOR splitting. The iteration is stopped when the norm of the residual is less than the given threshold specified in RPARM(1). The algorithm is allowed to perform 20 iterations. The process converges after 12 iterations.

Call Statement Input

```
            STOR INIT   N   AR   JA   IA   B   X   IPARM   RPARM   AUX1   NAUX1   AUX2   NAUX2
             |    |    |   |    |    |    |   |     |       |       |       |      |       |
CALL DSRIS( 'U' , 'I' , 9 , AR , JA , IA , B , X , IPARM , RPARM , AUX1 ,  219  , AUX2 ,  109 )
```

```
AR       =  (2.0, -1.0, 2.0, -1.0, 2.0, -1.0, 2.0, -1.0, 2.0, -1.0,
             2.0, -1.0, 2.0, -1.0, 2.0, 2.0)
JA       =  (1, 3, 2, 4, 3, 5, 4, 6, 5, 7, 6, 8, 7, 9, 8, 9)
IA       =  (1, 3, 5, 7, 9, 11, 13, 15, 16, 17)
B        =  (1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0)
X        =  (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
IPARM(1) =  20
IPARM(2) =  3
IPARM(3) =  5
IPARM(4) =  3
IPARM(5) =  1
RPARM(1) =  1.D-7
RPARM(3) =  2.0
```

Output

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(6) =  12
RPARM(2) =  0.33D-7
```

DSMCG—Sparse Positive Definite or Negative Definite Symmetric Matrix Iterative Solve Using Compressed-Matrix Storage Mode

This subroutine solves a symmetric, positive definite or negative definite linear system, using the conjugate gradient method, with or without preconditioning by an incomplete Cholesky factorization, for a sparse matrix stored in compressed-matrix storage mode. Matrix ***A*** and vectors ***x*** and ***b*** are used:

$$\boldsymbol{Ax} = \boldsymbol{b}$$

where ***A***, ***x***, and ***b*** contain long-precision real numbers.

Notes:

1. These subroutines are provided only for migration purposes. You get better performance and a wider choice of algorithms if you use the DSRIS subroutine.
2. If your sparse matrix is stored by rows, as defined in "Storage-by-Rows" on page 99, you should first use the utility subroutine DSRSM to convert your sparse matrix to compressed-matrix storage mode. See "DSRSM—Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode" on page 986

Syntax

Fortran	CALL DSMCG (*m*, *nz*, *ac*, *ka*, *lda*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	dsmcg (*m*, *nz*, *ac*, *ka*, *lda*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*);
PL/I	CALL DSMCG (*m*, *nz*, *ac*, *ka*, *lda*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

m
is the order of the linear system $\boldsymbol{Ax} = \boldsymbol{b}$ and the number of rows in sparse matrix ***A***. Specified as: a fullword integer; $m \geq 0$.

nz
is the maximum number of nonzero elements in each row of sparse matrix ***A***. Specified as: a fullword integer; $nz \geq 0$.

ac
is the array, referred to as AC, containing the values of the nonzero elements of the sparse matrix, stored in compressed-matrix storage mode. Specified as: an *lda* by (at least) *nz* array, containing long-precision real numbers.

ka
is the array, referred to as KA, containing the column numbers of the matrix ***A*** elements stored in the corresponding positions in array AC. Specified as: an *lda* by (at least) *nz* array, containing fullword integers, where $1 \leq$ (elements of KA) $\leq m$.

lda
is the leading dimension of the arrays specified for *ac* and *ka*. Specified as: a fullword integer; $lda > 0$ and $lda \geq m$.

b
is the vector ***b*** of length *m*, containing the right-hand side of the matrix problem. Specified as: a one-dimensional array of (at least) length *m*, containing long-precision real numbers.

x
is the vector $\boldsymbol{x}$ of length *m*, containing your initial guess of the solution of the linear system. Specified as: a one-dimensional array of (at least) length *m*, containing long-precision real numbers. The elements can have any value, and if no guess is available, the value can be zero.

iparm
is an array of parameters, IPARM(*i*), where:

- IPARM(1) controls the number of iterations.

  If IPARM(1) > 0, IPARM(1) is the maximum number of iterations allowed.

  If IPARM(1) = 0, the following default values are used:

  ```
  IPARM(1) = 300
  IPARM(2) = 1
  IPARM(3) = 0
  RPARM(1) = 10**-6
  ```

- IPARM(2) is the flag used to select the stopping criterion.

  If IPARM(2) = 0, the conjugate gradient iterative procedure is stopped when:

  $$\|\boldsymbol{r}\|_2 / \|\boldsymbol{x}\|_2 < \varepsilon$$

  where $\boldsymbol{r} = \boldsymbol{b}-\boldsymbol{A}\boldsymbol{x}$ is the residual, and ε is the desired relative accuracy. ε is stored in RPARM(1).

  If IPARM(2) = 1, the conjugate gradient iterative procedure is stopped when:

  $$\|\boldsymbol{r}\|_2 / \lambda\|\boldsymbol{x}\|_2 < \varepsilon$$

  where λ is an estimate to the minimum eigenvalue of the iteration matrix. λ is computed adaptively by this program and, on output, is stored in RPARM(2).

  If IPARM(2) = 2, the conjugate gradient iterative procedure is stopped when:

  $$\|\boldsymbol{r}\|_2 / \lambda\|\boldsymbol{x}\|_2 < \varepsilon$$

  where λ is a predetermined estimate to the minimum eigenvalue of the iteration matrix. This eigenvalue estimate, on input, is stored in RPARM(2) and may be obtained by an earlier call to this subroutine with the same matrix.

- IPARM(3) is the flag that determines whether the system is to be solved using the conjugate gradient method, preconditioned by an incomplete Cholesky factorization with no fill-in.

  If IPARM(3) = 0, the system is not preconditioned.

  If IPARM(3) = 10, the system is preconditioned by an incomplete Cholesky factorization.

  If IPARM(3) = −10, the system is preconditioned by an incomplete Cholesky factorization, where the factorization matrix was computed in an earlier call to this subroutine and is stored in *aux2*.

- IPARM(4), see On Return.

Specified as: an array of (at least) length 4, containing fullword integers, where:
IPARM(1) ≥ 0
IPARM(2) = 0, 1, or 2

IPARM(3) = 0, 10, or −10

rparm
is an array of parameters, RPARM(*i*), where ε is stored in RPARM(1), and λ is stored in RPARM(2).

RPARM(1) > 0, is the relative accuracy ϵ used in the stopping criterion.

RPARM(2) > 0, is the estimate of the smallest eigenvalue, λ, of the iteration matrix. It is only used when IPARM(2) = 2.

RPARM(3), see On Return.

Specified as: a one-dimensional array of (at least) length 3, containing long-precision real numbers.

aux1
has the following meaning:

If *naux1* = 0 and error 2015 is unrecoverable, *aux1* is ignored.

Otherwise, it is a storage work area used by this subroutine, which is available for use by the calling program between calls to this subroutine. Its size is specified by *naux1*.

Specified as: an area of storage, containing long-precision real numbers.

naux1
is the size of the work area specified by *aux1*—that is, the number of elements in *aux1*. Specified as: a fullword integer, where:

If *naux1* = 0 and error 2015 is unrecoverable, DSMCG dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux1* must have at least the following value, where:

If IPARM(2) = 0 or 2, use *naux1* ≥ 3*m*.

If IPARM(2) = 1 and IPARM(1) ≠ 0, use *naux1* ≥ 3*m*+2(IPARM(1)).

If IPARM(2) = 1 and IPARM(1) = 0, use *naux1* ≥ 3*m*+600.

aux2
is a storage work area used by this subroutine. If IPARM(3) = −10, *aux2* must contain the incomplete Cholesky factorization of matrix ***A***, computed in an earlier call to DSMCG. The size of *aux2* is specified by *naux2*. Specified as: an area of storage, containing long-precision real numbers.

naux2
is the size of the work area specified by *aux2*—that is, the number of elements in *aux2*. Specified as: a fullword integer. When IPARM(3) = 10 or −10, *naux2* must have at least the following value: *naux2* ≥ *m*(*nz*−1)1.5+2(*m*+6).

On Return

x
is the vector ***x*** of length *m*, containing the solution of the system ***Ax*** = ***b***. Returned as: a one-dimensional array of (at least) length *m*, containing long-precision real numbers.

iparm
is an array of parameters, IPARM(*i*), where:

IPARM(1) is unchanged.

IPARM(2) is unchanged.

IPARM(3) is unchanged.

IPARM(4) contains the number of iterations performed by this subroutine.

Returned as: a one-dimensional array of length 4, containing fullword integers.

rparm
is an array of parameters, RPARM(*i*), where:

RPARM(1) is unchanged.

RPARM(2) is unchanged if IPARM(2) = 0 or 2. If IPARM(2) = 1, RPARM(2) contains λ, an estimate of the smallest eigenvalue of the iteration matrix.

RPARM(3) contains the estimate of the error of the solution. If the process converged, RPARM(3) $\leq \epsilon$.

Returned as: a one-dimensional array of length 3, containing long-precision real numbers; $\lambda > 0$.

aux2
is the storage work area used by this subroutine.

If IPARM(3) = 10, *aux2* contains the incomplete Cholesky factorization of matrix ***A***.

If IPARM(3) = −10, *aux2* is unchanged.

See "Notes" for additional information on *aux2*. Returned as: an area of storage, containing long-precision real numbers.

Notes

1. When IPARM(3) = −10, this subroutine uses the incomplete Cholesky factorization in *aux2*, computed in an earlier call to this subroutine. When IPARM(3) = 10, this subroutine computes the incomplete Cholesky factorization and stores it in *aux2*.
2. If you solve the same sparse linear system of equations several times with different right-hand sides using the preconditioned algorithm, specify IPARM(3) = 10 on the first invocation. The incomplete factorization is stored in *aux2*. You may save computing time on subsequent calls by setting IPARM(3) = −10. In this way, the algorithm reutilizes the incomplete factorization that was computed the first time. Therefore, you should not modify the contents of *aux2* between calls.
3. Matrix ***A*** must have no common elements with vectors ***x*** and ***b***; otherwise, results are unpredictable.
4. In the iterative solvers for sparse matrices, the relative accuracy ε (RPARM(1)) must be specified "reasonably" (10^{-4} to 10^{-8}). The algorithm computes a sequence of approximate solution vectors ***x*** that converge to the solution. The iterative procedure is stopped when the norm of the residual is sufficiently small—that is, when:

   $$\|\boldsymbol{b}-\boldsymbol{A}\boldsymbol{x}\|_2 \,/\, \lambda\|\boldsymbol{x}\|_2 < \varepsilon$$

   where λ is an estimate of the minimum eigenvalue of the iteration matrix, which is either estimated adaptively or given by the user. As a result, if you specify a larger ε, the algorithm takes fewer iterations to converge to a solution. If you specify a smaller ε, the algorithm requires more iterations and computer time, but converges to a more precise solution. If the value you specify is unreasonably small, the algorithm may fail to converge within the number of iterations it is allowed to perform.
5. For a description of how sparse matrices are stored in compressed-matrix storage mode, see "Compressed-Matrix Storage Mode" on page 93.

6. On output, array AC and vector ***b*** are not bitwise identical to what they were on input, because the matrix ***A*** and the right-hand side are scaled before starting the iterative process and are unscaled before returning control to the user. In addition, arrays AC and KA may be rearranged on output, but still contain a mathematically equivalent mapping of the elements in matrix ***A***.
7. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: The sparse positive definite or negative definite linear system:

$$\boldsymbol{Ax} = \boldsymbol{b}$$

is solved, where:

A is a symmetric, positive definite or negative definite sparse matrix of order *m*, stored in compressed-matrix storage mode in AC and KA.
x is a vector of length *m*.
b is a vector of length *m*.

The system is solved using the two-term conjugate gradient method, with or without preconditioning by an incomplete Cholesky factorization. In both cases, the matrix is scaled by the square root of the diagonal.

See references [59] and [62]. [36].

If your program uses a sparse matrix stored by rows and you want to use this subroutine, first convert your sparse matrix to compressed-matrix storage mode by using the subroutine DSRSM described on page 986.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux1* = 0, and unable to allocate work area.

Computational Errors: The following errors, with their corresponding return codes, can occur in this subroutine. Where a value of *i* is indicated, it can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for that particular error code in the ESSL error option table; otherwise, the default value causes your program to terminate when the error occurs. For details, see "What Can You Do about ESSL Computational Errors?" on page 48.

- For error 2110, return code 1 indicates that the subroutine exceeded IPARM(1) iterations without converging. Vector ***x*** contains the approximate solution computed at the last iteration.
- For error 2111, return code 2 indicates that *aux2* contains an incorrect factorization. The subroutine has been called with IPARM(3) = −10, and *aux2* contains an incomplete factorization of the input matrix ***A*** that was computed by a previous call to the subroutine when IPARM(3) = 10. This error indicates that *aux2* has been modified since the last call to the subroutine, or that the input matrix is not the same as the one that was factored. If the default action has been overridden, the subroutine can be called again with the same parameters, with the exception of IPARM(3) = 0 or 10.

- For error 2109, return code 3 indicates that the inner product ($\boldsymbol{y}$,$\boldsymbol{Ay}$) is negative in the iterative procedure after iteration i. This should not occur, because the input matrix is assumed to be positive or negative definite. Vector $\boldsymbol{x}$ contains the results of the last iteration. The value i is identified in the computational error message.
- For error 2108, return code 4 indicates that the matrix is not positive definite. AC is partially modified and does not represent the same matrix as on entry.

Input-Argument Errors

1. $m < 0$
2. $lda < 1$
3. $lda < m$
4. $nz < 0$
5. $nz = 0$ and $m > 0$
6. IPARM(1) < 0
7. IPARM(2) $\neq$ 0, 1, or 2
8. IPARM(3) $\neq$ 0, 10, or –10
9. RPARM(1) < 0
10. RPARM(2) < 0
11. Error 2015 is recoverable or *naux1*$\neq$0, and *naux1* is too small—that is, less than the minimum required value. Return code 5 is returned if error 2015 is recoverable.
12. *naux2* is too small—that is, less than the minimum required value. Return code 5 is returned if error 2015 is recoverable.

Example 1: This example finds the solution of the linear system $\boldsymbol{Ax} = \boldsymbol{b}$ for the sparse matrix $\boldsymbol{A}$, which is stored in compressed-matrix storage mode in arrays AC and KA. The system is solved using the conjugate gradient method. Matrix $\boldsymbol{A}$ is:

$$\begin{bmatrix} 2.0 & 0.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & -1.0 & 2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ -1.0 & 0.0 & 0.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & -1.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & -1.0 & 2.0 & -1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -1.0 & 2.0 & -1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -1.0 & 2.0 & -1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -1.0 & 2.0 \end{bmatrix}$$

Note: For input matrix KA, (.) indicates any value between 1 and 9.

Call Statement and Input

```
           M   NZ  AC  KA LDA  B   X  IPARM  RPARM  AUX1 NAUX1  AUX2 NAUX2
           |   |   |   |   |   |   |    |      |     |     |     |     |
CALL DSMCG( 9 , 3 , AC, KA, 9 , B , X, IPARM, RPARM, AUX1, 27  , AUX2,  0  )

IPARM(1) =  20
IPARM(2) =  0
IPARM(3) =  0
RPARM(1) =  1.D-7
```

```
        ⎡                    ⎤
        ⎢  2.0  -1.0   0.0 ⎥
        ⎢  2.0  -1.0   0.0 ⎥
        ⎢ -1.0   2.0   0.0 ⎥
        ⎢ -1.0   2.0  -1.0 ⎥
AC   =  ⎢ -1.0   2.0  -1.0 ⎥
        ⎢ -1.0   2.0  -1.0 ⎥
        ⎢ -1.0   2.0  -1.0 ⎥
        ⎢ -1.0   2.0  -1.0 ⎥
        ⎢ -1.0   2.0   0.0 ⎥
        ⎣                    ⎦

        ⎡          ⎤
        ⎢ 1  4  . ⎥
        ⎢ 2  3  . ⎥
        ⎢ 2  3  . ⎥
        ⎢ 1  4  5 ⎥
KA   =  ⎢ 4  5  6 ⎥
        ⎢ 5  6  7 ⎥
        ⎢ 6  7  8 ⎥
        ⎢ 7  8  9 ⎥
        ⎢ 8  9  . ⎥
        ⎣          ⎦

B        =  (1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0)
X        =  (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
```

Output

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(4) =  5
RPARM(2) =  0
RPARM(3) =  0.351D-15
```

Example 2: This example finds the solution of the linear system $\boldsymbol{Ax} = \boldsymbol{b}$ for the same sparse matrix $\boldsymbol{A}$ as in Example 1, which is stored in compressed-matrix storage mode in arrays AC and KA. The system is solved using the conjugate gradient method, preconditioned with an incomplete Cholesky factorization. The smallest eigenvalue of the iteration matrix is computed and used in stopping the computation.

Note: For input matrix KA, (.) indicates any value between 1 and 9.

Call Statement and Input

```
            M   NZ  AC  KA LDA  B   X  IPARM  RPARM  AUX1 NAUX1  AUX2 NAUX2
            |   |   |   |   |   |   |    |      |     |     |     |     |
CALL DSMCG( 9 , 3 , AC, KA, 9 , B , X, IPARM, RPARM, AUX1, 67  , AUX2, 74  )

IPARM(1) =  20
IPARM(2) =  1
IPARM(3) =  10
RPARM(1) =  1.D-7
AC       =(same as input AC in Example 1)
KA       =(same as input KA in Example 1)
B        =  (1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0)
X        =  (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
```

Output

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(4) =  1
RPARM(2) =  1
RPARM(3) =  0.100D-15
```

DSDCG—Sparse Positive Definite or Negative Definite Symmetric Matrix Iterative Solve Using Compressed-Diagonal Storage Mode

This subroutine solves a symmetric, positive definite or negative definite linear system, using the two-term conjugate gradient method, with or without preconditioning by an incomplete Cholesky factorization, for a sparse matrix stored in compressed-diagonal storage mode. Matrix ***A*** and vectors ***x*** and ***b*** are used:

$$\boldsymbol{Ax} = \boldsymbol{b}$$

where ***A***, ***x***, and ***b*** contain long-precision real numbers.

Syntax

Fortran	CALL DSDCG (*iopt*, *m*, *nd*, *ad*, *lda*, *la*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	dsdcg (*iopt*, *m*, *nd*, *ad*, *lda*, *la*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*);
PL/I	CALL DSDCG (*iopt*, *m*, *nd*, *ad*, *lda*, *la*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

iopt
indicates the type of storage used, where:

If *iopt* = 0, all the nonzero diagonals of the sparse matrix are stored in compressed-diagonal storage mode.

If *iopt* = 1, the sparse matrix, stored in compressed-diagonal storage mode, is symmetric. Only the main diagonal and one of each pair of identical diagonals are stored in array AD.

Specified as: a fullword integer; *iopt* = 0 or 1.

m
is the order of the linear system ***Ax*** = ***b*** and the number of rows in sparse matrix ***A***. Specified as: a fullword integer; $m \geq 0$.

nd
is the number of nonzero diagonals stored in the columns of array AD, the number of columns in the array AD, and the number of elements in array LA. Specified as: a fullword integer; it must have the following value, where:

If $m > 0$, then $nd > 0$.

If $m = 0$, then $nd \geq 0$.

ad
is the array, referred to as AD, containing the values of the nonzero elements of the sparse matrix stored in compressed-diagonal storage mode. If *iopt* = 1, the main diagonal and one of each pair of identical diagonals is stored in this array.

Specified as: an *lda* by (at least) *nd* array, containing long-precision real numbers.

lda
is the leading dimension of the array specified for *ad*. Specified as: a fullword integer; $lda > 0$ and $lda \geq m$.

la
is the array, referred to as LA, containing the diagonal numbers *k* for the diagonals stored in each corresponding column in array AD. For an explanation of how diagonal numbers are assigned, see "Compressed-Diagonal Storage Mode" on page 94.

Specified as: a one-dimensional array of (at least) length *nd*, containing fullword integers, where $1-m \le$ (elements of LA) $\le m-1$.

b

is the vector ***b*** of length *m*, containing the right-hand side of the matrix problem. Specified as: a one-dimensional array of (at least) length *m*, containing long-precision real numbers.

x

is the vector ***x*** of length *m*, containing your initial guess of the solution of the linear system. Specified as: a one-dimensional array of (at least) length *m*, containing long-precision real numbers. The elements can have any value, and if no guess is available, the value can be zero.

iparm

is an array of parameters, IPARM(*i*), where:

- IPARM(1) controls the number of iterations.

  If IPARM(1) > 0, IPARM(1) is the maximum number of iterations allowed.

  If IPARM(1) = 0, the following default values are used:

  ```
  IPARM(1) = 300
  IPARM(2) = 1
  IPARM(3) = 0
  RPARM(1) = 10**-6
  ```

- IPARM(2) is the flag used to select the stopping criterion.

  If IPARM(2) = 0, the conjugate gradient iterative procedure is stopped when:

  $$\|\boldsymbol{r}\|_2 / \|\boldsymbol{x}\|_2 < \varepsilon$$

  where $\boldsymbol{r} = \boldsymbol{b}-\boldsymbol{A}\boldsymbol{x}$ is the residual and ε is the desired relative accuracy. ε is stored in RPARM(1).

  If IPARM(2) = 1, the conjugate gradient iterative procedure is stopped when:

  $$\|\boldsymbol{r}\|_2 / \lambda\|\boldsymbol{x}\|_2 < \varepsilon$$

  where λ is an estimate to the minimum eigenvalue of the iteration matrix. λ is computed adaptively by this program and, on output, is stored in RPARM(2).

  If IPARM(2) = 2, the conjugate gradient iterative procedure is stopped when:

  $$\|\boldsymbol{r}\|_2 / \lambda\|\boldsymbol{x}\|_2 < \varepsilon$$

  where λ is a predetermined estimate to the minimum eigenvalue of the iteration matrix. This eigenvalue estimate, on input, is stored in RPARM(2) and may be obtained by an earlier call to this subroutine with the same matrix.

- IPARM(3) is the flag that determines whether the system is to be solved using the conjugate gradient method, preconditioned by an incomplete Cholesky factorization with no fill-in.

  If IPARM(3) = 0, the system is not preconditioned.

  If IPARM(3) = 10, the system is preconditioned by an incomplete Cholesky factorization.

If IPARM(3) = −10, the system is preconditioned by an incomplete Cholesky factorization, where the factorization matrix was computed in an earlier call to this subroutine and is stored in *aux2*.

- IPARM(4), see On Return.

Specified as: an array of (at least) length 4, containing fullword integers, where:
IPARM(1) = 0
IPARM(2) = 0, 1, or 2
IPARM(3) = 0, 10, or −10

rparm
is an array of parameters, RPARM(*i*), where ε is stored in RPARM(1), and λ is stored in RPARM(2).

RPARM(1) > 0, is the relative accuracy ϵ used in the stopping criterion.

RPARM(2) > 0, is the estimate of the smallest eigenvalue, λ, of the iteration matrix. It is only used when IPARM(2) = 2.

RPARM(3), see On Return.

Specified as: a one-dimensional array of (at least) length 3, containing long-precision real numbers.

aux1
has the following meaning:

If *naux1* = 0 and error 2015 is unrecoverable, *aux1* is ignored.

Otherwise, it is a storage work area used by this subroutine, which is available for use by the calling program between calls to this subroutine. Its size is specified by *naux1*.

Specified as: an area of storage, containing long-precision real numbers.

naux1
is the size of the work area specified by *aux1*—that is, the number of elements in *aux1*.

Specified as: a fullword integer, where:

If *naux* = 0 and error 2015 is unrecoverable, DSDCG dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, it must have at least the following value, where:

If IPARM(2) = 0 or 2, use *naux1* ≥ 3*m*.

If IPARM(2) = 1 and IPARM(1) ≠ 0, use *naux1* ≥ 3*m*+2(IPARM(1)).

If IPARM(2) = 1 and IPARM(1) = 0, use *naux1* ≥ 3*m*+600.

aux2
is the storage work area used by this subroutine. If IPARM(3) = −10, *aux2* must contain the incomplete Cholesky factorization of matrix ***A***, computed in an earlier call to DSDCG. Its size is specified by *naux2*. Specified as: an area of storage, containing long-precision real numbers.

naux2
is the size of the work area specified by *aux2*—that is, the number of elements in *aux2*. Specified as: a fullword integer. When IPARM(3) = 10 or −10, *naux2* must have at least the following value, where:

If *iopt* = 0, use *naux2* ≥ *m*(1.5*nd*+2)1.5+2(*m*+6).

If *iopt* = 1, use *naux2* ≥ *m*(3*nd*+2)+8.

On Return

x
is the vector $\boldsymbol{x}$ of length *m*, containing the solution of the system $\boldsymbol{Ax} = \boldsymbol{b}$. Returned as: a one-dimensional array, containing long-precision real numbers.

iparm
As an array of parameters, IPARM(*i*), where:

IPARM(1) is unchanged.

IPARM(2) is unchanged.

IPARM(3) is unchanged.

IPARM(4) contains the number of iterations performed by this subroutine.

Returned as: a one-dimensional array of length 4, containing fullword integers.

rparm
is an array of parameters, RPARM(*i*), where:

RPARM(1) is unchanged.

RPARM(2) is unchanged if IPARM(2) = 0 or 2. If IPARM(2) = 1, RPARM(2) contains λ, an estimate of the smallest eigenvalue of the iteration matrix.

RPARM(3) contains the estimate of the error of the solution. If the process converged, RPARM(3) ≤ ϵ.

Returned as: a one-dimensional array of length 3, containing long-precision real numbers; λ > 0.

aux2
is the storage work area used by this subroutine.

If IPARM(3) = 10, *aux2* contains the incomplete Cholesky factorization of matrix $\boldsymbol{A}$.

If IPARM(3) = −10, *aux2* is unchanged.

See "Notes" for additional information on *aux2*. Returned as: an area of storage, containing long-precision real numbers.

Notes

1. When IPARM(3) = −10, this subroutine uses the incomplete Cholesky factorization in *aux2*, computed in an earlier call to this subroutine. When IPARM(3) = 10, this subroutine computes the incomplete Cholesky factorization and stores it in *aux2*.
2. If you solve the same sparse linear system of equations several times with different right-hand sides using the preconditioned algorithm, specify IPARM(3) = 10 on the first invocation. The incomplete factorization is stored in *aux2*. You may save computing time on subsequent calls by setting IPARM(3) = −10. In this way, the algorithm reutilizes the incomplete factorization that was computed the first time. Therefore, you should not modify the contents of *aux2* between calls.
3. Matrix $\boldsymbol{A}$ must have no common elements with vectors $\boldsymbol{x}$ and $\boldsymbol{b}$; otherwise, results are unpredictable.
4. In the iterative solvers for sparse matrices, the relative accuracy ε (RPARM(1)) must be specified "reasonably" (10^{-4} to 10^{-8}). The algorithm computes a sequence of approximate solution vectors $\boldsymbol{x}$ that converge to the solution. The iterative procedure is stopped when the norm of the residual is sufficiently small—that is, when:

$$\|\boldsymbol{b}-\boldsymbol{A}\boldsymbol{x}\|_2 \;/\; \lambda\|\boldsymbol{x}\|_2 \;<\; \varepsilon$$

where λ is an estimate of the minimum eigenvalue of the iteration matrix, which is either estimated adaptively or given by the user. As a result, if you specify a larger ε, the algorithm takes fewer iterations to converge to a solution. If you specify a smaller ε, the algorithm requires more iterations and computer time, but converges to a more precise solution. If the value you specify is unreasonably small, the algorithm may fail to converge within the number of iterations it is allowed to perform.

5. For a description of how sparse matrices are stored in compressed-matrix storage mode, see "Compressed-Matrix Storage Mode" on page 93.
6. On output, array AD and vector ***b*** are not bitwise identical to what they were on input, because the matrix ***A*** and the right-hand side are scaled before starting the iterative process and are unscaled before returning control to the user. In addition, arrays AD and LA may be rearranged on output, but still contain a mathematically equivalent mapping of the elements in matrix ***A***.
7. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: The sparse positive definite or negative definite linear system:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

is solved, where:

A is a symmetric, positive definite or negative definite sparse matrix of order *m*, stored in compressed-diagonal storage mode in arrays AD and LA.
x is a vector of length *m*.
b is a vector of length *m*.

The system is solved using the two-term conjugate gradient method, with or without preconditioning by an incomplete Cholesky factorization. In both cases, the matrix is scaled by the square root of the diagonal.

See references [59] and [62]. [36].

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux1* = 0, and unable to allocate work area.

Computational Errors: The following errors, with their corresponding return codes, can occur in this subroutine. Where a value of *i* is indicated, it can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for that particular error code in the ESSL error option table; otherwise, the default value causes your program to terminate when the error occurs. For details, see "What Can You Do about ESSL Computational Errors?" on page 48.

- For error 2110, return code 1 indicates that the subroutine exceeded IPARM(1) iterations without converging. Vector ***x*** contains the approximate solution computed at the last iteration.

- For error 2111, return code 2 indicates that *aux2* contains an incorrect factorization. The subroutine has been called with `IPARM(3) = −10`, and *aux2* contains an incomplete factorization of the input matrix ***A*** that was computed by a previous call to the subroutine when `IPARM(3) = 10`. This error indicates that *aux2* has been modified since the last call to the subroutine, or that the input matrix is not the same as the one that was factored. If the default action has been overridden, the subroutine can be called again with the same parameters, with the exception of `IPARM(3) = 0` or 10.
- For error 2109, return code 3 indicates that the inner product (***y***,***Ay***) is negative in the iterative procedure after iteration *i*. This should not occur, because the input matrix is assumed to be positive or negative definite. Vector ***x*** contains the results of the last iteration. The value *i* is identified in the computational error message.
- For error 2108, return code 4 indicates that the matrix is not positive definite. `AC` is partially modified and does not represent the same matrix as on entry.

Input-Argument Errors

1. *iopt* ≠ 0 or 1
2. *m* < 0
3. *lda* < 1
4. *lda* < *m*
5. *nd* < 0
6. *nd* = 0 and *m* > 0
7. $|\lambda(i)| > m-1$ for *i* = 1, *nd*
8. `IPARM(1)` < 0
9. `IPARM(2)` ≠ 0, 1, or 2
10. `IPARM(3)` ≠ 0, 10, or −10
11. `RPARM(1)` < 0
12. `RPARM(2)` < 0
13. Error 2015 is recoverable or *naux1*≠0, and *naux1* is too small—that is, less than the minimum required value. Return code 5 is returned if error 2015 is recoverable.
14. *naux2* is too small—that is, less than the minimum required value. Return code 5 is returned if error 2015 is recoverable.

Example 1: This example finds the solution of the linear system ***Ax*** = ***b*** for sparse matrix ***A***, which is stored in compressed-diagonal storage mode in arrays `AD` and `LA`. The system is solved using the two-term conjugate gradient method. In this example, `IOPT = 0`.. Matrix ***A*** is:

```
[                                                                  ]
|  2.0   0.0  -1.0   0.0   0.0   0.0   0.0   0.0   0.0 |
|  0.0   2.0   0.0  -1.0   0.0   0.0   0.0   0.0   0.0 |
| -1.0   0.0   2.0   0.0  -1.0   0.0   0.0   0.0   0.0 |
|  0.0  -1.0   0.0   2.0   0.0  -1.0   0.0   0.0   0.0 |
|  0.0   0.0  -1.0   0.0   2.0   0.0  -1.0   0.0   0.0 |
|  0.0   0.0   0.0  -1.0   0.0   2.0   0.0  -1.0   0.0 |
|  0.0   0.0   0.0   0.0  -1.0   0.0   2.0   0.0  -1.0 |
|  0.0   0.0   0.0   0.0   0.0  -1.0   0.0   2.0   0.0 |
|  0.0   0.0   0.0   0.0   0.0   0.0  -1.0   0.0   2.0 |
[                                                                  ]
```

Call Statement and Input

```
           IOPT  M   ND  AD  LDA  LA   B   X   IPARM   RPARM   AUX1  NAUX1  AUX2  NAUX2
            |    |   |   |    |   |    |   |     |       |      |      |     |      |
CALL DSDCG( 0  , 9 , 3 , AD , 9 , LA , B , X , IPARM , RPARM , AUX1 , 283 , AUX2 ,  0  )
```

```
IPARM(1) =  20
IPARM(2) =  0
IPARM(3) =  0
RPARM(1) =  1.D-7
```

$$AD = \begin{bmatrix} 2.0 & 0.0 & -1.0 \\ 2.0 & 0.0 & -1.0 \\ 2.0 & -1.0 & -1.0 \\ 2.0 & -1.0 & -1.0 \\ 2.0 & -1.0 & -1.0 \\ 2.0 & -1.0 & -1.0 \\ 2.0 & -1.0 & -1.0 \\ 2.0 & -1.0 & 0.0 \\ 2.0 & -1.0 & 0.0 \end{bmatrix}$$

```
LA       =  (0, -2, 2)
B        =  (1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0)
X        =  (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
```

Output

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(4) =  5
RPARM(2) =  0
RPARM(3) =  0.46D-16
```

Example 2: This example finds the solution of the linear system $\boldsymbol{Ax} = \boldsymbol{b}$ for the same sparse matrix $\boldsymbol{A}$ as in Example 1, which is stored in compressed-diagonal storage mode in arrays AD and LA. The system is solved using the two-term conjugate gradient method. In this example, IOPT = 1, indicating that the matrix is symmetric, and only the main diagonal and one of each pair of identical diagonals are stored in array AD.

Call Statement and Input

```
           IOPT  M   ND  AD  LDA  LA   B   X   IPARM   RPARM   AUX1  NAUX1  AUX2  NAUX2
            |    |   |   |    |   |    |   |     |       |      |      |     |      |
CALL DSDCG( 1  , 9 , 2 , AD , 9 , LA , B , X , IPARM , RPARM , AUX1 , 283 , AUX2 , 80  )
```

```
IPARM(1) =  20
IPARM(2) =  0
IPARM(3) =  10
RPARM(1) =  1.D-7
```

```
        ⎡           ⎤
        ⎢ 2.0   0.0 ⎥
        ⎢ 2.0   0.0 ⎥
        ⎢ 2.0  -1.0 ⎥
        ⎢ 2.0  -1.0 ⎥
AD   =  ⎢ 2.0  -1.0 ⎥
        ⎢ 2.0  -1.0 ⎥
        ⎢ 2.0  -1.0 ⎥
        ⎢ 2.0  -1.0 ⎥
        ⎣ 2.0  -1.0 ⎦

LA       =  (0, -2)
B        =  (1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0)
X        =  (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
```

Output

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(4) =  1
RPARM(2) =  0
RPARM(3) =  0.89D-16
```

DSMGCG—General Sparse Matrix Iterative Solve Using Compressed-Matrix Storage Mode

This subroutine solves a general sparse linear system of equations using an iterative algorithm, conjugate gradient squared or generalized minimum residual, with or without preconditioning by an incomplete LU factorization. The subroutine is suitable for positive real matrices—that is, when the symmetric part of the matrix, $(\boldsymbol{A}+\boldsymbol{A}^{\mathrm{T}})/2$, is positive definite. The sparse matrix is stored in compressed-matrix storage mode. Matrix $\boldsymbol{A}$ and vectors $\boldsymbol{x}$ and $\boldsymbol{b}$ are used:

$$\boldsymbol{Ax} = \boldsymbol{b}$$

where $\boldsymbol{A}$, $\boldsymbol{x}$, and $\boldsymbol{b}$ contain long-precision real numbers.

Notes:

1. These subroutines are provided only for migration purposes. You get better performance and a wider choice of algorithms if you use the DSRIS subroutine.
2. If your sparse matrix is stored by rows, as defined in "Storage-by-Rows" on page 99, you should first use the utility subroutine DSRSM to convert your sparse matrix to compressed-matrix storage mode. See "DSRSM—Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode" on page 986.

Syntax

Fortran	CALL DSMGCG (*m*, *nz*, *ac*, *ka*, *lda*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	dsmgcg (*m*, *nz*, *ac*, *ka*, *lda*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*);
PL/I	CALL DSMGCG (*m*, *nz*, *ac*, *ka*, *lda*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

m
is the order of the linear system $\boldsymbol{Ax} = \boldsymbol{b}$ and the number of rows in sparse matrix $\boldsymbol{A}$. Specified as: a fullword integer; $m \geq 0$.

nz
is the maximum number of nonzero elements in each row of sparse matrix $\boldsymbol{A}$. Specified as: a fullword integer; $nz \geq 0$.

ac
is the array, referred to as AC, containing the values of the nonzero elements of the sparse matrix, stored in compressed-matrix storage mode. Specified as: an *lda* by (at least) *nz* array, containing long-precision real numbers.

ka
is the array, referred to as KA, containing the column numbers of the matrix $\boldsymbol{A}$ elements stored in the corresponding positions in array AC. Specified as: an *lda* by (at least) *nz* array, containing fullword integers, where $1 \leq$ (elements of KA) $\leq m$.

lda
is the leading dimension of the arrays specified for *ac* and *ka*. Specified as: a fullword integer; $lda > 0$ and $lda \geq m$.

b
is the vector $\boldsymbol{b}$ of length *m*, containing the right-hand side of the matrix problem. Specified as: a one-dimensional array of (at least) length *m*, containing long-precision real numbers.

x
is the vector ***x*** of length *m*, containing your initial guess of the solution of the linear system. Specified as: a one-dimensional array of (at least) length *m*, containing long-precision real numbers. The elements can have any value, and if no guess is available, the value can be zero.

iparm
is an array of parameters, IPARM(*i*), where:

- IPARM(1) controls the number of iterations.

  If IPARM(1) > 0, IPARM(1) is the maximum number of iterations allowed.

  If IPARM(1) = 0, the following default values are used:

  IPARM(1) = 300
  IPARM(2) = 0
  IPARM(3) = 10
  RPARM(1) = 10^{-6}

- IPARM(2) is the flag used to select the iterative procedure used in this subroutine.

  If IPARM(2) = 0, the conjugate gradient squared method is used.

  If IPARM(2) = *k*, the generalized minimum residual method, restarted after *k* steps, is used. Note that the size of the work area *aux1* becomes larger as *k* increases. A value for *k* in the range of 5 to 10 is suitable for most problems.

- IPARM(3) is the flag that determines whether the system is to be preconditioned by an incomplete LU factorization with no fill-in.

  If IPARM(3) = 0, the system is not preconditioned.

  If IPARM(3) = 10, the system is preconditioned by an incomplete LU factorization.

  If IPARM(3) = −10, the system is preconditioned by an incomplete LU factorization, where the factorization matrix was computed in an earlier call to this subroutine and is stored in *aux2*.

- IPARM(4), see On Return.

Specified as: an array of (at least) length 4, containing fullword integers, where:
IPARM(1) ≥ 0
IPARM(2) ≥ 0
IPARM(3) = 0, 10, or −10

rparm
is an array of parameters, RPARM(*i*), where:

RPARM(1) > 0, is the relative accuracy ϵ used in the stopping criterion. The iterative procedure is stopped when:
$\|\boldsymbol{b}-\boldsymbol{A}\boldsymbol{x}\|_2 / \|\boldsymbol{x}\|_2 < \varepsilon$

RPARM(2) is reserved.

RPARM(3), see On Return.

Specified as: a one-dimensional array of (at least) length 3, containing long-precision real numbers.

aux1
has the following meaning:

If *naux1* = 0 and error 2015 is unrecoverable, *aux1* is ignored.

Otherwise, it is a storage work area used by this subroutine, which is available for use by the calling program between calls to this subroutine. Its size is specified by *naux1*.

Specified as: an area of storage, containing long-precision real numbers.

naux1
is the size of the work area specified by *aux1*—that is, the number of elements in *aux1*. Specified as: a fullword integer, where:

If *naux1* = 0 and error 2015 is unrecoverable, DSMGCG dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, it must have at least the following value, where:

If IPARM(2) = 0, use $naux1 \geq 7m$.

If IPARM(2) > 0, use $naux1 \geq (k+2)m+k(k+4)+1$, where k = IPARM(2).

aux2
is the storage work area used by this subroutine. If IPARM(3) = −10, *aux2* must contain the incomplete LU factorization of matrix $\boldsymbol{A}$, computed in an earlier call to DSMGCG. The size of *aux2* is specified by *naux2*.

Specified as: an area of storage, containing long-precision real numbers.

naux2
is the size of the work area specified by *aux2*—that is, the number of elements in *aux2*. Specified as: a fullword integer. When IPARM(3) = 10, *naux2* must have at least the following value: $naux2 \geq 3+2m+1.5nz(m)$.

On Return

x
is the vector $\boldsymbol{x}$ of length m, containing the solution of the system $\boldsymbol{Ax} = \boldsymbol{b}$. Returned as: a one-dimensional array of (at least) length m, containing long-precision real numbers.

iparm
is an array of parameters, IPARM(*i*), where:

IPARM(1) is unchanged.

IPARM(2) is unchanged.

IPARM(3) is unchanged.

IPARM(4) contains the number of iterations performed by this subroutine.

Returned as: a one-dimensional array of length 4, containing fullword integers.

rparm
is an array of parameters, RPARM(*i*), where:

RPARM(1) is unchanged.

RPARM(2) is reserved.

RPARM(3) contains the estimate of the error of the solution. If the process converged, RPARM(3) ≤ RPARM(1)

Returned as: a one-dimensional array of length 3, containing long-precision real numbers.

aux2
is the storage work area used by this subroutine.

If IPARM(3) = 10, *aux2* contains the incomplete LU factorization of matrix ***A***.

If IPARM(3) = −10, *aux2* is unchanged.

See "Notes" for additional information on *aux2*. Returned as: an area of storage, containing long-precision real numbers.

Notes

1. When IPARM(3) = −10, this subroutine uses the incomplete LU factorization in *aux2*, computed in an earlier call to this subroutine. When IPARM(3) = 10, this subroutine computes the incomplete LU factorization and stores it in *aux2*.
2. If you solve the same sparse linear system of equations several times with different right-hand sides using the preconditioned algorithm, specify IPARM(2) = 10 on the first invocation. The incomplete factorization is stored in *aux2*. You may save computing time on subsequent calls by setting IPARM(3) equal to −10. In this way, the algorithm reutilizes the incomplete factorization that was computed the first time. Therefore, you should not modify the contents of *aux2* between calls.
3. Matrix ***A*** must have no common elements with vectors ***x*** and ***b***; otherwise, results are unpredictable.
4. In the iterative solvers for sparse matrices, the relative accuracy ε (RPARM(1)) must be specified "reasonably" (10^{-4} to 10^{-8}). The algorithm computes a sequence of approximate solution vectors ***x*** that converge to the solution. The iterative procedure is stopped when the norm of the residual is sufficiently small—that is, when:

   $$\|\boldsymbol{b}-\boldsymbol{A}\boldsymbol{x}\|_2 \;/\; \|\boldsymbol{x}\|_2 \;<\; \varepsilon$$

   As a result, if you specify a larger ε, the algorithm takes fewer iterations to converge to a solution. If you specify a smaller ε, the algorithm requires more iterations and computer time, but converges to a more precise solution. If the value you specify is unreasonably small, the algorithm may fail to converge within the number of iterations it is allowed to perform.
5. For a description of how sparse matrices are stored in compressed-matrix storage mode, see "Compressed-Matrix Storage Mode" on page 93.
6. On output, array AC is not bitwise identical to what it was on input because the matrix ***A*** is scaled before starting the iterative process and is unscaled before returning control to the user.
7. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: The linear system:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

is solved using either the conjugate gradient squared method or the generalized minimum residual method, with or without preconditioning by an incomplete LU factorization, where:

A is a sparse matrix of order *m*, stored in compressed-matrix storage mode in arrays AC and KA.
x is a vector of length *m*.
b is a vector of length *m*.

See references [80] and [82]. [36].

If your program uses a sparse matrix stored by rows and you want to use this subroutine, first convert your sparse matrix to compressed-matrix storage mode by using the subroutine DSRSM described on page 986.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux1* = 0, and unable to allocate work area.

Computational Errors: The following errors, with their corresponding return codes, can occur in this subroutine. For details on error handling, see "What Can You Do about ESSL Computational Errors?" on page 48.

- For error 2110, return code 1 indicates that the subroutine exceeded IPARM(1) iterations without converging. Vector ***x*** contains the approximate solution computed at the last iteration.
- For error 2111, return code 2 indicates that *aux2* contains an incorrect factorization. The subroutine has been called with IPARM(3) = −10, and *aux2* contains an incomplete factorization of the input matrix ***A*** that was computed by a previous call to the subroutine when IPARM(3) = 10. This error indicates that *aux2* has been modified since the last call to the subroutine, or that the input matrix is not the same as the one that was factored. If the default action has been overridden, the subroutine can be called again with the same parameters, with the exception of IPARM(3) = 0 or 10.
- For error 2112, return code 3 indicates that the incomplete LU factorization of ***A*** could not be completed, because one pivot was 0.
- For error 2116, return code 4 indicates that the matrix is singular, because all elements in one row of the matrix contain 0. Array AC is partially modified and does not represent the same matrix as on entry.

Input-Argument Errors

1. *m* < 0
2. *lda* < 1
3. *lda* < *m*
4. *nz* < 0
5. *nz* = 0 and *m* > 0
6. IPARM(1) < 0
7. IPARM(2) < 0
8. IPARM(3) ≠ 0, 10, or −10
9. RPARM(1) < 0
10. RPARM(2) < 0
11. Error 2015 is recoverable or *naux1*≠0, and *naux1* is too small—that is, less than the minimum required value. Return code 5 is returned if error 2015 is recoverable.
12. *naux2* is too small—that is, less than the minimum required value. Return code 5 is returned if error 2015 is recoverable.

Example 1: This example finds the solution of the linear system $\boldsymbol{Ax} = \boldsymbol{b}$ for the sparse matrix $\boldsymbol{A}$, which is stored in compressed-matrix storage mode in arrays AC and KA. The system is solved using the conjugate gradient squared method. Matrix $\boldsymbol{A}$ is:

```
⎡                                                                  ⎤
⎢ 2.0  0.0   0.0  0.0   0.0   0.0   0.0   0.0   0.0 ⎥
⎢ 0.0  2.0  -1.0  0.0   0.0   0.0   0.0   0.0   0.0 ⎥
⎢ 0.0  1.0   2.0  0.0   0.0   0.0   0.0   0.0   0.0 ⎥
⎢ 1.0  0.0   0.0  2.0  -1.0   0.0   0.0   0.0   0.0 ⎥
⎢ 0.0  0.0   0.0  1.0   2.0  -1.0   0.0   0.0   0.0 ⎥
⎢ 0.0  0.0   0.0  0.0   1.0   2.0  -1.0   0.0   0.0 ⎥
⎢ 0.0  0.0   0.0  0.0   0.0   1.0   2.0  -1.0   0.0 ⎥
⎢ 0.0  0.0   0.0  0.0   0.0   0.0   1.0   2.0  -1.0 ⎥
⎢ 0.0  0.0   0.0  0.0   0.0   0.0   0.0   1.0   2.0 ⎥
⎣                                                                  ⎦
```

Note: For input matrix KA, (.) indicates any value between 1 and 9.

Call Statement and Input

```
             M   NZ   AC   KA   LDA   B   X   IPARM   RPARM   AUX1   NAUX1   AUX2   NAUX2
             |    |    |    |    |    |   |     |       |      |       |      |       |
CALL DSMGCG( 9 ,  3 , AC , KA ,  9  , B , X , IPARM , RPARM , AUX1 ,  63   , AUX2 ,   0 )
```

```
IPARM(1) =  20
IPARM(2) =  0
IPARM(3) =  0
RPARM(1) =  1.D-7
```

```
         ⎡                  ⎤
         ⎢ 2.0   0.0   0.0 ⎥
         ⎢ 2.0  -1.0   0.0 ⎥
         ⎢ 1.0   2.0   0.0 ⎥
         ⎢ 1.0   2.0  -1.0 ⎥
AC   =   ⎢ 1.0   2.0  -1.0 ⎥
         ⎢ 1.0   2.0  -1.0 ⎥
         ⎢ 1.0   2.0  -1.0 ⎥
         ⎢ 1.0   2.0  -1.0 ⎥
         ⎢ 1.0   2.0   0.0 ⎥
         ⎣                  ⎦
```

```
         ⎡         ⎤
         ⎢ 1  .  . ⎥
         ⎢ 2  3  . ⎥
         ⎢ 2  3  . ⎥
         ⎢ 1  4  5 ⎥
KA   =   ⎢ 4  5  6 ⎥
         ⎢ 5  6  7 ⎥
         ⎢ 6  7  8 ⎥
         ⎢ 7  8  9 ⎥
         ⎢ 8  9  . ⎥
         ⎣         ⎦
```

```
B        =  (2.0, 1.0, 3.0, 2.0, 2.0, 2.0, 2.0, 2.0, 3.0)
X        =  (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
```

Output

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(4) =  9
RPARM(3) =  0.150D-19
```

Example 2: This example finds the solution of the linear system $\boldsymbol{Ax} = \boldsymbol{b}$ for the same sparse matrix $\boldsymbol{A}$ as in Example 1, which is stored in compressed-matrix storage mode in arrays AC and KA. The system is solved using the generalized minimum residual method, restarted after 5 steps and preconditioned with an incomplete LU factorization. Most of the input is the same as in Example 1.

Note: For input matrix KA, (.) indicates any value between 1 and 9.

Call Statement and Input

```
             M   NZ   AC   KA   LDA   B   X   IPARM   RPARM   AUX1   NAUX1   AUX2   NAUX2
             |    |    |    |    |    |   |     |       |      |       |      |       |
CALL DSMGCG( 9 ,  3 , AC , KA ,  9  , B , X , IPARM , RPARM , AUX1 ,  109  , AUX2 ,  46 )
```

```
IPARM(1) =  20
IPARM(2) =  5
IPARM(3) =  10
RPARM(1) =  1.D-7
AC       =(same as input AC in Example 1)
KA       =(same as input KA in Example 1)
B        =  (2.0, 1.0, 3.0, 2.0, 2.0, 2.0, 2.0, 2.0, 3.0)
X        =  (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
```

Output

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(4) =  2
RPARM(3) =  0.290D-15
```

DSDGCG—General Sparse Matrix Iterative Solve Using Compressed-Diagonal Storage Mode

This subroutine solves a general sparse linear system of equations using an iterative algorithm, conjugate gradient squared or generalized minimum residual, with or without preconditioning by an incomplete LU factorization. The subroutine is suitable for positive real matrices—that is, when the symmetric part of the matrix, $(\boldsymbol{A}+\boldsymbol{A}^{T})/2$, is positive definite. The sparse matrix is stored in compressed-diagonal storage mode. Matrix $\boldsymbol{A}$ and vectors $\boldsymbol{x}$ and $\boldsymbol{b}$ are used:

$$\boldsymbol{Ax} = \boldsymbol{b}$$

where $\boldsymbol{A}$, $\boldsymbol{x}$, and $\boldsymbol{b}$ contain long-precision real numbers.

Syntax

Fortran	CALL DSDGCG (*m*, *nd*, *ad*, *lda*, *la*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	dsdgcg (*m*, *nd*, *ad*, *lda*, *la*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*);
PL/I	CALL DSDGCG (*m*, *nd*, *ad*, *lda*, *la*, *b*, *x*, *iparm*, *rparm*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

m
is the order of the linear system $\boldsymbol{Ax} = \boldsymbol{b}$ and the number of rows in sparse matrix $\boldsymbol{A}$. Specified as: a fullword integer; $m \geq 0$.

nd
is the number of nonzero diagonals stored in the columns of array AD, the number of columns in array AD, and the number of elements in array LA. Specified as: a fullword integer; it must have the following value, where:

If $m > 0$, then $nd > 0$.

If $m = 0$, then $nd \geq 0$.

ad
is the array, referred to as AD, containing the values of the nonzero elements of the sparse matrix, stored in compressed-matrix storage mode. Specified as: an *lda* by (at least) *nd* array, containing long-precision real numbers.

lda
is the leading dimension of the arrays specified for *ad*. Specified as: a fullword integer; $lda > 0$ and $lda \geq m$.

la
is the array, referred to as LA, containing the diagonal numbers *k* for the diagonals stored in each corresponding column in array AD. For an explanation of how diagonal numbers are stored, see "Compressed-Diagonal Storage Mode" on page 94.

Specified as: a one-dimensional array of (at least) length *nd*, containing fullword integers, where $1-m \leq$ (elements of LA) $\leq (m-1)$.

b
is the vector $\boldsymbol{b}$ of length *m*, containing the right-hand side of the matrix problem. Specified as: a one-dimensional array of (at least) length *m*, containing long-precision real numbers.

x
is the vector $\boldsymbol{x}$ of length *m*, containing your initial guess of the solution of the linear system. Specified as: a one-dimensional array of (at least) length *m*, containing

long-precision real numbers. The elements can have any value, and if no guess is available, the value can be zero.

iparm
is an array of parameters, IPARM(*i*), where:

- IPARM(1) controls the number of iterations.

  If IPARM(1) > 0, IPARM(1) is the maximum number of iterations allowed.

  If IPARM(1) = 0, the following default values are used:

  IPARM(1) = 300
  IPARM(2) = 0
  IPARM(3) = 10
  RPARM(1) = 10^{-6}

- IPARM(2) is the flag used to select the iterative procedure used in this subroutine.

  If IPARM(2) = 0, the conjugate gradient squared method is used.

  If IPARM(2) = *k*, the generalized minimum residual method, restarted after *k* steps, is used. Note that the size of the work area *aux1* becomes larger as *k* increases. A value for *k* in the range of 5 to 10 is suitable for most problems.

- IPARM(3) is the flag that determines whether the system is to be preconditioned by an incomplete LU factorization with no fill-in.

  If IPARM(3) = 0, the system is not preconditioned.

  If IPARM(3) = 10, the system is preconditioned by an incomplete LU factorization.

  If IPARM(3) = −10, the system is preconditioned by an incomplete LU factorization, where the factorization matrix was computed in an earlier call to this subroutine and is stored in *aux2*.

- IPARM(4), see On Return.

Specified as: an array of (at least) length 4, containing fullword integers, where:
IPARM(1) ≥ 0
IPARM(2) ≥ 0
IPARM(3) = 0, 10, or −10

rparm
is an array of parameters, RPARM(*i*), where:

If RPARM(1) > 0, is the relative accuracy ϵ used in the stopping criterion. The iterative procedure is stopped when:

$$\|\boldsymbol{b}-\boldsymbol{A}\boldsymbol{x}\|_2 \,/\, \|\boldsymbol{x}\|_2 < \varepsilon$$

RPARM(2) is reserved.

RPARM(3), see On Return.

Specified as: a one-dimensional array of (at least) length 3, containing long-precision real numbers.

aux1
has the following meaning:

If *naux1* = 0 and error 2015 is unrecoverable, *aux1* is ignored.

Otherwise, it is a storage work area used by this subroutine, which is available for use by the calling program between calls to this subroutine. Its size is specified by *naux1*.

Specified as: an area of storage, containing long-precision real numbers.

naux1
is the size of the work area specified by *aux1*—that is, the number of elements in *aux1*. Specified as: a fullword integer, where:

If *naux1* = 0 and error 2015 is unrecoverable, DSDGCG dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux1* > 0 and must have at least the following value, where:

If IPARM(2) = 0, use *naux1* ≥ 7*m*.

If IPARM(2) > 0, use *naux1* ≥ (*k*+2)*m*+*k*(*k*+4)+1, where *k* = PARM(2).

aux2
is a storage work area used by this subroutine. If IPARM(3) = −10, *aux2* must contain the incomplete LU factorization of matrix ***A***, computed in an earlier call to DSDGCG. The size of *aux2* is specified by *naux2*.

Specified as: an area of storage, containing long-precision real numbers.

naux2
is the size of the work area specified by *aux2*—that is, the number of elements in *aux2*. Specified as: a fullword integer. When IPARM(3) = 10 or −10, *naux2* must have at least the following value: *naux2* ≥ 3+2*m*+1.5*nd*(*m*).

On Return

x
is the vector ***x*** of length *m*, containing the solution of the system ***Ax*** = ***b***. Returned as: a one-dimensional array of (at least) length *m*, containing long-precision real numbers.

iparm
is an array of parameters, IPARM(*i*), where:

IPARM(1) is unchanged.

IPARM(2) is unchanged.

IPARM(3) is unchanged.

IPARM(4) contains the number of iterations performed by this subroutine.

Returned as: a one-dimensional array of length 4, containing fullword integers.

rparm
is an array of parameters, RPARM(*i*), where:

RPARM(1) is unchanged.

RPARM(2) is reserved.

RPARM(3) contains the estimate of the error of the solution. If the process converged, RPARM(3) ≤ RPARM(1).

Returned as: a one-dimensional array of length 3, containing long-precision real numbers.

aux2
is the storage work area used by this subroutine.

If IPARM(3) = 10, *aux2* contains the incomplete LU factorization of matrix ***A***.

If IPARM(3) = −10, *aux2* is unchanged.

See "Notes" for additional information on *aux2*. Returned as: an area of storage, containing long-precision real numbers.

Notes

1. When IPARM(3) = −10, this subroutine uses the incomplete LU factorization in *aux2*, computed in an earlier call to this subroutine. When IPARM(3) = 10, this subroutine computes the incomplete LU factorization and stores it in *aux2*.
2. If you solve the same sparse linear system of equations several times with different right-hand sides, using the preconditioned algorithm, specify IPARM(3) = 10 on the first invocation. The incomplete factorization is stored in *aux2*. You may save computing time on subsequent calls by setting IPARM(3) = −10. In this way, the algorithm reutilizes the incomplete factorization that was computed the first time. Therefore, you should not modify the contents of *aux2* between calls.
3. Matrix ***A*** must have no common elements with vectors ***x*** and ***b***; otherwise, results are unpredictable.
4. In the iterative solvers for sparse matrices, the relative accuracy ε (RPARM(1)) must be specified "reasonably" (10^{-4} to 10^{-8}). The algorithm computes a sequence of approximate solution vectors ***x*** that converge to the solution. The iterative procedure is stopped when the norm of the residual is sufficiently small—that is, when:

   $$\|\boldsymbol{b}-\boldsymbol{A}\boldsymbol{x}\|_2 \;/\; \|\boldsymbol{x}\|_2 \;<\; \varepsilon$$

   As a result, if you specify a larger ε, the algorithm takes fewer iterations to converge to a solution. If you specify a smaller ε, the algorithm requires more iterations and computer time, but converges to a more precise solution. If the value you specify is unreasonably small, the algorithm may fail to converge within the number of iterations it is allowed to perform.
5. For a description of how sparse matrices are stored in compressed-diagonal storage mode, see "Compressed-Diagonal Storage Mode" on page 94.
6. On output, array AD is not bitwise identical to what it was on input, because matrix ***A*** is scaled before starting the iterative process and is unscaled before returning control to the user.
7. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: The linear system:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

is solved using either the conjugate gradient squared method or the generalized minimum residual method, with or without preconditioning by an incomplete LU factorization, where:

A is a sparse matrix of order *m*, stored in compressed-diagonal storage mode in arrays AD and LA.

x is a vector of length *m*.
b is a vector of length *m*.

See references [80] and [82]. [36].

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux1* = 0, and unable to allocate work area.

Computational Errors: The following errors, with their corresponding return codes, can occur in this subroutine. For details on error handling, see "What Can You Do about ESSL Computational Errors?" on page 48.

- For error 2110, return code 1 indicates that the subroutine exceeded IPARM(1) iterations without converging. Vector ***x*** contains the approximate solution computed at the last iteration.
- For error 2111, return code 2 indicates that *aux2* contains an incorrect factorization. The subroutine has been called with IPARM(3) = −10, and *aux2* contains an incomplete factorization of the input matrix ***A*** that was computed by a previous call to the subroutine when IPARM(3) = 10. This error indicates that *aux2* has been modified since the last call to the subroutine, or that the input matrix is not the same as the one that was factored. If the default action has been overridden, the subroutine can be called again with the same parameters, with the exception of IPARM(3) = 0 or 10.
- For error 2112, return code 3 indicates that the incomplete LU factorization of ***A*** could not be completed, because one pivot was 0.
- For error 2116, return code 4 indicates that the matrix is singular, because all elements in one row of the matrix contain 0. Array AC is partially modified and does not represent the same matrix as on entry.

Input-Argument Errors

1. *m* < 0
2. *lda* < 1
3. *lda* < *m*
4. *nd* < 0
5. *nd* = 0 and *m* > 0
6. IPARM(1) < 0
7. IPARM(2) < 0
8. IPARM(3) ≠ 0, 10, or −10
9. RPARM(1) < 1.D0
10. Error 2015 is recoverable or *naux1*≠0, and *naux1* is too small—that is, less than the minimum required value. Return code 5 is returned if error 2015 is recoverable.
11. *naux2* is too small—that is, less than the minimum required value. Return code 5 is returned if error 2015 is recoverable.

Example 1: This example finds the solution of the linear system ***Ax*** = ***b*** for the sparse matrix ***A***, which is stored in compressed-diagonal storage mode in arrays AD and LA. The system is solved using the conjugate gradient squared method. Matrix ***A*** is:

$$\begin{bmatrix} 2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 2.0 & 0.0 & -1.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 2.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 2.0 \end{bmatrix}$$

Call Statement and Input

```
          M   ND   AD   LDA   LA   B   X   IPARM   RPARM   AUX1   NAUX1   AUX2   NAUX2
          |    |    |    |     |   |   |     |       |      |       |      |      |
DSDGCG( 9 ,  3 , AD ,  9  , LA , B , X , IPARM , RPARM , AUX1 ,  63   , AUX2 ,   0 )
```

```
IPARM(1) =  20
IPARM(2) =  0
IPARM(3) =  0
RPARM(1) =  1.D-7
```

$$AD = \begin{bmatrix} 2.0 & -1.0 & 0.0 \\ 2.0 & -1.0 & 0.0 \\ 2.0 & -1.0 & 0.0 \\ 2.0 & -1.0 & 0.0 \\ 2.0 & -1.0 & 1.0 \\ 2.0 & -1.0 & 1.0 \\ 2.0 & -1.0 & 1.0 \\ 2.0 & 0.0 & 1.0 \\ 2.0 & 0.0 & 1.0 \end{bmatrix}$$

```
LA       =  (0, 2, -4)
B        =  (1, 1, 1, 1, 2, 2, 2, 3, 3)
X        =  (0, 0, 0, 0, 0, 0, 0, 0, 0)
```

Output

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(4) =  8
RPARM(3) =  0.308D-17
```

Example 2: This example finds the solution of the linear system $\boldsymbol{Ax} = \boldsymbol{b}$ for the same sparse matrix $\boldsymbol{A}$ as in Example 1, which is stored in compressed-diagonal storage mode in arrays AD and LA. The system is solved using the generalized minimum residual method, restarted after 5 steps and preconditioned with an incomplete LU factorization. Most of the input is the same as in Example 1.

Call Statement and Input

```
          M   ND   AD   LDA   LA   B   X   IPARM   RPARM   AUX1   NAUX1   AUX2   NAUX2
          |    |    |    |     |   |   |     |       |      |       |      |      |
CALL DSDGCG( 9 ,  3 , AD ,  9  , LA , B , X , IPARM , RPARM , AUX1 ,  109  , AUX2 ,  46 )
```

```
IPARM(1) =  20
IPARM(2) =  5
IPARM(3) =  10
RPARM(1) =  1.D-7
AD       =(same as input AD in Example 1)
LA       =(same as input LA in Example 1)
B        =  (1, 1, 1, 1, 2, 2, 2, 3, 3)
X        =  (0, 0, 0, 0, 0, 0, 0, 0, 0)
```

Output

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(4) =  6
RPARM(3) =  0.250D-15
```

Linear Least Squares Subroutines

This section contains the linear least squares subroutine descriptions.

SGESVF and DGESVF—Singular Value Decomposition for a General Matrix

These subroutines compute the singular value decomposition of general matrix ***A*** in preparation for solving linear least squares problems. To compute the minimal norm linear least squares solution of ***AX*** ≅ ***B***, follow the call to these subroutines with a call to SGESVS or DGESVS, respectively.

Table 114. Data Types

A, ***B***, *s*, *aux*	**Subroutine**
Short-precision real	SGESVF
Long-precision real	DGESVF

Syntax

Fortran	CALL SGESVF | DGESVF (*iopt*, *a*, *lda*, *b*, *ldb*, *nb*, *s*, *m*, *n*, *aux*, *naux*)
C and C++	sgesvf | dgesvf (*iopt*, *a*, *lda*, *b*, *ldb*, *nb*, *s*, *m*, *n*, *aux*, *naux*);
PL/I	CALL SGESVF | DGESVF (*iopt*, *a*, *lda*, *b*, *ldb*, *nb*, *s*, *m*, *n*, *aux*, *naux*);

On Entry

iopt

indicates the type of computation to be performed, where:

If *iopt* = 0 or 10, singular values are computed.

If *iopt* = 1 or 11, singular values and ***V*** are computed.

If *iopt* = 2 or 12, singular values, ***V***, and $\boldsymbol{U}^{T}\boldsymbol{B}$ are computed.

Specified as: a fullword integer; *iopt* = 0, 1, 2, 10, 11, or 12.

If *iopt* < 10, singular values are unordered.

If *iopt* ≥ 10, singular values are sorted in descending order and, if applicable, the columns of ***V*** and the rows of $\boldsymbol{U}^{T}\boldsymbol{B}$ are swapped to correspond to the sorted singular values.

a

is the *m* by *n* general matrix ***A***, whose singular value decomposition is to be computed. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 114.

lda

is the leading dimension of the array specified for *a*. Specified as: a fullword integer; *lda* > 0 and *lda* ≥ max(*m*, *n*).

b

has the following meaning, where:

If *iopt* = 0, 1, 10, or 11, this argument is not used in the computation.

If *iopt* = 2 or 12, it is the *m* by *nb* matrix ***B***.

Specified as: an *ldb* by (at least) *nb* array, containing numbers of the data type indicated in Table 114.

If this subroutine is followed by a call to SGESVS or DGESVS, ***B*** should contain the right-hand side of the linear least squares problem, $\boldsymbol{AX} \cong \boldsymbol{B}$. (The *nb* column vectors of ***B*** contain right-hand sides for *nb* distinct linear least squares problems.) However, if the matrix $\boldsymbol{U}^T$ is desired on output, ***B*** should be equal to the identity matrix of order *m*.

ldb

has the following meaning, where:

If *iopt* = 0, 1, 10, or 11, this argument is not used in the computation.

If *iopt* = 2 or 12, it is the leading dimension of the array specified for *b*.

Specified as: a fullword integer. It must have the following values, where:

If *iopt* = 0, 1, 10, or 11, $ldb > 0$.

If *iopt* = 2 or 12, $ldb > 0$ and $ldb \geq \max(m, n)$.

nb

has the following meaning, where:

If *iopt* = 0, 1, 10, or 11, this argument is not used in the computation.

If *iopt* = 2 or 12, it is the number of columns in matrix ***B***.

Specified as: a fullword integer; if *iopt* = 2 or 12, $nb > 0$.

s

See On Return.

m

is the number of rows in matrices ***A*** and ***B***. Specified as: a fullword integer; $m \geq 0$.

n

is the number of columns in matrix ***A*** and the number of elements in vector ***s***. Specified as: a fullword integer; $n \geq 0$.

aux

has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is the storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing numbers of the data type indicated in Table 114 on page 676.

naux

is the size of the work area specified by *aux*—that is, the number of elements in *aux*. Specified as: a fullword integer, where:

If *naux* = 0 and error 2015 is unrecoverable, SGESVF and DGESVF dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, It must have the following value, where:

If *iopt* = 0 or 10, $naux \geq n+\max(m, n)$.

If *iopt* = 1 or 11, $naux \geq 2n+\max(m, n)$.

If *iopt* = 2 or 12, $naux \geq 2n+\max(m, n, nb)$.

On Return

a

has the following meaning, where:

If *iopt* = 0, or 10, ***A*** is overwritten; that is, the original input is not preserved.

If *iopt* = 1, 2, 11, or 12, ***A*** contains the real orthogonal matrix ***V***, of order *n*, in its first *n* rows and *n* columns. If *iopt* = 11 or 12, the columns of ***V*** are swapped to correspond to the sorted singular values. If *m* > *n*, rows *n*+1, *n*+2, ..., *m* of array A are overwritten; that is, the original input is not preserved.

Returned as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 114 on page 676.

b

has the following meaning, where:

If *iopt* = 0, 1, 10, or 11, ***B*** is not used in the computation.

If *iopt* = 2 or 12, ***B*** is overwritten by the *n* by *nb* matrix $\boldsymbol{U}^T\boldsymbol{B}$.

If *iopt* = 12, the rows of $\boldsymbol{U}^T\boldsymbol{B}$ are swapped to correspond to the sorted singular values. If *m* > *n*, rows *n*+1, *n*+2, ..., *m* of array B are overwritten; that is, the original input is not preserved.

Returned as: an *ldb* by (at least) *nb* array, containing numbers of the data type indicated in Table 114 on page 676.

s

is a the vector ***s*** of length *n*, containing the singular values of matrix ***A***. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 114 on page 676; $s_i \geq 0$, where:

If *iopt* < 10, the singular values are unordered in ***s***.

If *iopt* ≥ 10, the singular values are sorted in descending order in ***s***; that is, $s_1 \geq s_2 \geq ... \geq s_n \geq 0$. If applicable, the columns of ***V*** and the rows of $\boldsymbol{U}^T\boldsymbol{B}$ are swapped to correspond to the sorted singular values.

Notes

1. The following items must have no common elements; otherwise, results are unpredictable: matrices ***A*** and ***B***, vector ***s***, and the data area specified for *aux*.
2. When you specify *iopt* = 0, 1, 10, or 11, you must also specify:
   - A dummy argument for *b*
   - A positive value for *ldb*

   See “Example” on page 564.
3. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see “Using Auxiliary Storage in ESSL” on page 31.

Function: The singular value decomposition of a real general matrix is computed as follows:

$$\boldsymbol{A} = \boldsymbol{U}\Sigma\boldsymbol{V}^T$$

where:

$$\boldsymbol{U}^T\boldsymbol{U} = \boldsymbol{V}^T\boldsymbol{V} = \boldsymbol{V}\boldsymbol{V}^T = \boldsymbol{I}$$

A is an *m* by *n* real general matrix.
V is a real general orthogonal matrix of order *n*. On output, ***V*** overwrites the first *n* rows and *n* columns of ***A***.
$\boldsymbol{U}^{\mathrm{T}}\boldsymbol{B}$ is an *n* by *nb* real general matrix. On output, $\boldsymbol{U}^{\mathrm{T}}\boldsymbol{B}$ overwrites the first *n* rows and *nb* columns of ***B***.
Σ is an *n* by *n* real diagonal matrix. The diagonal elements of Σ are the singular values of ***A***, returned in the vector ***s***.

If *m* or *n* is equal to 0, no computation is performed.

One of the following algorithms is used:

1. Golub-Reinsch Algorithm (See pages 134 to 151 in reference [93].)
   a. Reduce the real general matrix ***A*** to bidiagonal form using Householder transformations.
   b. Iteratively reduce the bidiagonal form to diagonal form using a variant of the QR algorithm.
2. Chan Algorithm (See reference [13].)
   a. Compute the QR decomposition of matrix ***A*** using Householder transformations; that is, $\boldsymbol{A} = \boldsymbol{QR}$.
   b. Apply the Golub-Reinsch Algorithm to the matrix ***R***.

      If $\boldsymbol{R} = \boldsymbol{XWY}^{\mathrm{T}}$ is the singular value decomposition of ***R***, the singular value decomposition of matrix ***A*** is given by:

$$\boldsymbol{A} = \boldsymbol{Q}\begin{bmatrix}\boldsymbol{X}\\0\end{bmatrix}\boldsymbol{WY}^{\mathrm{T}}$$

      where:

$$\boldsymbol{U} = \boldsymbol{Q}\begin{bmatrix}\boldsymbol{X}\\0\end{bmatrix}$$

$$\Sigma = \boldsymbol{W}$$

$$\boldsymbol{V} = \boldsymbol{Y}$$

Also, see references [13], [55], [72], and pages 134 to 151 in reference [93]. These algorithms have a tendency to generate underflows that may hurt overall performance. The system default is to mask underflow, which improves the performance of these subroutines.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors: Singular value (*i*) failed to converge after (*x*) iterations.

- The singular values (s_j, $j = n, n-1, ..., i+1$) are correct. If $iopt < 10$, they are unordered. Otherwise, they are ordered.
- *a* has been modified.
- If $iopt = 2$ or 12, then *b* has been modified.
- The return code is set to 1.
- *i* and *x* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2107 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. See "What Can You Do about ESSL Computational Errors?" on page 48.

Input-Argument Errors

1. $iopt \neq 0, 1, 2, 10, 11,$ or 12
2. $lda \leq 0$
3. $\max(m, n) > lda$
4. $ldb \leq 0$ and $iopt = 2, 12$
5. $\max(m, n) > ldb$ and $iopt = 2, 12$
6. $nb \leq 0$ and $iopt = 2, 12$
7. $m < 0$
8. $n < 0$
9. Error 2015 is recoverable or $naux \neq 0$, and *naux* is too small—that is, less than the minimum required value. Return code 2 is returned if error 2015 is recoverable.

Example 1: This example shows how to find only the singular values, ***s***, of a real long-precision general matrix ***A***, where:

- M is greater than N.
- NAUX is greater than or equal to N+max(M, N) = 7.
- LDB has been set to 1 to avoid a Fortran error message.
- DUMMY is a placeholder for argument *b*, which is not used in the computation.
- The singular values are returned in S.
- On output, matrix ***A*** is overwritten; that is, the original input is not preserved.

Call Statement and Input

```
            IOPT  A  LDA    B    LDB  NB  S   M   N   AUX  NAUX
             |    |   |     |     |   |   |   |   |    |    |
CALL DGESVF( 0  , A , 4 , DUMMY , 1 , 0 , S , 4 , 3 , AUX , 7  )
```

$$A = \begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \\ 7.0 & 8.0 & 9.0 \\ 10.0 & 11.0 & 12.0 \end{bmatrix}$$

Output

```
S        =   (25.462, 1.291, 0.000)
```

Example 2: This example computes the singular values, ***s***, of a real long-precision general matrix ***A*** and the matrix ***V***, where:

- M is equal to N.

- NAUX is greater than or equal to 2N+max(M, N) = 9.
- LDB has been set to 1 to avoid a Fortran error message.
- DUMMY is a placeholder for argument *b*, which is not used in the computation.
- The singular values are returned in S.
- The matrix ***V*** is returned in A.

Call Statement and Input

```
            IOPT  A  LDA    B    LDB  NB  S   M   N   AUX  NAUX
             |    |   |     |     |   |   |   |   |    |    |
CALL DGESVF( 1  , A , 3 , DUMMY , 1 , 0 , S , 3 , 3 , AUX , 9  )
```

$$A = \begin{bmatrix} 2.0 & 1.0 & 1.0 \\ 4.0 & 1.0 & 0.0 \\ -2.0 & 2.0 & 1.0 \end{bmatrix}$$

Output

$$A = \begin{bmatrix} -0.994 & 0.105 & -0.041 \\ -0.112 & -0.870 & 0.480 \\ -0.015 & -0.482 & -0.876 \end{bmatrix}$$

```
S        =    (4.922, 2.724, 0.597)
```

Example 3: This example computes the singular values, ***s***, and computes matrices ***V*** and $\boldsymbol{U}^{T}\boldsymbol{B}$ in preparation for solving the underdetermined system $\boldsymbol{AX} \cong \boldsymbol{B}$, where:

- M is less than N.
- NAUX is greater than or equal to 2N+max(M, N, NB) = 9.
- The singular values are returned in S.
- The matrix ***V*** is returned in A.
- The matrix $\boldsymbol{U}^{T}\boldsymbol{B}$ is returned in B.

Call Statement and Input

```
            IOPT  A  LDA  B  LDB  NB  S   M   N   AUX  NAUX
             |    |   |   |   |   |   |   |   |    |    |
CALL DGESVF( 2  , A , 3 , B , 3 , 1 , S , 2 , 3 , AUX , 9  )
```

$$A = \begin{bmatrix} 1.0 & 2.0 & 2.0 \\ 2.0 & 4.0 & 5.0 \\ . & . & . \end{bmatrix}$$

$$B = \begin{bmatrix} 1.0 \\ 4.0 \\ . \end{bmatrix}$$

Output

```
        ┌                      ┐
        │ -0.304  -0.894   0.328 │
A    =  │ -0.608   0.447   0.656 │
        │ -0.733   0.000  -0.680 │
        └                      ┘

        ┌        ┐
        │ -4.061 │
B    =  │  0.000 │
        │ -0.714 │
        └        ┘

S       =   (7.342, 0.000, 0.305)
```

Example 4: This example computes the singular values, ***s***, and matrices ***V*** and $\boldsymbol{U}^{T}\boldsymbol{B}$ in preparation for solving the overdetermined system $\boldsymbol{AX} \cong \boldsymbol{B}$, where:

- M is greater than N.
- NAUX is greater than or equal to 2N+max(M, N, NB) = 7.
- The singular values are returned in S.
- The matrix ***V*** is returned in A.
- The matrix $\boldsymbol{U}^{T}\boldsymbol{B}$ is returned in B.

Call Statement and Input

```
            IOPT  A  LDA  B  LDB  NB  S   M   N   AUX  NAUX
             |    |   |   |   |   |   |   |   |    |    |
CALL DGESVF( 2  , A , 3 , B , 3 , 2 , S , 3 , 2 , AUX , 7  )

        ┌        ┐
        │ 1.0  4.0 │
A    =  │ 2.0  5.0 │
        │ 3.0  6.0 │
        └        ┘

        ┌         ┐
        │ 7.0  10.0 │
B    =  │ 8.0  11.0 │
        │ 9.0  12.0 │
        └         ┘
```

Output

```
        ┌                ┐
        │  0.922  -0.386 │
A    =  │ -0.386  -0.922 │
        │   .       .    │
        └                ┘

        ┌                  ┐
        │  -1.310   -2.321 │
B    =  │ -13.867  -18.963 │
        │    .        .    │
        └                  ┘

X       =   (0.773, 9.508)
```

Example 5: This example computes the singular values, s, and matrices V and U^TB in preparation for solving the overdetermined system $AX \cong B$. The singular values are sorted in descending order, and the columns of V and the rows of U^TB are swapped to correspond to the sorted singular values.

- M is greater than N.
- NAUX is greater than or equal to 2N+max(M, N, NB) = 7.
- The singular values are returned in S.
- The matrix V is returned in A.
- The matrix U^TB is returned in B.

Call Statement and Input

```
            IOPT  A  LDA  B  LDB  NB  S   M   N   AUX  NAUX
             |    |   |   |   |   |   |   |   |    |    |
CALL DGESVF( 12 , A , 3 , B , 3 , 2 , S , 3 , 2 , AUX , 7  )
```

```
        ⎡          ⎤
        ⎢ 1.0  4.0 ⎥
A    =  ⎢ 2.0  5.0 ⎥
        ⎢ 3.0  6.0 ⎥
        ⎣          ⎦
```

```
        ⎡           ⎤
        ⎢ 7.0  10.0 ⎥
B    =  ⎢ 8.0  11.0 ⎥
        ⎢ 9.0  12.0 ⎥
        ⎣           ⎦
```

Output

```
        ⎡              ⎤
        ⎢ -0.386   0.922 ⎥
A    =  ⎢ -0.922  -0.386 ⎥
        ⎢   .       .    ⎥
        ⎣              ⎦
```

```
        ⎡                 ⎤
        ⎢ -13.867  -18.963 ⎥
B    =  ⎢  -1.310   -2.321 ⎥
        ⎢   .         .    ⎥
        ⎣                 ⎦
```

```
S        =   (9.508, 0.773)
```

SGESVS and DGESVS—Linear Least Squares Solution for a General Matrix Using the Singular Value Decomposition

These subroutines compute the minimal norm linear least squares solution of $\boldsymbol{AX} \cong \boldsymbol{B}$, where $\boldsymbol{A}$ is a general matrix, using the singular value decomposition computed by SGESVF or DGESVF.

Table 115. Data Types

V, ***UB***, ***X***, ***s***, τ	**Subroutine**
Short-precision real	SGESVS
Long-precision real	DGESVS

Syntax

Fortran	CALL SGESVS | DGESVS (*v*, *ldv*, *ub*, *ldub*, *nb*, *s*, *x*, *ldx*, *m*, *n*, *tau*)
C and C++	sgesvs | dgesvs (*v*, *ldv*, *ub*, *ldub*, *nb*, *s*, *x*, *ldx*, *m*, *n*, *tau*);
PL/I	CALL SGESVS | DGESVS (*v*, *ldv*, *ub*, *ldub*, *nb*, *s*, *x*, *ldx*, *m*, *n*, *tau*);

On Entry

v
is the orthogonal matrix $\boldsymbol{V}$ of order *n* in the singular value decomposition of matrix $\boldsymbol{A}$. It is produced by a preceding call to SGESVF or DGESVF, where it corresponds to output argument *a*.

Specified as: an *ldv* by (at least) *n* array, containing numbers of the data type indicated in Table 115.

ldv
is the leading dimension of the array specified for *v*. Specified as: a fullword integer; $ldv > 0$ and $ldv \geq n$.

ub
is an *n* by *nb* matrix, containing $\boldsymbol{U}^T\boldsymbol{B}$. It is produced by a preceding call to SGESVF or DGESVF, where it corresponds to output argument *b*. On output, $\boldsymbol{U}^T\boldsymbol{B}$ is overwritten; that is, the original input is not preserved.

Specified as: an *ldub* by (at least) *nb* array, containing numbers of the data type indicated in Table 115.

ldub
is the leading dimension of the array specified for *ub*. Specified as: a fullword integer; $ldub > 0$ and $ldub \geq n$.

nb
is the number of columns in matrices $\boldsymbol{X}$ and $\boldsymbol{U}^T\boldsymbol{B}$. Specified as: a fullword integer; $nb > 0$.

s
is the vector $\boldsymbol{s}$ of length *n*, containing the singular values of matrix $\boldsymbol{A}$. It is produced by a preceding call to SGESVF or DGESVF, where it corresponds to output argument *s*.

Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 115; $s_i \geq 0$.

x
See On Return.

ldx
is the leading dimension of the array specified for *x*. Specified as: a fullword integer; $ldx > 0$ and $ldx \geq n$.

m
is the number of rows in matrix ***A***. Specified as: a fullword integer; $m \geq 0$.

n
is the number of columns in matrix ***A***, the order of matrix ***V***, the number of elements in vector ***s***, the number of rows in matrix ***UB***, and the number of rows in matrix ***X***. Specified as: a fullword integer; $n \geq 0$.

tau
is the error tolerance τ. Any singular values in vector ***s*** that are less than τ are treated as zeros when computing matrix ***X***. Specified as: a number of the data type indicated in Table 115 on page 684; $\tau \geq 0$. For more information on the values for τ, see "Notes."

On Return

x
is an *n* by *nb* matrix, containing the minimal norm linear least solutions of $\boldsymbol{AX} \cong \boldsymbol{B}$. The *nb* column vectors of ***X*** contain minimal norm solution vectors for *nb* distinct linear least squares problems.

Returned as: an *ldx* by (at least) *nb* array, containing numbers of the data type indicated in Table 115 on page 684.

Notes

1. ***V***, ***X***, ***s***, and $\boldsymbol{U}^T\boldsymbol{B}$ can have no common elements; otherwise the results are unpredictable.
2. In problems involving experimental data, τ should reflect the absolute accuracy of the matrix elements:

   $$\tau \geq \max(|\Delta_{ij}|)$$

   where Δ_{ij} are the errors in a_{ij}. In problems where the matrix elements are known exactly or are only affected by roundoff errors:

   $$\left[\tau \geq \varepsilon \left(\sqrt{mn}\right)\right] \max\left(s_j\right) \qquad \text{for } j = (1, \ldots, n)$$

   where:

   ε is equal to 0.11920E−06 for SGESVS and 0.22204D−15 for DGESVS.
   s is a vector containing the singular values of matrix ***A***.

   For more information, see references [13], [55], [72], and pages 134 to 151 in reference [93].

Function: The minimal norm linear least squares solution of $\boldsymbol{AX} \cong \boldsymbol{B}$, where ***A*** is a real general matrix, is computed using the singular value decomposition, produced by a preceding call to SGESVF or DGESVF. From SGESVF or DGESVF, the singular value decomposition of ***A*** is given by the following:

$$\boldsymbol{A} = \boldsymbol{U}\Sigma\boldsymbol{V}^T$$

The linear least squares of solution X, for $AX \cong B$, is given by the following formula:

$$X = V\Sigma^{+}U^{T}B$$

where:

$\Sigma +$ is the diagonal matrix with elements σ_j^+, where:
$\sigma_j^+ = 1.0 / \sigma_j$ if $\sigma_j \geq \tau$ and $\sigma_j \neq 0$
$\sigma_j^+ = 0$ for all other cases

If *m* or *n* is equal to 0, no computation is performed. See references [13], [55], [72], and pages 134 to 151 in reference [93]. These algorithms have a tendency to generate underflows that may hurt overall performance. The system default is to mask underflow, which improves the performance of these subroutines.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. $ldv \leq 0$
2. $n > ldv$
3. $ldub \leq 0$
4. $n > ldub$
5. $ldx \leq 0$
6. $n > ldx$
7. $nb \leq 0$
8. $m < 0$
9. $n < 0$
10. $\tau < 0$

Example 1: This example finds the linear least squares solution for the underdetermined system $AX \cong B$, using the singular value decomposition computed by DGESVF. Matrix A is:

$$\begin{bmatrix} 1.0 & 2.0 & 2.0 \\ 2.0 & 4.0 & 5.0 \end{bmatrix}$$

and matrix B is:

$$\begin{bmatrix} 1.0 \\ 4.0 \end{bmatrix}$$

On output, matrix U^TB is overwritten.

Note: This example corresponds to Example 3 of DGESVF on page 681.

Call Statement and Input

```
            V  LDV  UB  LDUB  NB  S   X  LDX  M   N   TAU
            |   |   |    |    |   |   |   |   |   |    |
CALL DGESVS( V , 3 , UB , 3  , 1 , S , X , 3 , 2 , 3 , TAU )
```

$$V = \begin{bmatrix} -0.304 & -0.894 & 0.328 \\ -0.608 & 0.447 & 0.656 \\ -0.733 & 0.000 & -0.680 \end{bmatrix}$$

$$UB = \begin{bmatrix} -4.061 \\ 0.000 \\ -0.714 \end{bmatrix}$$

```
S        =  (7.342, 0.000, 0.305)
TAU      =  0.3993D-14
```

Output

$$X = \begin{bmatrix} -0.600 \\ -1.200 \\ 2.000 \end{bmatrix}$$

Example 2: This example finds the linear least squares solution for the overdetermined system $\boldsymbol{AX} \cong \boldsymbol{B}$, using the singular value decomposition computed by DGESVF. Matrix $\boldsymbol{A}$ is:

$$\begin{bmatrix} 1.0 & 4.0 \\ 2.0 & 5.0 \\ 3.0 & 6.0 \end{bmatrix}$$

and where $\boldsymbol{B}$ is:

$$\begin{bmatrix} 7.0 & 10.0 \\ 8.0 & 11.0 \\ 9.0 & 12.0 \end{bmatrix}$$

On output, matrix $\boldsymbol{U}^{T}\boldsymbol{B}$ is overwritten.

Note: This example corresponds to Example 4 of DGESVF on page 682.

Call Statement

```
             V  LDV  UB  LDUB  NB  S   X  LDX  M   N   TAU
             |   |   |    |    |   |   |   |   |   |    |
CALL DGESVS( V , 3 , UB , 3  , 2 , S , X , 2 , 3 , 2 , TAU )
```

Input

```
         ┌                ┐
         │  0.922  -0.386 │
V    =   │ -0.386  -0.922 │
         │   .       .    │
         └                ┘

         ┌                  ┐
         │  -1.310   -2.321 │
UB   =   │ -13.867  -18.963 │
         │    .        .    │
         └                  ┘

S        =  (0.773, 9.508)
TAU      =  0.5171D-14
```

Output

```
         ┌                ┐
X    =   │ -1.000  -2.000 │
         │  2.000   3.000 │
         └                ┘
```

SGELLS and DGELLS—Linear Least Squares Solution for a General Matrix Using a QR Decomposition with Column Pivoting

These subroutines compute the minimal norm linear least squares solution of $\boldsymbol{AX} \cong \boldsymbol{B}$, using a QR decomposition with column pivoting.

Table 116. Data Types

A, B, X, rn, τ, *aux*	**Subroutine**
Short-precision real	SGELLS
Long-precision real	DGELLS

Syntax

Fortran	CALL SGELLS | DGELLS (*iopt*, *a*, *lda*, *b*, *ldb*, *x*, *ldx*, *rn*, *tau*, *m*, *n*, *nb*, *k*, *aux*, *naux*)
C and C++	sgells | dgells (*iopt*, *a*, *lda*, *b*, *ldb*, *x*, *ldx*, *rn*, *tau*, *m*, *n*, *nb*, *k*, *aux*, *naux*);
PL/I	CALL SGELLS | DGELLS (*iopt*, *a*, *lda*, *b*, *ldb*, *x*, *ldx*, *rn*, *tau*, *m*, *n*, *nb*, *k*, *aux*, *naux*);

On Entry

iopt
indicates the type of computation to be performed, where:

If *iopt* = 0, ***X*** is computed.

If *iopt* = 1, ***X*** and the Euclidean Norm of the residual vectors are computed.

Specified as: a fullword integer; *iopt* = 0 or 1.

a
is the *m* by *n* coefficient matrix ***A***. On output, ***A*** is overwritten; that is, the original input is not preserved. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 116.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; *lda* > 0 and *lda* ≥ *m*.

b
is the *m* by *nb* matrix ***B***, containing the right-hand sides of the linear systems. The *nb* column vectors of ***B*** contain right-hand sides for *nb* distinct linear least squares problems. On output, ***B*** is overwritten; that is, the original input is not preserved.

Specified as: an *ldb* by (at least) *nb* array, containing numbers of the data type indicated in Table 116.

ldb
is the leading dimension of the array specified for *b*. Specified as: a fullword integer; *ldb* > 0 and *ldb* ≥ *m*.

x
See On Return.

ldx
is the leading dimension of the array specified for *x*. Specified as: a fullword integer; *ldx* > 0 and *ldx* ≥ *n*.

rn
See On Return.

tau
is the tolerance τ, used to determine the subset of the columns of ***A*** used in the solution. Specified as: a number of the data type indicated in Table 116; $\tau \geq 0$. For more information on how to select a value for τ, see "Notes" on page 691.

m
is the number of rows in matrices ***A*** and ***B***. Specified as: a fullword integer; $m \geq 0$.

n
is the number of columns in matrix ***A*** and the number of rows in matrix ***X***. Specified as: a fullword integer; $n \geq 0$.

nb
is the number of columns in matrices ***B*** and ***X*** and the number of elements in vector ***rn***. Specified as: a fullword integer; $nb > 0$.

k
See On Return.

aux
has the following meaning:

If $naux = 0$ and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is the storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing numbers of the data type indicated in Table 116 on page 689. On output, the contents of *aux* are overwritten.

naux
is the size of the work area specified by *aux*—that is, the number of elements in *aux*. Specified as: a fullword integer, where:

If $naux = 0$ and error 2015 is unrecoverable, SGELLS and DGELLS dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, It must have the following values:

$naux \geq 3n+\max(n, nb)$ for SGELLS

$naux \geq [\text{ceiling}(2.5n)+\max(n, nb)]$ for DGELLS

On Return

x
is the solution matrix ***X***, with *n* rows and *nb* columns, where:

If $k \neq 0$, the *nb* column vectors of ***X*** contain minimal norm least squares solutions for *nb* distinct linear least squares problems. The elements in each solution vector correspond to the original columns of ***A***.

If $k = 0$, the *nb* column vectors of ***X*** are set to 0.

Returned as: an *ldx* by (at least) *nb* array, containing numbers of the data type indicated in Table 116 on page 689.

rn
is the vector ***rn*** of length *nb*, where:

If $iopt = 0$ or $k = 0$, ***rn*** is not used in the computation.

If $iopt = 1$, rn_i is the Euclidean Norm of the residual vector for the linear least squares problem defined by the *i*-th column vector of ***B***.

Returned as: a one-dimensional array of (at least) *nb*, containing numbers of the data type indicated in Table 116 on page 689.

k
is the number of columns of matrix ***A*** used in the solution. Returned as: a fullword integer; k = (the number of diagonal elements of matrix ***R*** exceeding τ in magnitude).

Notes

1. In your C program, argument *k* must be passed by reference.
2. If *ldb* $\geq$ max(*m*, *n*), matrix ***X*** and matrix ***B*** can be the same; otherwise, matrix ***X*** and matrix ***B*** can have no common elements, or the results are unpredictable.
3. The following items must have no common elements; otherwise, results are unpredictable:
   - Matrices ***A*** and ***X***, vector ***rn***, and the data area specified for *aux*
   - Matrices ***A*** and ***B***, vector ***rn***, and the data area specified for *aux*.
4. If the relative uncertainty in the matrix ***B*** is ρ, then:

   $$\tau \geq \rho\|\boldsymbol{A}\|_F$$

   See references [44], [59], and [72] for additional guidance on determining suitable values for τ.
5. When you specify *iopt* = 0, you must also specify a dummy argument for *rn*. For more details, see "Example 1" on page 692.
6. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: The minimal norm linear least squares solution of $\boldsymbol{AX} \cong \boldsymbol{B}$ is computed using a QR decomposition with column pivoting, where:

A is an *m* by *n* real general matrix.
B is an *m* by *nb* real general matrix.
X is an *n* by *nb* real general matrix.

Optionally, the Euclidean Norms of the residual vectors can be computed. Following are the steps involved in finding the minimal norm linear least squares solution of $\boldsymbol{AX} \cong \boldsymbol{B}$. ***A*** is decomposed, using Householder transformations and column pivoting, into the following form:

$$\boldsymbol{AP} = \boldsymbol{QR}$$

where:

P is a permutation matrix.
Q is an orthogonal matrix.
R is an upper triangular matrix.

k is the first index, where:

$$|r_{k+1,k+1}| \leq \tau$$

If $k = n$, the minimal norm linear least squares solution is obtained by solving $\boldsymbol{RX} = \boldsymbol{Q}^T\boldsymbol{B}$ and reordering ***X*** to correspond to the original columns of ***A***.

If $k < n$, $\boldsymbol{R}$ has the following form:

$$\boldsymbol{R} = \begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ 0 & 0 \end{bmatrix}$$

To find the minimal norm linear least squares solution, it is necessary to zero the submatrix $\boldsymbol{R}_{12}$ using Householder transformations. See references [44], [59], and [72]. If m or n is equal to 0, no computation is performed. These algorithms have a tendency to generate underflows that may hurt overall performance. The system default is to mask underflow, which improves the performance of these subroutines.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, $naux = 0$, and unable to allocate work area.

Computational Errors: None

Input-Argument Errors

1. $iopt \neq 0$ or 1
2. $lda \leq 0$
3. $m > lda$
4. $ldb \leq 0$
5. $m > ldb$
6. $ldx \leq 0$
7. $n > ldx$
8. $m < 0$
9. $n < 0$
10. $nb \leq 0$
11. $\tau < 0$
12. Error 2015 is recoverable or $naux \neq 0$, and $naux$ is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example 1: This example solves the underdetermined system $\boldsymbol{AX} \cong \boldsymbol{B}$. On output, $\boldsymbol{A}$ and $\boldsymbol{B}$ are overwritten. DUMMY is used as a placeholder for argument *rn*, which is not used in the computation.

Call Statement and Input

```
            IOPT  A  LDA  B  LDB  X  LDX   RN     TAU   M   N   NB  K   AUX  NAUX
             |    |   |   |   |   |   |    |      |    |   |   |   |    |    |
CALL DGELLS( 0  , A , 2 , B , 2 , X , 3 , DUMMY , TAU , 2 , 3 , 1 , K , AUX , 11 )
```

```
        ⎡                  ⎤
A    =  ⎢ 1.0  2.0  2.0 ⎥
        ⎢ 2.0  4.0  5.0 ⎥
        ⎣                  ⎦

        ⎡      ⎤
B    =  ⎢ 1.0 ⎥
        ⎢ 4.0 ⎥
        ⎣      ⎦

TAU      =   0.0
```

Output

```
        ⎡         ⎤
        ⎢ -0.600 ⎥
X    =  ⎢ -1.200 ⎥
        ⎢  2.000 ⎥
        ⎣         ⎦

K        =   2
```

Example 2: This example solves the overdetermined system $\boldsymbol{AX} \cong \boldsymbol{B}$. On output, $\boldsymbol{A}$ and $\boldsymbol{B}$ are overwritten. `DUMMY` is used as a placeholder for argument *rn*, which is not used in the computation.

Call Statement and Input

```
            IOPT  A  LDA  B  LDB  X  LDX   RN     TAU   M   N   NB  K   AUX  NAUX
             |    |   |   |   |   |   |     |      |    |   |   |   |    |    |
CALL DGELLS( 0  , A , 3 , B , 3 , X , 2 , DUMMY , TAU , 3 , 2 , 2 , K , AUX , 7  )
```

```
        ⎡           ⎤
        ⎢ 1.0  4.0 ⎥
A    =  ⎢ 2.0  5.0 ⎥
        ⎢ 3.0  6.0 ⎥
        ⎣           ⎦

        ⎡            ⎤
        ⎢ 7.0  10.0 ⎥
B    =  ⎢ 8.0  11.0 ⎥
        ⎢ 9.0  12.0 ⎥
        ⎣            ⎦

TAU      =   0.0
```

Output

```
        ⎡                  ⎤
X    =  ⎢ -1.000  -2.000 ⎥
        ⎢  2.000   3.000 ⎥
        ⎣                  ⎦

K        =   2
```

Example 3: This example solves the overdetermined system $\boldsymbol{AX} \cong \boldsymbol{B}$ and computes the Euclidean Norms of the residual vectors. On output, $\boldsymbol{A}$ and $\boldsymbol{B}$ are overwritten.

Call Statement and Input

```
            IOPT  A  LDA  B  LDB  X  LDX  RN   TAU   M   N   NB  K   AUX  NAUX
             |    |   |   |   |   |   |   |     |    |   |   |   |    |    |
CALL DGELLS( 1  , A , 3 , B , 3 , X , 2 , RN , TAU , 3 , 2 , 2 , K , AUX , 7  )
```

```
                 ⎡           ⎤
                 ⎢ 1.1  -4.3 ⎥
A    =           ⎢ 2.0  -5.0 ⎥
                 ⎢ 3.0  -6.0 ⎥
                 ⎣           ⎦

                 ⎡           ⎤
                 ⎢ -7.0  10.0 ⎥
B    =           ⎢ -8.0  11.0 ⎥
                 ⎢ -9.0  12.0 ⎥
                 ⎣           ⎦

TAU      =   0.0
```

Output

```
                 ⎡               ⎤
X    =           ⎢ 0.543  -1.360 ⎥
                 ⎢ 1.785  -2.699 ⎥
                 ⎣               ⎦

                 ⎡       ⎤
RN   =           ⎢ 0.196 ⎥
                 ⎢ 0.275 ⎥
                 ⎣       ⎦

K        =   2
```

Chapter 11. Eigensystem Analysis

The eigensystem analysis subroutines are described in this chapter.

Overview of the Eigensystem Analysis Subroutines

The eigensystem analysis subroutines provide solutions to the algebraic eigensystem analysis problem $\boldsymbol{Az} = w\boldsymbol{z}$ and the generalized eigensystem analysis problem $\boldsymbol{Az} = w\boldsymbol{Bz}$ (Table 117). Many of the eigensystem analysis subroutines use the algorithms presented in *Linear Algebra* by Wilkinson and Reinsch [93] or use adaptations of EISPACK routines, as described in the *EISPACK Guide Lecture Notes in Computer Science* in reference [81] or in the *EISPACK Guide Extension Lecture Notes in Computer Science* in reference [55]. (EISPACK is available from the sources listed in reference [49].)

Table 117. List of Eigensystem Analysis Subroutines

Descriptive Name	Short-Precision Subroutine	Long-Precision Subroutine	Page
Eigenvalues and, Optionally, All or Selected Eigenvectors of a General Matrix	SGEEV CGEEV	DGEEV ZGEEV	698
Eigenvalues and, Optionally, the Eigenvectors of a Real Symmetric Matrix or a Complex Hermitian Matrix	SSPEV CHPEV	DSPEV ZHPEV	710
Extreme Eigenvalues and, Optionally, the Eigenvectors of a Real Symmetric Matrix or a Complex Hermitian Matrix	SSPSV CHPSV	DSPSV ZHPSV	719
Eigenvalues and, Optionally, the Eigenvectors of a Generalized Real Eigensystem, Az=wBz, where A and B Are Real General Matrices	SGEGV	DGEGV	727
Eigenvalues and, Optionally, the Eigenvectors of a Generalized Real Symmetric Eigensystem, Az=wBz, where A Is Real Symmetric and B Is Real Symmetric Positive Definite	SSYGV	DSYGV	733

Performance and Accuracy Considerations

1. The accuracy of the resulting eigenvalues and eigenvectors varies between the short- and long-precision versions of each subroutine. The degree of difference depends on the size and conditioning of the matrix computation. Some of the subroutines have examples illustrating this difference.
2. The short precision subroutines provide increased accuracy by accumulating intermediate results in long precision. Occasionally, for performance reasons, these intermediate results are stored.
3. If you want to compute 10% or fewer eigenvalues only, or you want to compute 30% or fewer eigenvalues and eigenvectors, you get better performance if you use _SPSV and _HPSV instead of _SPEV and _HPEV, respectively. For all other uses, you should use _SPEV and _HPEV.

© Copyright IBM Corp. 1997

4. There are some ESSL-specific rules that apply to the results of computations on the workstation processors using the ANSI/IEEE standards. For details, see "What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?" on page 44.

Eigensystem Analysis Subroutines

This section contains the eigensystem analysis subroutine

SGEEV, DGEEV, CGEEV, and ZGEEV—Eigenvalues and, Optionally, All or Selected Eigenvectors of a General Matrix

SGEEV and DGEEV compute the eigenvalues and, optionally, all or selected eigenvectors of real general matrix ***A***. CGEEV and ZGEEV compute the eigenvalues and, optionally, all or selected eigenvectors of complex general matrix ***A***. Eigenvalues are returned in complex vector ***w***, and eigenvectors are returned in complex matrix ***Z***:

$$Az = wz$$

Table 118. Data Types

A	***w*, *Z***	*aux*	**Subroutine**
Short-precision real	Short-precision complex	Short-precision real	SGEEV
Long-precision real	Long-precision complex	Long-precision real	DGEEV
Short-precision complex	Short-precision complex	Short-precision real	CGEEV
Long-precision complex	Long-precision complex	Long-precision real	ZGEEV

Syntax

Fortran	CALL SGEEV | DGEEV | CGEEV | ZGEEV (*iopt*, *a*, *lda*, *w*, *z*, *ldz*, *select*, *n*, *aux*, *naux*)
C and C++	sgeev | dgeev | cgeev | zgeev (*iopt*, *a*, *lda*, *w*, *z*, *ldz*, *select*, *n*, *aux*, *naux*);
PL/I	CALL SGEEV | DGEEV | CGEEV | ZGEEV (*iopt*, *a*, *lda*, *w*, *z*, *ldz*, *select*, *n*, *aux*, *naux*);

On Entry

iopt
indicates the type of computation to be performed, where:

If *iopt* = 0, eigenvalues only are computed.

If *iopt* = 1, eigenvalues and eigenvectors are computed.

If *iopt* = 2, eigenvalues and eigenvectors corresponding to selected eigenvalues are computed.

Specified as: a fullword integer; *iopt* = 0, 1, or 2.

a
is the real or complex general matrix ***A*** of order *n*, whose eigenvalues and, optionally, eigenvectors are computed. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 118. On output, A is overwritten; that is, the original input is not preserved.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; *lda* > 0 and *lda* ≥ *n*.

w
See On Return.

z
See On Return.

ldz
has the following meaning, where:

If *iopt* = 0, it is not used in the computation.

If *iopt* = 1 or 2, it is the leading dimension of the output array specified for *z*.

Specified as: a fullword integer. It must have the following values, where:

If *iopt* = 0, *ldz* > 0.

If *iopt* = 1 or 2, *ldz* > 0 and *ldz* ≥ *n*.

select
has the following meaning, where:

If *iopt* = 0 or 1, it is not used in the computation.

If *iopt* = 2, it is the logical vector ***select*** of length *n* whose true elements indicate those eigenvalues whose corresponding eigenvectors are to be computed.

Specified as: a one-dimensional array of (at least) length *n*, containing logical data items. Element values can be true (.TRUE.) or false (.FALSE.).

n
is the order of matrix ***A***. Specified as: a fullword integer; *n* ≥ 0.

aux
has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is a storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing numbers of the data type indicated in Table 118 on page 698. On output, the contents are overwritten.

naux
is the size of the work area specified by *aux*—that is, the number of elements in *aux*. Specified as: a fullword integer, where:

If *naux* = 0 and error 2015 is unrecoverable, SGEEV, DGEEV, CGEEV, and ZGEEV dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, it must have the following value, where:

For SGEEV and DGEEV:
If *iopt* = 0, *naux* ≥ *n*.
If *iopt* = 1, *naux* ≥ 2*n*.
If *iopt* = 2,

naux ≥ n^2+4*n*.

For CGEEV and ZGEEV:
If *iopt* = 0, *naux* ≥ 2*n*.
If *iopt* = 1, *naux* ≥ 3*n*.
If *iopt* = 2, *naux* ≥ 2n^2+5*n*.

On Return

w
is the vector ***w*** of length *n*, containing the eigenvalues of ***A***. The eigenvalues are unordered. For SGEEV and DGEEV, complex conjugate pairs appear consecutively with the eigenvalue having the positive imaginary part first.

Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 118 on page 698.

z

has the following meaning, where:

If *iopt* = 0, it is not used in the computation.

If *iopt* = 1, it is the matrix ***Z*** of order *n*, containing the eigenvectors of matrix ***A***. The eigenvector in column *i* of matrix ***Z*** corresponds to the eigenvalue w_i.

If *iopt* = 2, it is the *n* by *m* matrix ***Z***, containing the *m* eigenvectors of matrix ***A***, which correspond to the *m* selected eigenvalues in w_i,

where *m* is the number of true elements in the logical vector ***select***. The eigenvector in column *i* of matrix ***Z*** corresponds to the *i*-th selected eigenvalue. Any eigenvector that does not converge is set to 0.

Returned as: a two-dimensional array, containing numbers of the data type indicated in Table 118 on page 698, where:

If *iopt* = 1, the array must be *ldz* by (at least) *n*.

If *iopt* = 2, the array must be *ldz* by (at least) *m*.

Notes

1. When you specify *iopt* = 0, you must also specify:
   - A positive value for *ldz*
   - A dummy argument for *z* (see "Example 1" on page 703)
   - A dummy argument for *select*.
2. When you specify *iopt* = 1, you must also specify a dummy argument for *select*.
3. The following items must have no common elements: matrix ***A***, matrix ***Z***, vector ***w***, vector ***select***, and the data area specified for *aux*; otherwise, results are unpredictable. See "Concepts" on page 55.
4. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: The next two sections describe the methods used to compute the eigenvectors and eigenvalues for either a real general matrix or a complex general matrix. For more information on these methods, see references [39], [43], [45], [60], [81], [91], and [93]. If *n* is 0, no computation is performed. The results of the computations using short- and long-precision data can vary in accuracy. See the examples for an illustration of the difference in the results.

These algorithms have a tendency to generate underflows that may hurt overall performance. The system default is to mask underflow, which improves the performance of these subroutines.

Real General Matrix: The eigenvalues and, optionally, all or selected eigenvectors of a real general matrix ***A*** are computed as follows.

- For *iopt* = 0, the eigenvalues are computed as follows:
  1. Balance the real general matrix ***A***.
  2. Reduce the balanced matrix to a real upper Hessenberg matrix using orthogonal similarity transformations.

3. Compute the eigenvalues of the real upper Hessenberg matrix using the QR algorithm.
4. The eigenvalues are returned in vector ***w***.

- For *iopt* = 1, the eigenvalues and eigenvectors are computed as follows:

  1. Balance the real general matrix ***A***.
  2. Reduce the balanced matrix to a real upper Hessenberg matrix using orthogonal similarity transformations.
  3. Accumulate the orthogonal similarity transformations used in the reduction of the real balanced matrix to upper Hessenberg form.
  4. Compute the eigenvalues of the real upper Hessenberg matrix and the eigenvectors of the corresponding real balanced matrix using the QR algorithm.
  5. Back transform the eigenvectors of the balanced matrix to the eigenvectors of the original matrix.
  6. The eigenvalues are returned in vector ***w***, and the eigenvectors are returned in matrix ***Z***.

- For *iopt* = 2, the eigenvalues and selected eigenvectors are computed as follows:

  1. Balance the real general matrix ***A***.
  2. Reduce the balanced matrix to a real upper Hessenberg matrix using orthogonal similarity transformations.
  3. Compute the eigenvalues of the real upper Hessenberg matrix using the QR algorithm.
  4. Compute the eigenvectors of the real upper Hessenberg matrix corresponding to selected eigenvalues, indicated in the logical vector ***select***, using inverse iteration.
  5. Back transform the eigenvectors of the real upper Hessenberg matrix to the eigenvectors of the balanced matrix.
  6. Back transform the eigenvectors of the balanced matrix to the eigenvectors of the original matrix.
  7. The eigenvalues are returned in vector ***w***, and the eigenvectors are returned in matrix ***Z***.

Complex General Matrix: The eigenvalues and, optionally, all or selected eigenvectors of a complex general matrix ***A*** are computed as follows.

- For *iopt* = 0, the eigenvalues are computed as follows:

  1. Balance the complex general matrix ***A***.
  2. Reduce the complex balanced matrix to a complex upper Hessenberg matrix using unitary similarity transformations.
  3. Compute the eigenvalues of the complex upper Hessenberg matrix using a QR algorithm with implicit shift.
  4. The eigenvalues are returned in vector ***w***.

- For *iopt* = 1, the eigenvalues and eigenvectors are computed as follows:

  1. Balance the complex general matrix ***A***.
  2. Reduce the complex balanced matrix to a complex upper Hessenberg matrix using unitary similarity transformations.
  3. Accumulate the unitary similarity transformations used in the reduction of the complex balanced matrix to complex upper Hessenberg form.
  4. Compute the eigenvalues of the complex upper Hessenberg matrix and the eigenvectors of the complex balanced matrix using a QR algorithm with implicit shift.
  5. Back transform the eigenvectors of the complex balanced matrix to the eigenvectors of the original matrix.

6. The eigenvalues are returned in vector ***w***, and the eigenvectors are returned in matrix ***Z***.

- For *iopt* = 2, the eigenvalues and selected eigenvectors are computed as follows:
  1. Balance the complex general matrix ***A***.
  2. Reduce the complex balanced matrix to a complex upper Hessenberg matrix using unitary similarity transformations.
  3. Compute the eigenvalues of the complex upper Hessenberg matrix using a QR algorithm with implicit shift.
  4. Compute the eigenvectors of the complex upper Hessenberg matrix corresponding to selected eigenvalues, indicated in the logical vector ***select***, using inverse iteration.
  5. Back transform the eigenvectors of the complex upper Hessenberg matrix to the eigenvectors of the complex balanced matrix.
  6. Back transform the eigenvectors of the complex balanced matrix to the eigenvectors of the original matrix.
  7. The eigenvalues are returned in vector ***w***, and the eigenvectors are returned in matrix ***Z***.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors

1. Eigenvalue (*i*) failed to converge after (*xxx*) iterations.
   - The eigenvalues (w_j, $j = n, n-1, ..., i+1$) are correct.
   - If *iopt* = 1, then *z* is modified, but no eigenvectors are correct.
   - *a* is modified.
   - The return code is set to 1.
   - *i* and *xxx* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2101 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. See "What Can You Do about ESSL Computational Errors?" on page 48.
2. Eigenvector (*yyy*) failed to converge after (*zzz*) iterations. (The computational error message may occur multiple times with processing continuing after each error, because the number of allowable errors for error code 2102 is set to be unlimited in the ESSL error option table.)
   - All eigenvalues are correct.
   - The eigenvector that failed to converge is set to 0.
   - Any selected eigenvectors for which this message has not been issued are correct.
   - *a* is modified.
   - The return code is set to 2.
   - *yyy* and *zzz* for the last eigenvector that failed to converge can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error 2199 in the ESSL error option table. See "What Can You Do about ESSL Computational Errors?" on page 48.

Input-Argument Errors

1. *iopt* ≠ 0, 1, or 2
2. *n* < 0
3. *lda* ≤ 0
4. *n* > *lda*
5. *ldz* ≤ 0 and *iopt* ≠ 0
6. *n* > *ldz* and *iopt* ≠ 0
7. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 3 is returned if error 2015 is recoverable.

Example 1: This example shows how to find the eigenvalues only of a real short-precision general matrix ***A*** of order 4, where:

- NAUX is equal to N.
- AUX contains N elements.
- LDZ is set to avoid an error condition.
- DUMMY1 and DUMMY2 are used as placeholders for arguments *z* and *select*, which are not used in the computation.
- On output, A has been overwritten.

Note: This matrix is used in Example 5.5 in referenced text [60].

Call Statement and Input

```
           IOPT  A  LDA  W     Z     LDZ  SELECT   N   AUX  NAUX
            |    |   |   |     |      |     |      |    |    |
CALL SGEEV( 0  , A , 4 , W , DUMMY1 , 1 , DUMMY2 , 4 , AUX , 4  )
```

```
        ⎡                    ⎤
        ⎢ -2.0  2.0  2.0  2.0 ⎥
A    =  ⎢ -3.0  3.0  2.0  2.0 ⎥
        ⎢ -2.0  0.0  4.0  2.0 ⎥
        ⎢ -1.0  0.0  0.0  5.0 ⎥
        ⎣                    ⎦
```

Output

```
        ⎡                      ⎤
        ⎢ (0.999999,  0.000000) ⎥
W    =  ⎢ (2.000001,  0.000000) ⎥
        ⎢ (2.999996,  0.000000) ⎥
        ⎢ (3.999999,  0.000000) ⎥
        ⎣                      ⎦
```

Example 2: This example shows how to find the eigenvalues and eigenvectors of a real short-precision general matrix ***A*** of order 3, where:

- NAUX is equal to 2N.
- AUX contains 2N elements.
- DUMMY is used as a placeholder for argument *select*, which is not used in the computation.
- On output, A has been overwritten.

Note: This matrix is used in Example 5.1 in referenced text [60].

Call Statement and Input

```
           IOPT  A  LDA  W   Z  LDZ  SELECT  N   AUX  NAUX
            |    |   |   |   |   |     |     |    |    |
CALL SGEEV( 1  , A , 3 , W , Z , 3 , DUMMY , 3 , AUX , 6  )
```

```
        ⎡                    ⎤
        ⎢  33.0   16.0   72.0 ⎥
A    =  ⎢ -24.0  -10.0  -57.0 ⎥
        ⎢  -8.0   -4.0  -17.0 ⎥
        ⎣                    ⎦
```

Output

```
        ⎡                    ⎤
        ⎢ (3.000022, 0.000000) ⎥
W    =  ⎢ (1.000019, 0.000000) ⎥
        ⎢ (1.999961, 0.000000) ⎥
        ⎣                    ⎦
```

```
        ⎡                                                                ⎤
        ⎢  (2.498781, 0.000000)   (76.837608,  0.000000)   (79.999451,  0.000000) ⎥
Z    =  ⎢ (-1.874081, 0.000000)  (-61.470169,  0.000000)  (-64.999649,  0.000000) ⎥
        ⎢ (-0.624695, 0.000000)  (-20.489990,  0.000000)  (-19.999886,  0.000000) ⎥
        ⎣                                                                ⎦
```

Example 3: This example shows how to find the eigenvalues and eigenvectors of a real short-precision general matrix ***A*** of order 3, where:

- `NAUX` is equal to 2`N`.
- `AUX` contains 2`N` elements.
- `DUMMY` is used as a placeholder for argument *select*, which is not used in the computation.
- On output, `A` has been overwritten.

Note: This matrix is used in Example 5.4 in referenced text [60].

Call Statement and Input

```
           IOPT  A  LDA  W   Z  LDZ  SELECT  N   AUX  NAUX
            |    |   |   |   |   |     |     |    |    |
CALL SGEEV( 1  , A , 3 , W , Z , 3 , DUMMY , 3 , AUX , 6  )
```

```
        ⎡                  ⎤
        ⎢  8.0  -1.0  -5.0 ⎥
A    =  ⎢ -4.0   4.0  -2.0 ⎥
        ⎢ 18.0  -5.0  -7.0 ⎥
        ⎣                  ⎦
```

Output

```
        ⎡                    ⎤
        ⎢ (1.999999,  3.999998) ⎥
W    =  ⎢ (1.999999, -3.999998) ⎥
        ⎢ (0.999997,  0.000000) ⎥
        ⎣                    ⎦
```

```
        ⎡                                                                ⎤
        ⎢  (0.044710, 0.410578)   (0.044710, -0.410578)  (1.732048, 0.000000) ⎥
Z    =  ⎢ (-0.365868, 0.455287)  (-0.365868, -0.455287)  (3.464096, 0.000000) ⎥
        ⎢  (0.455287, 0.365868)   (0.455287, -0.365868)  (1.732049, 0.000000) ⎥
        ⎣                                                                ⎦
```

Example 4: This example shows how to find the eigenvalues and selected eigenvectors of a real short-precision general matrix ***A*** of order 4, where:

- `NAUX` is equal to N^2+4N.
- `AUX` contains `NAUX` elements.
- The first, third, and fourth eigenvectors are selected and appear in the first, second, and third columns of matrix Z, respectively.
- On output, `A` has been overwritten.

Note: This matrix is used in Example 5.5 in referenced text [60].

Call Statement and Input

```
           IOPT  A  LDA  W   Z  LDZ  SELECT   N   AUX  NAUX
            |    |   |   |   |   |     |      |    |    |
CALL SGEEV( 2  , A , 4 , W , Z , 4 , SELECT , 4 , AUX , 32 )
```

```
        ⎡                    ⎤
        ⎢ -2.0  2.0  2.0  2.0 ⎥
A    =  ⎢ -3.0  3.0  2.0  2.0 ⎥
        ⎢ -2.0  0.0  4.0  2.0 ⎥
        ⎢ -1.0  0.0  0.0  5.0 ⎥
        ⎣                    ⎦
```

```
           ⎡         ⎤
           ⎢ .TRUE.  ⎥
SELECT =   ⎢ .FALSE. ⎥
           ⎢ .TRUE.  ⎥
           ⎢ .TRUE.  ⎥
           ⎣         ⎦
```

Output

```
        ⎡                    ⎤
        ⎢ (0.999999,  0.000000) ⎥
W    =  ⎢ (2.000001,  0.000000) ⎥
        ⎢ (2.999996,  0.000000) ⎥
        ⎢ (3.999999,  0.000000) ⎥
        ⎣                    ⎦
```

```
        ⎡                                                                  ⎤
        ⎢ (1.000000, 0.000000)  (-0.674014, 0.000000)  (-0.474306, 0.000000) ⎥
Z    =  ⎢ (0.750000, 0.000000)  (-0.674014, 0.000000)  (-0.474306, 0.000000) ⎥
        ⎢ (0.500000, 0.000000)  (-0.674013, 0.000000)  (-0.474306, 0.000000) ⎥
        ⎢ (0.250000, 0.000000)  (-0.337006, 0.000000)  (-0.474305, 0.000000) ⎥
        ⎣                                                                  ⎦
```

Example 5: This example shows how to find the eigenvalues and selected eigenvectors of a real short-precision general matrix ***A*** of order 3, where:

- NAUX is equal to N^2+4N.
- AUX contains NAUX elements.
- The first and second eigenvectors are selected and appear in the first and second columns of matrix Z, respectively.
- On output, A has been overwritten.

Note: This matrix is used in Example 5.4 in referenced text [60].

Call Statement and Input

```
           IOPT  A  LDA  W   Z  LDZ  SELECT   N   AUX  NAUX
            |    |   |   |   |   |     |      |    |    |
CALL SGEEV( 2  , A , 3 , W , Z , 3 , SELECT , 3 , AUX , 21 )

        ⎡                  ⎤
        ⎢  8.0  -1.0  -5.0 ⎥
A    =  ⎢ -4.0   4.0  -2.0 ⎥
        ⎢ 18.0  -5.0  -7.0 ⎥
        ⎣                  ⎦

          ⎡         ⎤
          ⎢ .TRUE.  ⎥
SELECT =  ⎢ .TRUE.  ⎥
          ⎢ .FALSE. ⎥
          ⎣         ⎦
```

Output

```
        ⎡                      ⎤
        ⎢ (1.999999,  3.999998) ⎥
W    =  ⎢ (1.999999, -3.999998) ⎥
        ⎢ (0.999997,  0.000000) ⎥
        ⎣                      ⎦

        ⎡                                                  ⎤
        ⎢ (0.500000,  0.000000)  (0.500000,  0.000000) ⎥
Z    =  ⎢ (0.500000,  0.500000)  (0.500000, -0.500000) ⎥
        ⎢ (0.500000, -0.500000)  (0.500000,  0.500000) ⎥
        ⎣                                                  ⎦
```

Example 6: This example shows how the results of Example 5 would differ if matrix ***A*** was a real long-precision general matrix. On output, A has been overwritten.

Call Statement and Input

```
           IOPT  A  LDA  W   Z  LDZ  SELECT   N   AUX  NAUX
            |    |   |   |   |   |     |      |    |    |
CALL DGEEV( 2  , A , 3 , W , Z , 3 , SELECT , 3 , AUX , 21 )
```

Output

```
        ⎡                        ⎤
        ⎢ (2.000000,  4.000000) ⎥
W    =  ⎢ (2.000000, -4.000000) ⎥
        ⎢ (1.000000,  0.000000) ⎥
        ⎣                        ⎦

        ⎡                                                ⎤
        ⎢ (0.500000,  0.000000)  (0.500000,  0.000000) ⎥
Z    =  ⎢ (0.500000,  0.500000)  (0.500000, -0.500000) ⎥
        ⎢ (0.500000, -0.500000)  (0.500000,  0.500000) ⎥
        ⎣                                                ⎦
```

Example 7: This example shows how to find the eigenvalues only of a complex long-precision general matrix ***A*** of order 3, where:

- `NAUX` is equal to `2N`.
- `AUX` contains `2N` elements.
- `LDZ` is set to avoid an error condition.
- `DUMMY1` and `DUMMY2` are used as placeholders for arguments *z* and *select*, which are not used in the computation.
- On output, `A` has been overwritten.

Note: This matrix is used in Example 6.4 in referenced text [60].

Call Statement and Input

```
           IOPT  A  LDA  W     Z     LDZ  SELECT   N   AUX  NAUX
            |    |   |   |     |      |     |      |    |    |
CALL ZGEEV( 0  , A , 3 , W , DUMMY1 , 1 , DUMMY2 , 3 , AUX , 6  )

        ⎡                                                ⎤
        ⎢  (1.0,  2.0)   (3.0,  4.0)  (21.0, 22.0) ⎥
A    =  ⎢ (43.0, 44.0)  (13.0, 14.0)  (15.0, 16.0) ⎥
        ⎢  (5.0,  6.0)   (7.0,  8.0)  (25.0, 26.0) ⎥
        ⎣                                                ⎦
```

Output

```
        ⎡                        ⎤
        ⎢ (39.776655, 42.995668) ⎥
W    =  ⎢ (-7.477530,  6.880321) ⎥
        ⎢  (6.700876, -7.875989) ⎥
        ⎣                        ⎦
```

Example 8: This example shows how to find the eigenvalues and eigenvectors of a complex long-precision general matrix ***A*** of order 4, where:

- `NAUX` is equal to `3N`.
- `AUX` contains `3N` elements.
- `DUMMY` is used as a placeholder for argument *select*, which is not used in the computation.
- On output, `A` has been overwritten.

Note: This matrix is used in Example 6.5 in referenced text [60].

Call Statement and Input

```
           IOPT  A  LDA  W   Z  LDZ  SELECT  N   AUX  NAUX
            |    |   |   |   |   |     |     |    |    |
CALL ZGEEV( 1  , A , 4 , W , Z , 4 , DUMMY , 4 , AUX , 12 )
```

```
        ⎡                                                    ⎤
        ⎢ (5.0, 9.0)  (5.0,  5.0)  (-6.0, -6.0)  (-7.0, -7.0) ⎥
A    =  ⎢ (3.0, 3.0)  (6.0, 10.0)  (-5.0, -5.0)  (-6.0, -6.0) ⎥
        ⎢ (2.0, 2.0)  (3.0,  3.0)  (-1.0,  3.0)  (-5.0, -5.0) ⎥
        ⎢ (1.0, 1.0)  (2.0,  2.0)  (-3.0, -3.0)   (0.0,  4.0) ⎥
        ⎣                                                    ⎦
```

Output

```
        ⎡                    ⎤
        ⎢ (4.000000, 8.000000) ⎥
W    =  ⎢ (2.000000, 6.000000) ⎥
        ⎢ (3.000000, 7.000000) ⎥
        ⎢ (1.000000, 5.000000) ⎥
        ⎣                    ⎦
```

```
        ⎡                                                                                ⎤
        ⎢ (0.625817, 0.229776)  (0.333009, -0.729358)  (-1.535458, 1.519551)  (0.000000, 3.464102) ⎥
Z    =  ⎢ (0.625817, 0.229776)  (0.666017, -1.458715)  (-1.535458, 1.519551)  (0.000000, 1.732051) ⎥
        ⎢ (0.625817, 0.229776)  (0.333009, -0.729358)  ( 0.000000, 0.000000)  (0.000000, 1.732051) ⎥
        ⎢ (0.000000, 0.000000)  (0.333009, -0.729358)  (-1.535458, 1.519551)  (0.000000, 1.732051) ⎥
        ⎣                                                                                ⎦
```

Example 9: This example shows how to find the eigenvalues and selected eigenvectors of a complex long-precision general matrix ***A*** of order 4, where:

- NAUX is equal to $2N^2+5N$.
- AUX contains NAUX elements.
- The first, third, and fourth eigenvectors are selected and appear in the first, second, and third columns of matrix Z, respectively.
- On output, A has been overwritten.

Note: This matrix is used in Example 6.5 in referenced text [60].

Call Statement and Input

```
           IOPT  A  LDA  W   Z  LDZ  SELECT   N   AUX  NAUX
            |    |   |   |   |   |     |      |    |    |
CALL ZGEEV( 2  , A , 4 , W , Z , 4 , SELECT , 4 , AUX , 52 )
```

```
        ⎡                                                    ⎤
        ⎢ (5.0, 9.0)  (5.0,  5.0)  (-6.0, -6.0)  (-7.0, -7.0) ⎥
A    =  ⎢ (3.0, 3.0)  (6.0, 10.0)  (-5.0, -5.0)  (-6.0, -6.0) ⎥
        ⎢ (2.0, 2.0)  (3.0,  3.0)  (-1.0,  3.0)  (-5.0, -5.0) ⎥
        ⎢ (1.0, 1.0)  (2.0,  2.0)  (-3.0, -3.0)   (0.0,  4.0) ⎥
        ⎣                                                    ⎦
```

```
          ⎡         ⎤
          ⎢ .TRUE.  ⎥
SELECT =  ⎢ .FALSE. ⎥
          ⎢ .TRUE.  ⎥
          ⎢ .TRUE.  ⎥
          ⎣         ⎦
```

Output

```
        ⎡                    ⎤
        ⎢ (4.000000, 8.000000) ⎥
W    =  ⎢ (2.000000, 6.000000) ⎥
        ⎢ (3.000000, 7.000000) ⎥
        ⎢ (1.000000, 5.000000) ⎥
        ⎣                    ⎦

        ⎡                                                                ⎤
        ⎢ (-0.748331, 0.000000)  (-0.935414, 0.000000)  (-1.247219, 0.000000) ⎥
Z    =  ⎢ (-0.748331, 0.000000)  (-0.935414, 0.000000)  (-0.623610, 0.000000) ⎥
        ⎢ (-0.748331, 0.000000)   (0.000000, 0.000000)  (-0.623610, 0.000000) ⎥
        ⎢  (0.000000, 0.000000)  (-0.935414, 0.000000)  (-0.623610, 0.000000) ⎥
        ⎣                                                                ⎦
```

SSPEV, DSPEV, CHPEV, and ZHPEV—Eigenvalues and, Optionally, the Eigenvectors of a Real Symmetric Matrix or a Complex Hermitian Matrix

SSPEV and DSPEV compute the eigenvalues and, optionally, the eigenvectors of real symmetric matrix ***A***, stored in lower- or upper-packed storage mode. CHPEV and ZHPEV compute the eigenvalues and, optionally, the eigenvectors of complex Hermitian matrix ***A***, stored in lower- or upper-packed storage mode. Eigenvalues are returned in vector ***w***, and eigenvectors are returned in matrix ***Z***:

$$Az = wz$$

where $A = A^T$ or $A = A^H$.

Table 119. Data Types

A, *z*	***w***, *aux*	**Subroutine**
Short-precision real	Short-precision real	SSPEV
Long-precision real	Long-precision real	DSPEV
Short-precision complex	Short-precision real	CHPEV
Long-precision complex	Long-precision real	ZHPEV

Note: For compatibility with earlier releases of ESSL, you can use the names SSLEV, DSLEV, CHLEV, and ZHLEV for SSPEV, DSPEV, CHPEV, and ZHPEV, respectively.

Syntax

Fortran	CALL SSPEV | DSPEV | CHPEV | ZHPEV (*iopt*, *ap*, *w*, *z*, *ldz*, *n*, *aux*, *naux*)
C and C++	sspev | dspev | chpev | zhpev (*iopt*, *ap*, *w*, *z*, *ldz*, *n*, *aux*, *naux*);
PL/I	CALL SSPEV | DSPEV | CHPEV | ZHPEV (*iopt*, *ap*, *w*, *z*, *ldz*, *n*, *aux*, *naux*);

On Entry

iopt
indicates the type of computation to be performed, where:

If *iopt* = 0 or 20, eigenvalues only are computed.

If *iopt* = 1 or 21, eigenvalues and eigenvectors are computed.

Specified as: a fullword integer; *iopt* = 0, 1, 20, or 21.

ap
is the real symmetric or complex Hermitian matrix ***A*** of order *n*, whose eigenvalues and, optionally, eigenvectors are computed. It is stored in an array, referred to as AP, where:

If *iopt* = 0 or 1, it is stored in lower-packed storage mode.

If *iopt* = 20 or 21, it is stored in upper-packed storage mode.

Specified as: a one-dimensional array of (at least) length $n(n+1)/2$, containing numbers of the data type indicated in Table 119. On output, for SSPEV and DSPEV

if *iopt* = 0 or 20, and for CHPEV and ZHPEV, AP is overwritten; that is, the original input is not preserved.

w

See On Return.

z

See On Return.

ldz

has the following meaning, where:

If *iopt* = 0 or 20, it is not used in the computation.

If *iopt* = 1 or 21, it is the leading dimension of the output array specified for *z*.

Specified as: a fullword integer. It must have the following value, where:

If *iopt* = 0 or 20, *ldz* > 0.

If *iopt* = 1 or 21, *ldz* > 0 and *ldz* ≥ *n*.

n

is the order of matrix ***A***. Specified as: a fullword integer; *n* ≥ 0.

aux

has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is a storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing numbers of the data type indicated in Table 119 on page 710. On output, the contents are overwritten.

naux

is the size of the work area specified by *aux*—that is, the number of elements in *aux*. Specified as: a fullword integer, where:

If *naux* = 0 and error 2015 is unrecoverable, SSPEV, DSPEV, CHPEV, and ZHPEV dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, It must have the following value, where:

For SSPEV and DSPEV:
If *iopt* = 0 or 20, *naux* ≥ *n*.
If *iopt* = 1 or 21, *naux* ≥ 2*n*.

For CHPEV and ZHPEV:
If *iopt* = 0 or 20, *naux* ≥ 3*n*.
If *iopt* = 1 or 21, *naux* ≥ 4*n*.

On Return

w

is the vector ***w*** of length *n*, containing the eigenvalues of ***A*** in ascending order. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 119 on page 710.

z

has the following meaning, where:

If *iopt* = 0 or 20, it is not used in the computation.

If *iopt* = 1 or 21, it is the matrix ***Z*** of order *n*, containing the orthonormal eigenvectors of matrix ***A***. The eigenvector in column *i* of matrix ***Z*** corresponds to the eigenvalue w_i.

Returned as: an *ldz* by (at least) *n* array, containing numbers of the data type indicated in Table 119 on page 710.

Notes

1. When you specify *iopt* = 0 or 20, you must specify:
   - A positive value for *ldz*
   - A dummy argument for *z* (see "Example 1" on page 714)
2. The following items must have no common elements: matrix ***A***, matrix ***Z***, vector ***w***, and the data area specified for *aux*; otherwise, results are unpredictable. See "Concepts" on page 55.
3. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values.
4. For a description of how real symmetric matrices are stored in lower- or upper-packed storage mode, see "Lower-Packed Storage Mode" on page 66 or "Upper-Packed Storage Mode" on page 67, respectively.

   For a description of how complex Hermitian matrices are stored in lower- or upper-packed storage mode, see "Complex Hermitian Matrix" on page 70.
5. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: The next two sections describe the methods used to compute the eigenvalues and, optionally, the eigenvectors for either a real symmetric matrix or a complex Hermitian matrix. For more information on these methods, see references [39], [43], [60], [81], [91], and [93]. If *n* is 0, no computation is performed. The results of the computations using short- and long-precision data can vary in accuracy. See "Example 3" on page 715 and "Example 4" on page 716 for an illustration of the difference in results. Eigenvalues computed using equivalent *iopt* values are mathematically equivalent, but are not guaranteed to be bitwise identical. For example, the results computed using *iopt* = 0 and *iopt* = 20 are mathematically equivalent, but are not necessarily bitwise identical.

These algorithms have a tendency to generate underflows that may hurt overall performance. The system default is to mask underflow, which improves the performance of these subroutines.

Real Symmetric Matrix: The eigenvalues and, optionally, the eigenvectors of a real symmetric matrix ***A*** are computed as follows:

- For *iopt* = 0 or 20, the eigenvalues are computed as follows:
  1. Reduce the real symmetric matrix ***A*** to a real symmetric tridiagonal matrix using orthogonal similarity transformations.
  2. Compute the eigenvalues of the real symmetric tridiagonal matrix using the implicit QL algorithm.
  3. The eigenvalues are ordered and returned in vector ***w***.
- For *iopt* = 1 or 21, the eigenvalues and eigenvectors are computed as follows:

1. Reduce the real symmetric matrix ***A*** to a real symmetric tridiagonal matrix using and accumulating orthogonal similarity transformations.
2. Compute the eigenvalues of the real symmetric tridiagonal matrix and the eigenvectors of the real symmetric matrix using the implicit QL algorithm.
3. The eigenvalues are ordered and returned in vector ***w***, and the corresponding eigenvectors are returned in matrix **Z**.

Complex Hermitian Matrix: The eigenvalues and, optionally, the eigenvectors of a complex Hermitian matrix ***A*** are computed as follows:

- For *iopt* = 0 or 20, the eigenvalues are computed as follows:
  1. Reduce the complex Hermitian matrix ***A*** to a real symmetric tridiagonal matrix using unitary similarity transformations.
  2. Compute the eigenvalues of the real symmetric tridiagonal matrix using the implicit QL algorithm.
  3. The eigenvalues are ordered and returned in vector ***w***.
- For *iopt* = 1 or 21, the eigenvalues and eigenvectors are computed as follows:
  1. Reduce the complex Hermitian matrix ***A*** to a real symmetric tridiagonal matrix using unitary similarity transformations.
  2. Compute the eigenvalues and eigenvectors of the real symmetric tridiagonal matrix using the implicit QL algorithm.
  3. Back transform the eigenvectors of the real symmetric tridiagonal matrix to those of the original complex Hermitian matrix.
  4. The eigenvalues are ordered and returned in vector ***w***, and the corresponding eigenvectors are returned in matrix **Z**.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors: Eigenvalue (*i*) failed to converge after (*xxx*) iterations:

- The eigenvalues (w_j, j = 1, 2, ..., i–1) are correct, but are unordered.
- If *iopt* = 1 or 21, then *z* is modified, but no eigenvectors are correct.
- If *iopt* = 0 or 20 for SSPEV and DSPEV, *ap* is modified.
- For CHPEV and ZHPEV, *ap* is modified.
- The return code is set to 1.
- *i* and *xxx* can be determined at run time by use of the ESSL error-handling-facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2101 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. See "What Can You Do about ESSL Computational Errors?" on page 48.

Input-Argument Errors

1. *iopt* ≠ 0, 1, 20, or 21
2. *n* < 0
3. *ldz* ≤ 0 and *iopt* = 1 or 21
4. *n* > *ldz* and *iopt* = 1 or 21
5. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 2 is returned if error 2015 is recoverable.

Example 1: This example shows how to find the eigenvalues only of a real short-precision symmetric matrix ***A*** of order 3, stored in lower-packed storage mode Matrix ***A*** is:

```
⎡                  ⎤
⎢  1.0  -1.0   0.0 ⎥
⎢ -1.0   2.0  -1.0 ⎥
⎢  0.0  -1.0   1.0 ⎥
⎣                  ⎦
```

where:

- NAUX is equal to N.
- AUX contains N elements.
- LDZ is set to 1 to avoid an error condition.
- DUMMY is used as a placeholder for argument *z*, which is not used in the computation.
- On output, AP has been overwritten.

Call Statement and Input

```
           IOPT  AP   W     Z    LDZ  N   AUZ  NAUX
            |    |    |     |     |   |    |    |
CALL SSPEV( 0  , AP , W , DUMMY , 1 , 3 , AUX , 3  )

AP       =  (1.0, -1.0, 0.0, 2.0, -1.0, 1.0)
```

Output

```
        ⎡          ⎤
        ⎢ 0.000000 ⎥
W    =  ⎢ 1.000000 ⎥
        ⎢ 3.000000 ⎥
        ⎣          ⎦
```

Example 2: This example shows how to find the eigenvalues and eigenvectors of a real short-precision symmetric matrix ***A*** of order 4, stored in upper-packed storage mode. Matrix ***A*** is:

```
⎡                    ⎤
⎢ 5.0  4.0  1.0  1.0 ⎥
⎢ 4.0  5.0  1.0  1.0 ⎥
⎢ 1.0  1.0  4.0  2.0 ⎥
⎢ 1.0  1.0  2.0  4.0 ⎥
⎣                    ⎦
```

where:

- NAUX is equal to 2N.
- AUX contains 2N elements.

Note: This matrix is used in Example 4.1 in referenced text [60].

Call Statement and Input

```
           IOPT  AP   W   Z  LDZ  N   AUZ  NAUX
            |    |    |   |   |   |    |    |
CALL SSPEV( 21 , AP , W , Z , 4 , 4 , AUX , 8  )

AP       =  (5.0, 4.0, 5.0, 1.0, 1.0, 4.0, 1.0, 1.0, 2.0, 4.0)
```

Output

```
          ┌          ┐
          │ 1.000000 │
W     =   │ 2.000000 │
          │ 5.000000 │
          │ 9.999999 │
          └          ┘

          ┌                                      ┐
          │  0.707107   0.000000   0.316227  0.632455 │
Z     =   │ -0.707107   0.000000   0.316228  0.632455 │
          │  0.000000  -0.707106  -0.632455  0.316227 │
          │  0.000000   0.707107  -0.632455  0.316228 │
          └                                      ┘
```

Example 3: This example shows how to find the eigenvalues and eigenvectors of a real short-precision symmetric matrix ***A***, stored in lower-packed storage mode, which has an eigenvalue of multiplicity 2. Matrix ***A*** is:

```
          ┌                  ┐
          │ 6.0  4.0  4.0  1.0 │
          │ 4.0  6.0  1.0  4.0 │
          │ 4.0  1.0  6.0  4.0 │
          │ 1.0  4.0  4.0  6.0 │
          └                  ┘
```

where:

- NAUX is equal to 2N.
- AUX contains 2N elements.

Note: This matrix is used in Example 4.2 in referenced text [60].

Call Statement and Input

```
           IOPT  AP   W   Z  LDZ  N   AUZ  NAUX
            |    |    |   |   |   |    |    |
CALL SSPEV( 1  , AP , W , Z , 7 , 4 , AUX , 8  )

AP       =  (6.0, 4.0, 4.0, 1.0, 6.0, 1.0, 4.0, 6.0, 4.0, 6.0)
```

Output

```
          ┌          ┐
          │ -1.000000 │
W     =   │  4.999999 │
          │  5.000000 │
          │ 15.000000 │
          └          ┘

          ┌                                      ┐
          │ -0.500000   0.000000   0.707107  0.500000 │
          │  0.500000   0.707107   0.000000  0.500000 │
          │  0.500000  -0.707107   0.000000  0.500000 │
Z     =   │ -0.500000   0.000000  -0.707107  0.500000 │
          │    .          .          .         .      │
          │    .          .          .         .      │
          │    .          .          .         .      │
          └                                      ┘
```

Example 4: This example shows how the results of Example 3 differ if matrix ***A*** is a real long-precision symmetric matrix.

Call Statement and Input

```
           IOPT  AP   W   Z  LDZ  N   AUZ  NAUX
            |    |    |   |   |   |    |    |
CALL DSPEV( 1  , AP , W , Z , 7 , 4 , AUX , 8  )
```

Output

```
        ┌          ┐
        │ -1.000000 │
W    =  │  5.000000 │
        │  5.000000 │
        │ 15.000000 │
        └          ┘

        ┌                                      ┐
        │ -0.500000  -0.216773  -0.673060  0.500000 │
        │  0.500000   0.673060  -0.216773  0.500000 │
        │  0.500000  -0.673060   0.216773  0.500000 │
Z    =  │ -0.500000   0.216773   0.673060  0.500000 │
        │    .          .          .         .      │
        │    .          .          .         .      │
        │    .          .          .         .      │
        └                                      ┘
```

Example 5: This example shows how to find the eigenvalues and eigenvectors of a complex Hermitian matrix ***A*** of order 2, stored in lower-packed storage mode. Matrix ***A*** is:

```
┌                        ┐
│ (1.0, 0.0)  (0.0, -1.0) │
│ (0.0, 1.0)  (1.0,  0.0) │
└                        ┘
```

where:

- NAUX is equal to 4N.
- AUX contains 4N elements.
- On output, AP has been overwritten.

Note: This matrix is used in Example 6.1 in referenced text [60].

Call Statement and Input

```
           IOPT  AP   W   Z  LDZ  N   AUZ  NAUX
            |    |    |   |   |   |    |    |
CALL ZHPEV( 1  , AP , W , Z , 2 , 2 , AUX , 8  )

AP       =  ((1.0, . ), (0.0, 1.0), (1.0, . ))
```

Output

```
        ┌          ┐
W    =  │ 0.000000 │
        │ 2.000000 │
        └          ┘
```

```
                                                                  ┌                                                  ┐
Z    =                                                            │  (0.000000, -0.707107)  (0.000000, -0.707107) │
                                                                  │ (-0.707107,  0.000000)  (0.707107,  0.000000) │
                                                                  └                                                  ┘
```

Example 6: This example shows how to find the eigenvalues only of a complex Hermitian matrix ***A*** of order 4, stored in upper-packed storage mode. Matrix ***A*** is:

```
                                                                  ┌                                                  ┐
                                                                  │ (3.0,  0.0)  (1.0, 0.0)  (0.0,  0.0)  (0.0, 2.0) │
                                                                  │ (1.0,  0.0)  (3.0, 0.0)  (0.0, -2.0)  (0.0, 0.0) │
                                                                  │ (0.0,  0.0)  (0.0, 2.0)  (1.0,  0.0)  (1.0, 0.0) │
                                                                  │ (0.0, -2.0)  (0.0, 0.0)  (1.0,  0.0)  (1.0, 0.0) │
                                                                  └                                                  ┘
```

where:

- NAUX is equal to 3N.
- AUX contains 3N elements.
- LDZ is set to 1 to avoid an error condition.
- DUMMY is used as a placeholder for argument *z*, which is not used in the computation.
- On output, AP has been overwritten.

Note: This matrix is used in Example 6.6 in referenced text [60].

Call Statement and Input

```
           IOPT  AP   W     Z     LDZ  N   AUZ  NAUX
            |    |    |     |     |    |    |    |
CALL ZHPEV( 20 , AP , W , DUMMY , 1 , 4 , AUX , 12 )

AP       =  ((3.0, . ), (1.0, 0.0), (3.0, . ), (0.0, 0.0),
             (0.0, -2.0), (1.0, . ), (0.0, 2.0), (0.0, 0.0),
             (1.0, 0.0), (1.0, . ))
```

Output

```
                  ┌          ┐
                  │ -0.828427 │
W    =            │  0.000000 │
                  │  4.000000 │
                  │  4.828427 │
                  └          ┘
```

Example 7: This example shows how to find the eigenvalues and eigenvectors of a complex Hermitian matrix ***A*** of order 2, stored in lower-packed storage mode. Matrix ***A*** is:

```
                                    ┌                          ┐
                                    │ (1.0, 0.0)  (1.0, -1.0) │
                                    │ (1.0, 1.0)  (1.0,  0.0) │
                                    └                          ┘
```

where:

- NAUX is equal to 4N.
- AUX contains 4N elements.
- On output, AP has been overwritten.

Note: This matrix is used in Example 6.2 in referenced text [60].

Call Statement and Input

```
          IOPT  AP   W   Z  LDZ  N   AUZ  NAUX
           |    |    |   |   |   |    |    |
CALL ZHPEV( 1  , AP , W , Z , 2 , 2 , AUX , 8  )

AP       =  ((1.0, . ), (1.0, 1.0), (1.0, . ))
```

Output

```
       ┌           ┐
W    = │ -0.414214 │
       │  2.414214 │
       └           ┘

       ┌                                                  ┐
Z    = │  (0.500000, -0.500000)  (0.500000, -0.500000) │
       │ (-0.707107,  0.000000)  (0.707107,  0.000000) │
       └                                                  ┘
```

SSPSV, DSPSV, CHPSV, and ZHPSV—Extreme Eigenvalues and, Optionally, the Eigenvectors of a Real Symmetric Matrix or a Complex Hermitian Matrix

SSPSV and DSPSV compute the extreme eigenvalues and, optionally, the eigenvectors of real symmetric matrix ***A***, stored in lower- or upper-packed storage mode. CHPSV and ZHPSV compute the extreme eigenvalues and, optionally, the eigenvectors of complex Hermitian matrix ***A***, stored in lower- or upper-packed storage mode. The extreme eigenvalues are returned in vector ***w***, and the corresponding eigenvectors are returned in matrix ***Z***:

$$Az = wz$$

where $A = A^{T}$ or $A = A^{H}$.

Table 120. Data Types

A, *z*	***w***, *aux*	**Subroutine**
Short-precision real	Short-precision real	SSPSV
Long-precision real	Long-precision real	DSPSV
Short-precision complex	Short-precision real	CHPSV
Long-precision complex	Long-precision real	ZHPSV

Note: If you want to compute 10% or fewer eigenvalues only, or you want to compute 30% or fewer eigenvalues and eigenvectors, you get better performance if you use _SPSV and _HPSV instead of _SPEV and _HPEV, respectively. For all other uses, you should use _SPEV and _HPEV.

Syntax

Fortran	CALL SSPSV | DSPSV | CHPSV | ZHPSV (*iopt*, *ap*, *w*, *z*, *ldz*, *n*, *m*, *aux*, *naux*)
C and C++	sspsv | dspsv | chpsv | zhpsv (*iopt*, *ap*, *w*, *z*, *ldz*, *n*, *m*, *aux*, *naux*);
PL/I	CALL SSPSV | DSPSV | CHPSV | ZHPSV (*iopt*, *ap*, *w*, *z*, *ldz*, *n*, *m*, *aux*, *naux*);

On Entry

iopt

indicates the type of computation to be performed, where:

If *iopt* = 0 or 20, the *m* smallest eigenvalues only are computed.

If *iopt* = 1 or 21, the *m* smallest eigenvalues and the eigenvectors are computed.

If *iopt* = 10 or 30, the *m* largest eigenvalues only are computed.

If *iopt* = 11 or 31, the *m* largest eigenvalues and the eigenvectors are computed.

Specified as: a fullword integer; *iopt* = 0, 1, 10, 11, 20, 21, 30, or 31.

ap

is the real symmetric or complex Hermitian matrix ***A*** of order *n*, whose *m* smallest or largest eigenvalues and, optionally, the corresponding eigenvectors are computed. It is stored in an array, referred to as AP, where:

If *iopt* = 0, 1, 10, or 11, it is stored in lower-packed storage mode.

If *iopt* = 20, 21, 30, or 31, it is stored in upper-packed storage mode.

Specified as: a one-dimensional array of (at least) length $n(n+1)/2$, containing numbers of the data type indicated in Table 120 on page 719. On output, AP is overwritten; that is, the original input is not preserved.

w

See On Return.

z

See On Return.

ldz

has the following meaning, where:

If *iopt* = 0, 10, 20, or 30, it is not used in the computation.

If *iopt* = 1, 11, 21, or 31, it is the leading dimension of the output array specified for *z*.

Specified as: a fullword integer. It must have the following value, where:

If *iopt* = 0, 10, 20, or 30, $ldz > 0$.

If *iopt* = 1, 11, 21, or 31, $ldz > 0$ and $ldz \geq n$.

n

is the order of matrix ***A***. Specified as: a fullword integer; $n \geq 0$.

m

is the number of eigenvalues and, optionally, eigenvectors to be computed. Specified as: a fullword integer; $0 \leq m \leq n$.

aux

has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is a storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing numbers of the data type indicated in Table 120 on page 719. On output, the contents are overwritten.

naux

is the size of the work area specified by *aux*—that is, the number of elements in *aux*. Specified as: a fullword integer, where:

If *naux* = 0 and error 2015 is unrecoverable, SSPSV, DSPSV, CHPSV, and ZHPSV dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, It must have the following value, where:

For SSPSV and DSPSV:

If *iopt* = 0, 10, 20, or 30, $naux \geq 3n$.
If *iopt* = 1, 11, 21, or 31, $naux \geq 9n$.

For CHPSV and ZHPSV:

If *iopt* = 0, 10, 20, or 30, $naux \geq 5n$.
If *iopt* = 1, 11, 21, or 31, $naux \geq 11n$.

On Return

w

is the vector ***w*** of length *n*, containing in the first *m* positions of either the *m* smallest eigenvalues of ***A*** in ascending order or the *m* largest eigenvalues of ***A*** in descending order.

Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 120 on page 719.

z

has the following meaning, where:

If *iopt* = 0, 10, 20, or 30, it is not used in the computation.

If *iopt* = 1, 11, 21, or 31, it is the *n* by *m* matrix ***Z***, containing *m* orthonormal eigenvectors of matrix ***A***. The eigenvector in column *i* of matrix ***Z*** corresponds to the eigenvalue w_i.

Returned as: an *ldz* by (at least) *m* array, containing numbers of the data type indicated in Table 120 on page 719.

Notes

1. When you specify *iopt* = 0, 10, 20, or 30, you must specify:
   - A positive value for *ldz*
   - A dummy argument for *z* (see "Example 4" on page 725)
2. The following items must have no common elements: matrix ***A***, matrix ***Z***, vector ***w***, and the data area specified for *aux*; otherwise, results are unpredictable. See "Concepts" on page 55.
3. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix ***A*** are assumed to be zero, so you do not have to set these values.
4. For a description of how real symmetric matrices are stored in lower- or upper-packed storage mode, see "Lower-Packed Storage Mode" on page 66 or "Upper-Packed Storage Mode" on page 67, respectively.

   For a description of how complex Hermitian matrices are stored in lower- or upper-packed storage mode, see "Complex Hermitian Matrix" on page 70.
5. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: The methods used to compute the extreme eigenvalues and, optionally, the eigenvectors for either a real symmetric matrix or a complex Hermitian matrix are described in the steps below. For more information on these methods, see references [39], [43], [60], [81], [91], and [93]. If *n* or *m* is 0, no computation is performed. The results of the computations using short- and long-precision data can vary in accuracy. Eigenvalues computed using equivalent *iopt* values are mathematically equivalent, but are not guaranteed to be bitwise identical. For example, the results computed using *iopt* = 0 and *iopt* = 20 are mathematically equivalent, but are not necessarily bitwise identical.

These algorithms have a tendency to generate underflows that may hurt overall performance. The system default is to mask underflow, which improves the performance of these subroutines.

The extreme eigenvalues and, optionally, the eigenvectors of a real symmetric matrix ***A*** or complex Hermitian matrix ***A*** are computed as follows:

- For *iopt* = 0, 10, 20, or 30, the eigenvalues are computed as follows:
  1. Reduce the real symmetric matrix ***A*** (for SSPSV and DSPSV) or complex Hermitian matrix ***A*** (for CHPSV and ZHPSV) to a real symmetric tridiagonal matrix using orthogonal similarity transformations (for SSPSV and DSPSV) or unitary similarity transforms (for CHPSV and ZHPSV).
  2. Compute the *m* smallest eigenvalues or *m* largest eigenvalues of the real symmetric tridiagonal matrix using a rational variant of the QR method with Newton corrections.
  3. The eigenvalues are returned in vector ***w*** in the first *m* positions, where the *m* smallest are placed in ascending order, or the *m* largest are placed in descending order.
- For *iopt* = 1, 11, 21, or 31, the eigenvalues and eigenvectors are computed as follows:
  1. Reduce the real symmetric matrix ***A*** (for SSPSV and DSPSV) or complex Hermitian matrix ***A*** (for CHPSV and ZHPSV) to a real symmetric tridiagonal matrix using orthogonal similarity transformations (for SSPSV and DSPSV) or unitary similarity transforms (for CHPSV and ZHPSV).
  2. Compute the *m* smallest eigenvalues or *m* largest eigenvalues of the real symmetric tridiagonal matrix using a rational variant of the QR method with Newton corrections.
  3. Compute the corresponding eigenvectors of the real symmetric tridiagonal matrix using inverse iteration.
  4. Back transform the eigenvectors of the real symmetric tridiagonal matrix to those of the original matrix.
  5. The eigenvalues are returned in vector ***w*** in the first *m* positions, where the *m* smallest are placed in ascending order, or the *m* largest are placed in descending order. The corresponding eigenvectors are returned in matrix **Z**.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors

1. Eigenvalue (*i*) failed to converge after (*xxx*) iterations. (The computational error message may occur multiple times with processing continuing after each error, because the number of allowable errors for error code 2114 is set to be unlimited in the ESSL error option table.)
   - The eigenvalue, w_i, is the best estimate obtained. Any eigenvalues for which this message has not been issued are correct.
   - *ap* is modified.
   - The return code is set to 1.
   - *i* and *xxx* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error 2199 in the ESSL error option table. See "What Can You Do about ESSL Computational Errors?" on page 48.
2. Eigenvector (*i*) failed to converge after (*xxx*) iterations. (The computational error message may occur multiple times with processing continuing after each error,

because the number of allowable errors for error code 2102 is set to be unlimited in the ESSL error option table.)

- All eigenvalues are correct.
- The eigenvector that failed to converge is set to zero; however, any selected eigenvectors for which this message is not issued are correct.
- *ap* is modified.
- The return code is set to 2.
- *i* and *xxx* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error 2199 in the ESSL error option table. See "What Can You Do about ESSL Computational Errors?" on page 48.

Input-Argument Errors

1. *iopt* ≠ 0, 1, 10, 11, 20, 21, 30, or 31
2. $n < 0$
3. $m < 0$
4. $m > n$
5. *ldz* ≤ 0 and *iopt* = 1, 11, 21, or 31
6. $n >$ *ldz* and *iopt* = 1, 11, 21, or 31
7. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 3 is returned if error 2015 is recoverable.

Example 1: This example shows how to find the two smallest eigenvalues and corresponding eigenvectors of a real long-precision symmetric matrix ***A*** of order 4, stored in upper-packed storage mode. Matrix ***A*** is:

$$\begin{bmatrix} 5.0 & 4.0 & 1.0 & 1.0 \\ 4.0 & 5.0 & 1.0 & 1.0 \\ 1.0 & 1.0 & 4.0 & 2.0 \\ 1.0 & 1.0 & 2.0 & 4.0 \end{bmatrix}$$

where:

- NAUX is equal to 9N.
- AUX contains 9N elements.
- On output, AP has been overwritten.

Note: This matrix is used in Example 4.1 in referenced text [60].

Call Statement and Input

```
           IOPT  AP   W   Z  LDZ  N   M   AUX  NAUX
            |    |    |   |   |   |   |    |    |
CALL DSPSV( 21 , AP , W , Z , 4 , 4 , 2 , AUX , 36 )

AP       =  (5.0, 4.0, 5.0, 1.0, 1.0, 4.0, 1.0, 1.0, 2.0, 4.0)
```

Output

```
        ⎡          ⎤
        ⎢ 1.000000 ⎥
W    =  ⎢ 2.000000 ⎥
        ⎢  .       ⎥
        ⎢  .       ⎥
        ⎣          ⎦
```

```
          ┌                    ┐
          │ -0.707107   0.000000 │
Z    =    │  0.707107   0.000000 │
          │  0.000000  -0.707107 │
          │  0.000000   0.707107 │
          └                    ┘
```

Example 2: This example shows how to find the three largest eigenvalues and corresponding eigenvectors of a real long-precision symmetric matrix ***A*** of order 4, stored in lower-packed storage mode, having an eigenvalue of multiplicity two. Matrix ***A*** is:

```
          ┌                    ┐
          │ 6.0  4.0  4.0  1.0 │
          │ 4.0  6.0  1.0  4.0 │
          │ 4.0  1.0  6.0  4.0 │
          │ 1.0  4.0  4.0  6.0 │
          └                    ┘
```

where:

- NAUX is equal to 9N.
- AUX contains 9N elements.
- On output, AP has been overwritten.

Note: This matrix is used in Example 4.2 in referenced text [60].

Call Statement and Input

```
           IOPT  AP   W   Z  LDZ  N   M   AUX  NAUX
            |    |    |   |   |   |   |    |    |
CALL DSPSV( 11 , AP , W , Z , 8 , 4 , 3 , AUX , 36 )

AP       =  (6.0, 4.0, 4.0, 1.0, 6.0, 1.0, 4.0, 6.0, 4.0, 6.0)
```

Output

```
          ┌           ┐
          │ 15.000000 │
W    =    │  5.000000 │
          │  5.000000 │
          │   .       │
          └           ┘

          ┌                     ┐
          │  0.500000   0.707107   0.000000 │
          │  0.500000   0.000000  -0.707107 │
          │  0.500000   0.000000   0.707107 │
Z    =    │  0.500000  -0.707107   0.000000 │
          │   .          .          .       │
          │   .          .          .       │
          │   .          .          .       │
          │   .          .          .       │
          └                     ┘
```

Example 3: This example shows how to find the largest eigenvalue and the corresponding eigenvector of a complex Hermitian matrix ***A*** of order 2, stored in lower-packed storage mode. Matrix ***A*** is:

```
⎡                          ⎤
⎢ (1.0, 0.0)  (0.0, -1.0) ⎥
⎢ (0.0, 1.0)  (1.0,  0.0) ⎥
⎣                          ⎦
```

where:

- NAUX is equal to 11N.
- AUX contains 11N elements.
- On output, AP has been overwritten.

Note: This matrix is used in Example 6.1 in referenced text [60].

Call Statement and Input

```
           IOPT  AP   W   Z  LDZ  N   M   AUX  NAUX
            |    |    |   |   |   |   |    |    |
CALL ZHPSV( 11 , AP , W , Z , 2 , 2 , 1 , AUX , 22 )

AP       =  ((1.0, . ), (0.0, 1.0), (1.0, . ))
```

Output

```
W    =  ⎡          ⎤
       ⎢ 2.000000 ⎥
       ⎢  .       ⎥
       ⎣          ⎦

Z    =  ⎡                        ⎤
       ⎢ (0.000000, -0.707107) ⎥
       ⎢ (0.707107,  0.000000) ⎥
       ⎣                        ⎦
```

Example 4: This example shows how to find the two smallest eigenvalues only of a complex Hermitian matrix ***A*** of order 4, stored in upper-packed storage mode. Matrix ***A*** is:

```
⎡                                                          ⎤
⎢ (3.0,  0.0)  (1.0, 0.0)  (0.0,  0.0)  (0.0, 2.0) ⎥
⎢ (1.0,  0.0)  (3.0, 0.0)  (0.0, -2.0)  (0.0, 0.0) ⎥
⎢ (0.0,  0.0)  (0.0, 2.0)  (1.0,  0.0)  (1.0, 0.0) ⎥
⎢ (0.0, -2.0)  (0.0, 0.0)  (1.0,  0.0)  (1.0, 0.0) ⎥
⎣                                                          ⎦
```

where:

- NAUX is equal to 5N.
- AUX contains 5N elements.
- LDZ is set to 1 to avoid an error condition.
- DUMMY is used as a placeholder for argument *z*, which is not used in the computation.
- On output, AP has been overwritten.

Note: This matrix is used in Example 6.6 in referenced text [60].

Call Statement and Input

```
           IOPT  AP   W     Z    LDZ  N   M   AUX  NAUX
            |    |    |     |     |   |   |    |    |
CALL ZHPSV( 20 , AP , W , DUMMY , 1 , 4 , 2 , AUX , 20 )

AP       =  ((3.0, . ), (1.0, 0.0), (3.0, . ), (0.0, 0.0),
             (0.0, -2.0), (1.0, . ), (0.0, 2.0), (0.0, 0.0),
             (1.0, 0.0), (1.0, . ))
```

Output

```
         ⌈          ⌉
         | -0.828427 |
W    =   |  0.000000 |
         |    .      |
         |    .      |
         ⌊          ⌋
```

SGEGV and DGEGV—Eigenvalues and, Optionally, the Eigenvectors of a Generalized Real Eigensystem, Az=wBz, where A and B Are Real General Matrices

These subroutines compute the eigenvalues and, optionally, the eigenvectors of a generalized real eigensystem, where ***A*** and ***B*** are real general matrices. Eigenvalues ***w*** are based on the two parts returned in vectors α and β, such that $w_i = \alpha_i/\beta_i$ for $\beta_i \neq 0$, and $w_i = \infty$ for $\beta_i = 0$. Eigenvectors are returned in matrix ***Z***:

$$\boldsymbol{Az} = w\boldsymbol{Bz}$$

Table 121. Data Types

A, ***B***, β, *aux*	α, ***Z***	**Subroutine**
Short-precision real	Short-precision complex	SGEGV
Long-precision real	Long-precision complex	DGEGV

Syntax

Fortran	CALL SGEGV | DGEGV (*iopt*, *a*, *lda*, *b*, *ldb*, *alpha*, *beta*, *z*, *ldz*, *n*, *aux*, *naux*)
C and C++	sgegv | dgegv (*iopt*, *a*, *lda*, *b*, *ldb*, *alpha*, *beta*, *z*, *ldz*, *n*, *aux*, *naux*);
PL/I	CALL SGEGV | DGEGV (*iopt*, *a*, *lda*, *b*, *ldb*, *alpha*, *beta*, *z*, *ldz*, *n*, *aux*, *naux*);

On Entry

iopt
indicates the type of computation to be performed, where:

If *iopt* = 0, eigenvalues only are computed.

If *iopt* = 1, eigenvalues and eigenvectors are computed.

Specified as: a fullword integer; *iopt* = 0 or 1.

a
is the real general matrix ***A*** of order *n*. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 121. On output, `A` is overwritten; that is, the original input is not preserved.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; *lda* > 0 and *lda* ≥ *n*.

b
is the real general matrix ***B*** of order *n*. Specified as: an *ldb* by (at least) *n* array, containing numbers of the data type indicated in Table 121. On output, `B` is overwritten; that is, the original input is not preserved.

ldb
is the leading dimension of the array specified for *b*. Specified as: a fullword integer; *ldb* > 0 and *ldb* ≥ *n*.

alpha
See On Return.

beta
See On Return.

z
See On Return.

ldz
has the following meaning, where:

If *iopt* = 0, it is not used in the computation.

If *iopt* = 1, it is the leading dimension of the output array specified for *z*.

Specified as: a fullword integer. It must have the following value, where:

If *iopt* = 0, *ldz* > 0.

If *iopt* = 1, *ldz* > 0 and *ldz* ≥ *n*.

n
is the order of matrices ***A*** and ***B***. Specified as: a fullword integer; *n* ≥ 0.

aux
has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is a storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing numbers of the data type indicated in Table 121 on page 727. On output, the contents are overwritten.

naux
is the size of the work area specified by *aux*—that is, the number of elements in *aux*. Specified as: a fullword integer, where:

If *naux* = 0 and error 2015 is unrecoverable, SGEGV and DGEGV dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux* ≥ 3*n*.

On Return

alpha
is the vector α of length *n*, containing the numerators of the eigenvalues of the generalized real eigensystem $\boldsymbol{Az} = w\boldsymbol{Bz}$. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 121 on page 727.

beta
is the vector β of length *n*, containing the denominators of the eigenvalues of the generalized real eigensystem $\boldsymbol{Az} = w\boldsymbol{Bz}$. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 121 on page 727.

z
has the following meaning, where:

If *iopt* = 0, it is not used in the computation.

If *iopt* = 1, it is the matrix ***Z*** of order *n*, containing the eigenvectors of the generalized real eigensystem, $\boldsymbol{Az} = w\boldsymbol{Bz}$. The eigenvector in column *i* of matrix ***Z*** corresponds to the eigenvalue w_i, computed using the α_i and β_i values. Each eigenvector is normalized so that the modulus of its largest element is 1.

Returned as: an *ldz* by (at least) *n* array, containing numbers of the data type indicated in Table 121 on page 727.

Notes

1. When you specify *iopt* = 0, you must specify:
   - A positive value for *ldz*
   - A dummy argument for *z* (see "Example 1" on page 730)
2. Matrices ***A***, ***B***, and ***Z***, vectors α and β, and the work area specified for *aux* must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.
3. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: The following steps describe the methods used to compute the eigenvalues and, optionally, the eigenvectors of a generalized real eigensystem, $\boldsymbol{Az} = w\boldsymbol{Bz}$, where ***A*** and ***B*** are real general matrices. The methods are based upon Moler and Stewart's QZ algorithm. You must calculate the resulting eigenvalues ***w*** based on the two parts returned in vectors α and β. Each eigenvalue is calculated as follows: $w_i = \alpha_i/\beta_i$ for $\beta_i \neq 0$ and $w_i = \infty$ for $\beta_i = 0$. Eigenvalues are unordered, except that complex conjugate pairs appear consecutively with the eigenvalue having the positive imaginary part first.

- For *iopt* = 0, the eigenvalues are computed as follows:
  1. Simultaneously reduce ***A*** to upper Hessenberg form and ***B*** to upper triangular form using orthogonal transformations.
  2. Reduce ***A*** from upper Hessenberg form to quasi-upper triangular form while maintaining the upper triangular form of ***B*** using orthogonal transformations.
  3. Compute the eigenvalues of the generalized real eigensystem with ***A*** in quasi-upper triangular form and ***B*** in upper triangular form using orthogonal transformations.
  4. The numerators and denominators of the eigenvalues are returned in vectors α and β, respectively.
- For *iopt* = 1, the eigenvalues and eigenvectors are computed as follows:
  1. Simultaneously reduce ***A*** to upper Hessenberg form and ***B*** to upper triangular form using and accumulating orthogonal transformations.
  2. Reduce ***A*** from upper Hessenberg form to quasi-upper triangular form while maintaining the upper triangular form of ***B*** using and accumulating orthogonal transformations.
  3. Compute the eigenvalues of the generalized real eigensystem with ***A*** in quasi-upper triangular form and ***B*** in upper triangular form using and accumulating orthogonal transformations.
  4. Compute the eigenvectors of the generalized real eigensystem with ***A*** in quasi-upper triangular form and ***B*** in upper triangular form using back substitution.
  5. The numerators and denominators of the eigenvalues are returned in vectors α and β, respectively, and the eigenvectors are returned in matrix ***Z***.

For more information on these methods, see references [39], [43], [55], [77], [60], [59], [81], [91], and [93]. If *n* is 0, no computation is performed. The results of the computations using short- and long-precision data can vary in accuracy.

These algorithms have a tendency to generate underflows that may hurt overall performance. The system default is to mask underflow, which improves the performance of these subroutines.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors: Eigenvalue (*i*) failed to converge after (*xxx*) iterations:

- The eigenvalues (w_j, $j = i+1, i+2, ..., n$) are correct.
- If *iopt* = 1, then ***Z*** is modified, but no eigenvectors are correct.
- ***A*** and ***B*** have been modified.
- The return code is set to 1.
- *i* and *xxx* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2101 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. See "What Can You Do about ESSL Computational Errors?" on page 48.

Input-Argument Errors

1. *iopt* ≠ 0 or 1
2. *n* < 0
3. *lda* ≤ 0
4. *n* > *lda*
5. *ldb* ≤ 0
6. *n* > *ldb*
7. *ldz* ≤ 0 and *iopt* = 1
8. *n* > *ldz* and *iopt* = 1
9. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 2 is returned if error 2015 is recoverable.

Example 1: This example shows how to find the eigenvalues only of a real generalized eigensystem problem, ***AZ*** = ***wBZ***, where:

- `NAUX` is equal to `3N`.
- `AUX` contains `3N` elements.
- `LDZ` is set to 1 to avoid an error condition.
- `DUMMY` is used as a placeholder for argument *z*, which is not used in the computation.
- On output, matrices `A` and `B` are overwritten.

Note: These matrices are from page 257 in referenced text [59].

Call Statement and Input

```
           IOPT  A  LDA  B  LDB  ALPHA   BETA     Z    LDZ  N   AUX  NAUX
            |    |   |   |   |     |      |       |     |   |    |    |
CALL DGEGV( 0  , A , 3 , B , 3 , ALPHA , BETA , DUMMY , 1 , 3 , AUX , 9  )
```

```
        ⌈                ⌉
        | 10.0  1.0   2.0 |
A    =  |  1.0  3.0  -1.0 |
        |  1.0  1.0   2.0 |
        ⌊                ⌋
```

```
        ⌈               ⌉
        | 1.0  2.0  3.0 |
B    =  | 4.0  5.0  6.0 |
        | 7.0  8.0  9.0 |
        ⌊               ⌋
```

Output

```
          ┌                          ┐
          │  (4.778424, 0.000000) │
ALPHA =   │ (-4.760580, 0.000000) │
          │  (2.769466, 0.000000) │
          └                          ┘

          ┌          ┐
          │  0.000000 │
BETA  =   │  0.934851 │
          │ 15.446215 │
          └          ┘
```

Example 2: This example shows how to find the eigenvalues and eigenvectors of a real generalized eigensystem problem, ***AZ = wBZ***, where:

- NAUX is equal to 3N.
- AUX contains 3N elements.
- On output, matrices A and B are overwritten.

Note: These matrices are from page 263 in referenced text [59].

Call Statement and Input

```
           IOPT  A  LDA  B  LDB  ALPHA   BETA   Z  LDZ  N   AUX  NAUX
            |    |   |   |   |     |      |     |   |   |    |    |
CALL DGEGV( 1  , A , 5 , B , 5 , ALPHA , BETA , Z , 5 , 5 , AUX , 15 )

          ┌                          ┐
          │ 2.0  3.0  4.0  5.0   6.0 │
          │ 4.0  4.0  5.0  6.0   7.0 │
A     =   │ 0.0  3.0  6.0  7.0   8.0 │
          │ 0.0  0.0  2.0  8.0   9.0 │
          │ 0.0  0.0  0.0  1.0  10.0 │
          └                          ┘

          ┌                          ┐
          │ 1.0  -1.0  -1.0  -1.0  -1.0 │
          │ 0.0   1.0  -1.0  -1.0  -1.0 │
B     =   │ 0.0   0.0   1.0  -1.0  -1.0 │
          │ 0.0   0.0   0.0   1.0  -1.0 │
          │ 0.0   0.0   0.0   0.0   1.0 │
          └                          ┘
```

Output

```
          ┌                          ┐
          │  (7.950050, 0.000000) │
          │ (-0.277338, 0.000000) │
ALPHA =   │  (2.149669, 0.000000) │
          │  (6.720718, 0.000000) │
          │ (10.987556, 0.000000) │
          └                          ┘
```

```
          ┌          ┐
          │ 0.374183 │
          │ 1.480299 │
BETA  =   │ 1.636872 │
          │ 1.213574 │
          │ 0.908837 │
          └          ┘

          ┌
          │ (1.000000, 0.000000)  (-0.483408, 0.000000)   (0.540696, 0.000000)
          │ (0.565497, 0.000000)   (1.000000, 0.000000)   (0.684441, 0.000000)
Z     =   │ (0.180429, 0.000000)  (-0.661372, 0.000000)  (-1.000000, 0.000000)
          │ (0.034182, 0.000000)   (0.180646, 0.000000)   (0.363671, 0.000000)
          │ (0.003039, 0.000000)  (-0.017732, 0.000000)  (-0.041865, 0.000000)
          └

                                                                          ┐
                                (1.000000, 0.000000)  (-1.000000, 0.000000) │
                                (0.722065, 0.000000)  (-0.610415, 0.000000) │
                               (-0.089003, 0.000000)  (-0.116987, 0.000000) │
                               (-0.223599, 0.000000)   (0.038979, 0.000000) │
                                (0.050111, 0.000000)   (0.018653, 0.000000) │
                                                                          ┘
```

SSYGV and DSYGV—Eigenvalues and, Optionally, the Eigenvectors of a Generalized Real Symmetric Eigensystem, Az=wBz, where A Is Real Symmetric and B Is Real Symmetric Positive Definite

These subroutines compute the eigenvalues and, optionally, the eigenvectors of a generalized real symmetric eigensystem, where ***A*** is a real symmetric matrix, and ***B*** is a real positive definite symmetric matrix. Both ***A*** and ***B*** are stored in lower storage mode in two-dimensional arrays. Eigenvalues are returned in vector ***w***, and eigenvectors are returned in matrix ***Z***:

$$\boldsymbol{Az} = w\boldsymbol{Bz}$$

where $\boldsymbol{A} = \boldsymbol{A}^{\mathrm{T}}$, $\boldsymbol{B} = \boldsymbol{B}^{\mathrm{T}}$, and $\boldsymbol{x}^{\mathrm{T}}\boldsymbol{Bx} > 0$.

Table 122. Data Types

A, ***B***, ***w***, ***Z***, *aux*	**Subroutine**
Short-precision real	SSYGV
Long-precision real	DSYGV

Syntax

Fortran	CALL SSYGV | DSYGV (*iopt*, *a*, *lda*, *b*, *ldb*, *w*, *z*, *ldz*, *n*, *aux*, *naux*)
C and C++	ssygv | dsygv (*iopt*, *a*, *lda*, *b*, *ldb*, *w*, *z*, *ldz*, *n*, *aux*, *naux*);
PL/I	CALL SSYGV | DSYGV (*iopt*, *a*, *lda*, *b*, *ldb*, *w*, *z*, *ldz*, *n*, *aux*, *naux*);

On Entry

iopt
indicates the type of computation to be performed, where:

If *iopt* = 0, eigenvalues only are computed.

If *iopt* = 1, eigenvalues and eigenvectors are computed.

Specified as: a fullword integer; *iopt* = 0 or 1.

a
is the real symmetric matrix ***A*** of order *n*. It is stored in lower storage mode. Specified as: an *lda* by (at least) *n* array, containing numbers of the data type indicated in Table 122. On output, the data in the lower triangle of ***A*** is overwritten; that is, the original input is not preserved.

lda
is the leading dimension of the array specified for *a*. Specified as: a fullword integer; *lda* > 0 and *lda* ≥ *n*.

b
is the real positive definite symmetric matrix ***B*** of order *n*. It is stored in lower storage mode. Specified as: an *ldb* by (at least) *n* array, containing numbers of the data type indicated in Table 122. On output, the data in the lower triangle of ***B*** is overwritten; that is, the original input is not preserved.

ldb
is the leading dimension of the array specified for *b*. Specified as: a fullword integer; *ldb* > 0 and *ldb* ≥ *n*.

w
See On Return.

z
See On Return.

ldz
has the following meaning, where:

If *iopt* = 0, it is not used in the computation.

If *iopt* = 1, it is the leading dimension of the output array specified for *z*.

Specified as: a fullword integer. It must have the following value, where:

If *iopt* = 0, *ldz* > 0.

If *iopt* = 1, *ldz* > 0 and *ldz* ≥ *n*.

n
is the order of matrices ***A*** and ***B***. Specified as: a fullword integer; *n* ≥ 0.

aux
has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is a storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing numbers of the data type indicated in Table 122 on page 733. On output, the contents are overwritten.

naux
is the size of the work area specified by *aux*—that is, the number of elements in *aux*. Specified as: a fullword integer, where:

If *naux* = 0 and error 2015 is unrecoverable, SSYGV and DSYGV dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, It must have the following value, where:

If *iopt* = 0, *naux* ≥ *n*.

If *iopt* = 1, *naux* ≥ 2*n*.

On Return

w
is the vector ***w*** of length *n*, containing the eigenvalues of the generalized real symmetric eigensystem $\boldsymbol{Az} = w\boldsymbol{Bz}$ in ascending order. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 122 on page 733.

z
has the following meaning, where:

If *iopt* = 0, it is not used in the computation.

If *iopt* = 1, it is the matrix ***Z*** of order *n*, containing the eigenvectors of the generalized real symmetric eigensystem, $\boldsymbol{Az} = w\boldsymbol{Bz}$. The eigenvectors are normalized so that $\boldsymbol{Z}^{\mathrm{T}}\boldsymbol{BZ} = \boldsymbol{I}$. The eigenvector in column *i* of matrix ***Z*** corresponds to the eigenvalue w_i.

Returned as: an *ldz* by (at least) *n* array, containing numbers of the data type indicated in Table 122 on page 733.

Notes

1. When you specify *iopt* = 0, you must specify:
   - A positive value for *ldz*
   - A dummy argument for *z* (see "Example 1" on page 736)
2. Matrices ***A*** and ***Z*** may coincide. Matrices ***A*** and ***B***, vector ***w***, and the data area specified for *aux* must have no common elements; otherwise, results are unpredictable. Matrices ***Z*** and ***B***, vector ***w***, and the data area specified for *aux* must also have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.
3. For a description of how real symmetric matrices are stored in lower storage mode, see "Lower Storage Mode" on page 68.
4. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: The following steps describe the methods used to compute the eigenvalues and, optionally, the eigenvectors of a generalized real symmetric eigensystem, $\boldsymbol{Az} = w\boldsymbol{Bz}$, where ***A*** is a real symmetric matrix, and ***B*** is a real positive definite symmetric matrix. Both ***A*** and ***B*** are stored in lower storage mode in two-dimensional arrays.

1. Compute the Cholesky Decomposition of ***B***:

   $$\boldsymbol{B} = \boldsymbol{L}\boldsymbol{L}^{\mathrm{T}}$$

   For a description of methods used in this computation, see "SPPF, DPPF, SPOF, DPOF, CPOF, and ZPOF—Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization" on page 494.
2. Compute ***C***:

   $$\boldsymbol{C} = \boldsymbol{L}^{-1}\boldsymbol{A}\boldsymbol{L}^{-\mathrm{T}}$$

   In this computation, ***C*** overwrites ***A***.
3. Solve the real symmetric eigensystems analysis problem, computing the eigenvalues ***w*** and, optionally, the eigenvectors ***Y***:

   $$\boldsymbol{CY} = \boldsymbol{wY}$$

   where:

   $$\boldsymbol{Y} = \boldsymbol{L}^{\mathrm{T}}\boldsymbol{Z}$$

   For a description of the methods used for this computation, see "Real Symmetric Matrix" on page 712. In this computation, ***Y*** overwrites ***Z***.
4. If eigenvectors are requested (with *iopt* = 1), transform the eigenvectors ***Y*** into the eigenvectors ***Z*** of the original system, $\boldsymbol{Az} = w\boldsymbol{Bz}$, by solving $\boldsymbol{L}^{\mathrm{T}}\boldsymbol{Z} = \boldsymbol{Y}$ for ***Z***:

   $$\boldsymbol{Z} = \boldsymbol{L}^{-\mathrm{T}}\boldsymbol{Y}$$

For more information on these methods, see references [39], [43], [55], [60], [59], [81], [91], and [93]. If *n* is 0, no computation is performed. The results of the computations using short- and long-precision data can vary in accuracy.

These algorithms have a tendency to generate underflows that may hurt overall performance. The system default is to mask underflow, which improves the performance of these subroutines.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors

1. The ***B*** matrix is not positive definite. The leading minor of order *i* has a nonpositive determinant.
   - ***B*** is modified, but no eigenvalues or eigenvectors are correct.
   - The return code is set to 1.
   - *i* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2115 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. See "What Can You Do about ESSL Computational Errors?" on page 48.
2. Eigenvalue (*i*) failed to converge after (*xxx*) iterations:
   - The eigenvalues (w_j, j = 1, 2, ..., i–1) are correct, but are unordered.
   - If *iopt* = 1, then *z* is modified, but no eigenvectors are correct.
   - ***A*** and ***B*** have been modified.
   - The return code is set to 2.
   - *i* and *xxx* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2101 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. See "What Can You Do about ESSL Computational Errors?" on page 48.

Input-Argument Errors

1. *iopt* ≠ 0 or 1
2. *n* < 0
3. *lda* ≤ 0
4. *n* > *lda*
5. *ldb* ≤ 0
6. *n* > *ldb*
7. *ldz* ≤ 0 and *iopt* = 1
8. *n* > *ldz* and *iopt* = 1
9. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 3 is returned if error 2015 is recoverable.

Example 1: This example shows how to find the eigenvalues only of a real symmetric generalized eigensystem problem, ***AZ*** = ***wBZ***, where:

- `NAUX` is equal to `N`.
- `AUX` contains `N` elements.
- `LDZ` is set to 1 to avoid an error condition.

- DUMMY is used as a placeholder for argument *z*, which is not used in the computation.
- On output, the lower triangle of A and B is overwritten.

Note: These matrices are used in Example 8.6.2 in referenced text [59].

Call Statement and Input

```
           IOPT  A  LDA  B  LDB  W     Z    LDZ  N   AUX  NAUX
            |    |   |   |   |   |     |     |   |    |    |
CALL DSYGV( 0  , A , 2 , B , 2 , W , DUMMY , 1 , 2 , AUX , 2  )

        ⌈                ⌉
A    =  | 229.0     .   |
        | 163.0  116.0  |
        ⌊                ⌋

        ⌈              ⌉
B    =  | 81.0    .   |
        | 59.0  43.0  |
        ⌊              ⌋
```

Output

```
        ⌈            ⌉
W    =  | -0.500000  |
        |  5.000000  |
        ⌊            ⌋
```

Example 2: This example shows how to find the eigenvalues and eigenvectors of a real symmetric generalized eigensystem problem, $\boldsymbol{AZ} = \boldsymbol{wBZ}$, where:

- NAUX is equal to 2N.
- AUX contains 2N elements.
- On output, the lower triangle of A and B is overwritten.

Note: These matrices are from page 67 in referenced text [55].

Call Statement and Input

```
           IOPT  A  LDA  B  LDB  W   Z  LDZ  N   AUX  NAUX
            |    |   |   |   |   |   |   |   |    |    |
CALL DSYGV( 1  , A , 3 , B , 3 , W , Z , 3 , N , AUX , 6  )

        ⌈                  ⌉
        | -1.0    .    .   |
A    =  |  1.0   1.0   .   |
        | -1.0  -1.0  1.0  |
        ⌊                  ⌋

        ⌈                ⌉
        | 2.0   .    .   |
B    =  | 1.0  2.0   .   |
        | 0.0  1.0  2.0  |
        ⌊                ⌋
```

Output

```
        ┌          ┐
        │ -1.500000 │
W    =  │  0.000000 │
        │  2.000000 │
        └          ┘

        ┌                                    ┐
        │  0.866025   0.000000   0.000000 │
Z    =  │ -0.577350  -0.408248  -0.707107 │
        │  0.288675  -0.408248   0.707107 │
        └                                    ┘
```

Chapter 12. Fourier Transforms, Convolutions and Correlations, and Related Computations

The signal processing subroutines, provided in three areas, are described in this chapter.

Overview of the Signal Processing Subroutines

This section describes the subroutines in each of the three signal processing areas:

- Fourier transform subroutines (Table 123)
- Convolution and correlation subroutines (Table 124)
- Related-computation subroutines (Table 125)

Fourier Transforms Subroutines

The Fourier transform subroutines perform mixed-radix transforms in one, two, and three dimensions.

Table 123. List of Fourier Transform Subroutines

Descriptive Name	Short-Precision Subroutine	Long-Precision Subroutine	Page
Complex Fourier Transform	SCFT SCFTP§	DCFT	749
Real-to-Complex Fourier Transform	SRCFT	DRCFT	758
Complex-to-Real Fourier Transform	SCRFT	DCRFT	766
Cosine Transform	SCOSF SCOSFT§	DCOSF	774
Sine Transform	SSINF	DSINF	782
Complex Fourier Transform in Two Dimensions	SCFT2 SCFT2P§	DCFT2	790
Real-to-Complex Fourier Transform in Two Dimensions	SRCFT2	DRCFT2	797
Complex-to-Real Fourier Transform in Two Dimensions	SCRFT2	DCRFT2	804
Complex Fourier Transform in Three Dimensions	SCFT3 SCFT3P§	DCFT3	812
Real-to-Complex Fourier Transform in Three Dimensions	SRCFT3	DRCFT3	818
Complex-to-Real Fourier Transform in Three Dimensions	SCRFT3	DCRFT3	824

§ This subroutine is provided only for migration from earlier releases of ESSL and is not intended for use in new programs. Documentation for this subroutine is no longer provided.

© Copyright IBM Corp. 1997

Convolution and Correlation Subroutines

The convolution and correlation subroutines provide the choice of using Fourier methods or direct methods. The Fourier-method subroutines contain a high-performance mixed-radix capability. There are also several direct-method subroutines that provide decimated output.

Table 124. List of Convolution and Correlation Subroutines

Descriptive Name	Short-Precision Subroutine	Long-Precision Subroutine	Page
Convolution or Correlation of One Sequence with One or More Sequences	SCON§ SCOR§		831
Convolution or Correlation of One Sequence with Another Sequence Using a Direct Method	SCOND SCORD		837
Convolution or Correlation of One Sequence with One or More Sequences Using the Mixed-Radix Fourier Method	SCONF SCORF		843
Convolution or Correlation with Decimated Output Using a Direct Method	SDCON SDCOR	DDCON DDCOR	852
Autocorrelation of One or More Sequences	SACOR§		856
Autocorrelation of One or More Sequences Using the Mixed-Radix Fourier Method	SACORF		860

§ These subroutines are provided only for migration from earlier releases of ESSL and are not intended for use in new programs.

Related-Computation Subroutines

The related-computation subroutines consist of a group of computations that can be used in general signal processing applications. They are similar to those provided on the IBM 3838 Array Processor; however, the ESSL subroutines generally solve a wider range of problems.

Table 125. List of Related-Computation Subroutines

Descriptive Name	Short-Precision Subroutine	Long-Precision Subroutine	Page
Polynomial Evaluation	SPOLY	DPOLY	867
I-th Zero Crossing	SIZC	DIZC	870
Time-Varying Recursive Filter	STREC	DTREC	873
Quadratic Interpolation	SQINT	DQINT	876
Wiener-Levinson Filter Coefficients	SWLEV	DWLEV	880

Fourier Transforms, Convolutions, and Correlations Considerations

This section describes some global information applying to the Fourier transform, convolution, and correlation subroutines.

Use Considerations

This section provides some key points about using the Fourier transform, convolution, and correlation subroutines.

Understanding the Terminology and Conventions Used for Your Array Data

These subroutines use the term "sequences," rather than vectors and matrices, to describe the data that is stored in the arrays. The conventions used for representing sequences are defined in "Sequences" on page xxiii.

Some of the sequences used in these computations use a zero origin rather than a one-origin. For example, x_j can be expressed with $j = 0, 1, ..., n-1$ rather than $j = 1, 2, ..., n$. When using the formulas provided in this book to calculate array sizes or offsets into arrays, you need to be careful that you substitute the correct values. For example, the number of x_j elements in the sequence is n, not $n-1$.

Concerns about Lengths of Transforms

The length of the transform you can use in your program depends on the limits of the addressability of your processor.

Determining an Acceptable Length of a Transform

To determine acceptable lengths of the transforms in the Fourier transform subroutines, you have several choices. First, you can use the formula or table of values in "Acceptable Lengths for the Transforms" to choose a value. Second, ESSL's input-argument error recovery provides a means of determining an acceptable length of the transform. It uses the optionally-recoverable error 2030. For details, see "Providing a Correct Transform Length to ESSL" on page 38.

Acceptable Lengths for the Transforms

Use the following formula to determine acceptable transform lengths:

$$n = (2^h)(3^i)(5^j)(7^k)(11^m) \quad \text{for } n \leq 37748736$$

where:

$h = 1, 2, ..., 25$
$i = 0, 1, 2$
$j, k, m = 0, 1$

Figure 13 on page 742 lists all the acceptable values for transform lengths in the Fourier transform subroutines.

2	4	6	8	10	12	14	16	18
20	22	24	28	30	32	36	40	42
44	48	56	60	64	66	70	72	80
84	88	90	96	110	112	120	126	128
132	140	144	154	160	168	176	180	192
198	210	220	224	240	252	256	264	280
288	308	320	330	336	352	360	384	396
420	440	448	462	480	504	512	528	560
576	616	630	640	660	672	704	720	768
770	792	840	880	896	924	960	990	1008
1024	1056	1120	1152	1232	1260	1280	1320	1344
1386	1408	1440	1536	1540	1584	1680	1760	1792
1848	1920	1980	2016	2048	2112	2240	2304	2310
2464	2520	2560	2640	2688	2772	2816	2880	3072
3080	3168	3360	3520	3584	3696	3840	3960	4032
4096	4224	4480	4608	4620	4928	5040	5120	5280
5376	5544	5632	5760	6144	6160	6336	6720	6930
7040	7168	7392	7680	7920	8064	8192	8448	8960
9216	9240	9856	10080	10240	10560	10752	11088	11264
11520	12288	12320	12672	13440	13860	14080	14336	14784
15360	15840	16128	16384	16896	17920	18432	18480	19712
20160	20480	21120	21504	22176	22528	23040	24576	24640
25344	26880	27720	28160	28672	29568	30720	31680	32256
32768	33792	35840	36864	36960	39424	40320	40960	42240
43008	44352	45056	46080	49152	49280	50688	53760	55440
56320	57344	59136	61440	63360	64512	65536	67584	71680
73728	73920	78848	80640	81920	84480	86016	88704	90112
92160	98304	98560	101376	107520	110880	112640	114688	118272
122880	126720	129024	131072	135168	143360	147456	147840	157696
161280	163840	168960	172032	177408	180224	184320	196608	197120
202752	215040	221760	225280	229376	236544	245760	253440	258048
262144	270336	286720	294912	295680	315392	322560	327680	337920
344064	354816	360448	368640	393216	394240	405504	430080	443520
450560	458752	473088	491520	506880	516096	524288	540672	573440
589824	591360	630784	645120	655360	675840	688128	709632	720896
737280	786432	788480	811008	860160	887040	901120	917504	946176
983040	1013760	1032192	1048576	1081344	1146880	1179648	1182720	1261568
1290240	1310720	1351680	1376256	1419264	1441792	1474560	1572864	1576960
1622016	1720320	1774080	1802240	1835008	1892352	1966080	2027520	2064384
2097152	2162688	2293760	2359296	2365440	2523136	2580480	2621440	2703360
2752512	2838528	2883584	2949120	3145728	3153920	3244032	3440640	3548160
3604480	3670016	3784704	3932160	4055040	4128768	4194304	4325376	4587520
4718592	4730880	5046272	5160960	5242880	5406720	5505024	5677056	5767168
5898240	6291456	6307840	6488064	6881280	7096320	7208960	7340032	7569408
7864320	8110080	8257536	8388608	8650752	9175040	9437184	9461760	10092544
10321920	10485760	10813440	11010048	11354112	11534336	11796480	12582912	12615680
12976128	13762560	14192640	14417920	14680064	15138816	15728640	16220160	16515072
16777216	17301504	18350080	18874368	18923520	20185088	20643840	20971520	21626880
22020096	22708224	23068672	23592960	25165824	25231360	25952256	27525120	28385280
28835840	29360128	30277632	31457280	32440320	33030144	33554432	34603008	36700160
37748736								

Figure 13. Table of Acceptable Lengths for the Transforms

Understanding Auxiliary Working Storage Requirements

Auxiliary working storage is required by the Fourier transform subroutines and by the SCONF, SCORF, and SACORF subroutines. This storage is provided through the calling sequence arguments *aux*, *aux1*, and *aux2*. The sizes of these storage areas are specified by the calling sequence arguments *naux*, *naux1*, and *naux2*, respectively.

AUX1: The *aux1* array is used for storing tables and other parameters when you call a Fourier transform, convolution, or correlation subroutine for initialization with *init* = 1. The initialized *aux1* array is then used on succeeding calls with *init* = 0, when the computation is actually done. You should not use this array between the initialization and the computation.

AUX and AUX2: The *aux* and *aux2* arrays are used for temporary storage during the running of the subroutine and are available for use by your program between calls to the subroutine.

AUX3: The *aux3* argument is provided for migration purposes only and is ignored.

Initializing Auxiliary Working Storage

In many of those subroutines requiring *aux1* auxiliary working storage, two invocations of the subroutines are necessary. The first invocation initializes the working storage in *aux1* for the subroutine, and the second performs the computations. (For an explanation of auxiliary working storage, see Understanding Auxiliary Working Storage Requirements.) As a result, the working storage in *aux1* should not be used by the calling program between the two calls to the subroutine. However, it can be reused after intervening calls to the subroutine with different arguments.

If you plan to repeat a computation many times using the same set of arguments, you only need to do one initialization of the *aux1* array; that is, the initialized *aux1* array can be saved and reused as many times as needed for the computation.

If you plan to perform different computations, with different sets of arguments (except for input argument *x*), you need to do an initialization for each different computation; that is, you initialize the various *aux1* arrays for use with the different computations, saving and reusing them until they are not needed any more.

Determining the Amount of Auxiliary Working Storage That You Need

To determine the size of auxiliary storage, you have several choices. First, you can use the formulas provided in each subroutine description. Second, ESSL's input-argument error recovery provides a means of determining the minimum size you need for auxiliary storage. It uses the optionally-recoverable error 2015. For details, see “Using Auxiliary Storage in ESSL” on page 31. Third, you can have ESSL dynamically allocate *aux* and *aux2*. For details, see “Dynamic Allocation of Auxiliary Storage” on page 32.

Performance and Accuracy Considerations

The following sections explain the performance and accuracy considerations for the Fourier transforms, convolution, and correlation subroutines. For further details about performance and accuracy, see Chapter 2 on page 25.

When Running on the Workstation Processors

There are ESSL-specific rules that apply to the results of computations on the workstation processors using the ANSI/IEEE standards. For details, see "What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?" on page 44.

Defining Arrays

The stride arguments, *inc1h, inc1x, inc1y, inc2x, inc2y*, *inc3x*, and *inc3y*, provide great flexibility in defining the input and output data arrays. The arrangement of data in storage, however, can have an effect upon cache performance. By using strides, you can have data scattered in storage. Best performance is obtained with data closely spaced in storage and with elements of the sequence in contiguous locations. The optimum values for *inc1h, inc1x*, and *inc1y* are 1.

In writing the calling program, you may find it convenient to declare `X` or `Y` as a two-dimensional array. For example, you can declare `X` in a DIMENSION statement as `X(INC2X,M)`.

Fourier Transform Considerations

This section describes some ways to optimize performance in the Fourier transform subroutines.

Setting Up Your Data

Many of the Fourier transform, convolution, and correlation subroutines provide the facility for processing many sequences in one call. For short sequences, for example 1024 elements or less, this facility should be used as much as possible. This provides improved performance compared to processing only one sequence at a time.

If possible, you should use the same array for input and output. In addition, the requirements for the strides of the input and output arrays are explained in the Notes for each subroutine.

For improved performance, small values of *inc1x* and *inc1y* should be used, where applicable, preferably $inc1x = 1$ and $inc1y = 1$. A stride of 1 means the sequence elements are stored contiguously. Also, if possible, the sequences should be stored close to each other. For all the Fourier transform subroutines except _RCFT and _CRFT, you should use the STRIDE subroutine to determine the optimal stride(s) for your input or output data. Complete instructions on how to use STRIDE for each of these subroutines is included in "STRIDE—Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines" on page 977.

To obtain the best performance in the three-dimensional Fourier transform subroutines, you should use strides, *inc2* and *inc3*, provided by the STRIDE subroutine and declare your three-dimensional data structure as a one-dimensional array. The three-dimensional Fourier transform subroutines assume that *inc1* for the array is 1. Therefore, each element x_{ijk} for $i = 0, 1, ..., n1-1$, $j = 0, 1, ..., n2-1$, and $k = 0, 1, ..., n3-1$ of the three-dimensional data structure of dimensions *n1* by *n2* by *n3* is stored in a one-dimensional array `X(0:L)` at location `X(`l`)`, where $l = i+inc2(j)+inc3(k)$. The minimum required value of `L` is calculated by inserting the maximum values for i, j, and k in the above equation, giving $L = (n1-1)+inc2(n2-1)+inc3(n3-1)$. The minimum total size of array `X` is `L`+1. To ensure that this mapping is unique so no two elements x_{ijk} occupy the same array element, `X(`l`)`, the subroutines have the following restriction: $inc2 \geq n1$ and $inc3 \geq (inc2)(n2)$. This arrangement of array data in storage leaves some

blank space between successive planes of the array X. By determining the best size for this space, specifying an optimum *inc3* stride, the third dimension of the array does not create conflicts in the 3090 storage hierarchy.

If the *inc3* stride value returned by the STRIDE subroutine turns out to be a multiple of *inc2*, the array X can be declared as a three-dimensional array as X(*inc2*,*inc3*/*inc2*,*n3*); otherwise, it can be declared as either a one-dimensional array, X(0:L), as described above, or a two-dimensional array X(0:*inc3*−1,0:*n3*−1), where x_{ijk} is stored in X(*l*,*k*) where $l = i+(inc2)(j)$.

Using the Scale Argument

If you must multiply either the input or the output sequences by a common factor, you can avoid the multiplication by letting the *scale* argument contain the factor. The subroutines multiply the sine and cosine values by the scale factor during the initialization. Thus, scaling takes no time after the initialization of the Fourier transform calculations.

How the Fourier Transform Subroutines Achieve High Performance

There are two levels of optimization for the fast Fourier transforms (FFTs) in the ESSL library. For sequences with a large power of 2 length, we provide efficient radix-2 and radix-8 transform implementations where cache use is optimized. The cache optimization includes ordering of operations to maximize stride-1 data access and prefetching cache lines.

Similar optimization techniques are used for sequence lengths which are not a power of 2 and mixed-radix FFT's are performed. Many short sequence FFT's have sequence size specific optimizations. Some of these optimizations were originally developed for a vector machine and have been adapted for cache based RISC machines (see references [1], [5], and [7])

The other optimization in the FFT routine is to treat multiple sequences as efficiently as possible. Techniques here include blocking sequences to fit into available CPU cache and transposing sequences to ensure stride-1 access. Whenever possible, the highest performance can be obtained when multiple sequences are transformed in a single call.

Convolution and Correlation Considerations

This section describes some ways to optimize performance in the convolution and correlation subroutines.

Performance Tradeoffs between Subroutines

The subroutines SCON, SCOR, SACOR, SCOND, SCORD, SDCON, SDCOR, DDCON, and DDCOR compute convolutions, correlations, and autocorrelations using essentially the same methods. They make a decision, based on estimated timings, to use one of two methods:

- A direct method that is most efficient when one or both of the input sequences are short
- A direct method that is most efficient when the output sequence is short

Using this approach has the following advantages:

- In most cases, improved performance can be achieved for direct methods because:
  - No initialization is required.

  - No working storage or padding of sequences is necessary.
- In some cases, greater accuracy may be available.
- Negative strides can be used.

In general, using SCONF, SCORF, and SACORF provides the best performance, because the mixed-radix Fourier transform subroutines are used. However, if you can determine from your arguments that a direct method is preferred, you should use SCOND and SCORD instead. These give you better performance for the direct methods, and also give you additional capabilities.

In cases where there is doubt as to the best choice of a subroutine, perform timing experiments.

Special Uses of SCORD

The subroutine SCORD can perform the functions of SCON and SACOR; that is, it can compute convolutions and autocorrelations. To compute a convolution, you must specify a negative stride for ***h*** (see Example 4 in SCORD). To compute the autocorrelation, you must specify the two input sequences to be the same (see Example 5 in SCORD).

Special Uses of _DCON and _DCOR

The _DCON and _DCOR subroutines compute convolutions and correlations, respectively, by the direct method with decimated output. Setting the decimation interval $id = 1$ in SDCON and SDCOR provides the same function as SCOND and SCORD, respectively. Doing the same in DDCON and DDCOR provides long-precision versions of SCOND and SCORD, respectively, which are not otherwise available.

Accuracy When Direct Methods Are Used

The direct methods used by the convolution and correlation subroutines use vector operations to accumulate sums of products. The products are computed and accumulated in long precision. As a result, higher accuracy can be obtained in the final results for some types of data. For example, if input data consists only of integers, and if no intermediate and final numbers become too large (larger than $2^{24}-1$ for short-precision computations and larger than $2^{56}-1$ for long-precision computations), the results are exact.

Accuracy When Fourier Methods Are Used

The Fourier methods used by the convolution and correlation subroutines compute Fourier transforms of input data that is multiplied element-by-element in short-precision arithmetic. The inverse Fourier transform is then computed. There are internally generated rounding errors in the Fourier transforms. It has been shown in references [90] and [79] that, in the case of white noise data, the relative root mean square (RMS) error of the Fourier transform is proportional to $log_2 n$ with a very small proportionality factor. In general, with random, evenly distributed data, this is better than the RMS error of the direct method. However, one must keep in mind the fact that, while the Fourier method may yield a smaller root mean square error, there can be points with large relative errors. Thus, it can happen that some points, usually at the ends of the output sequence, can be obtained with greater relative accuracy with direct methods.

Convolutions and Correlations by Fourier Methods

The convolution and correlation subroutines that use the Fourier methods determine a sequence length *n*, whose Fourier transform is computed using ESSL subroutines. In the simple case where *iy0* = 0 for convolution or *iy0* = −*nh*+1 for correlation, *n* is chosen as a value greater than or equal to the following, which is also acceptable to the Fourier tranform subroutines:

nt = min(*nh*+*nx*−1, *ny*) for convolution and correlation
nt = min(*nx*+*nx*−1, *ny*) for autocorrelation

which is also acceptable to the Fourier subroutines.

Related Computation Considerations

This section describes some key points about using the related-computation subroutines.

Accuracy Considerations

- Many of the subroutines performing short-precision computations provide increased accuracy by accumulating results in long precision. This is noted in the functional description for each subroutine.
- There are ESSL-specific rules that apply to the results of computations on the workstation processors using the ANSI/IEEE standards. For details, see "What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?" on page 44.

Fourier Transform Subroutines

This section contains the Fourier transform subroutine descriptions.

SCFT and DCFT—Complex Fourier Transform

These subroutines compute a set of *m* complex discrete *n*-point Fourier transforms of complex data.

Table 126. Data Types

X, Y	*scale*	**Subroutine**
Short-precision complex	Short-precision real	SCFT
Long-precision complex	Long-precision real	DCFT

Note: Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.

Syntax

Fortran	CALL SCFT | DCFT (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	scft | dcft (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);
PL/I	CALL SCFT | DCFT (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init
is a flag, where:

If *init* ≠ 0, trigonometric functions and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*. The contents of *x* and *y* are not used or changed.

If *init* = 0, the discrete Fourier transforms of the given sequences are computed. The only arguments that may change after initialization are *x*, *y*, and *aux2*. All scalar arguments must be the same as when the subroutine was called for initialization with *init* ≠ 0.

Specified as: a fullword integer. It can have any value.

x
is the array X, consisting of *m* sequences of length *n*. Specified as: an array of (at least) length 1+(*n*−1)*inc1x*+(*m*−1)*inc2x*, containing numbers of the data type indicated in Table 126.

inc1x
is the stride between the elements within each sequence in array X. Specified as: a fullword integer; *inc1x* > 0.

inc2x
is the stride between the first elements of the sequences in array X. (If *m* = 1, this argument is ignored.) Specified as: a fullword integer; *inc2x* > 0.

y
See On Return.

inc1y
is the stride between the elements within each sequence in array Y. Specified as: a fullword integer; *inc1y* > 0.

inc2y
is the stride between the first elements of each sequence in array Y. (If $m = 1$, this argument is ignored.) Specified as: a fullword integer; $inc2y > 0$.

n
is the length of each sequence to be transformed. Specified as: a fullword integer; $n \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 741. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument, as well as in the optionally-recoverable error 2030. For details, see "Providing a Correct Transform Length to ESSL" on page 38.

m
is the number of sequences to be transformed. Specified as: a fullword integer; $m > 0$.

isign
controls the direction of the transform, determining the sign *Isign* of the exponent of W_n, where:

If $isign$ = positive value, $Isign = +$ (transforming time to frequency).

If $isign$ = negative value, $Isign = -$ (transforming frequency to time).

Specified as: a fullword integer; $isign > 0$ or $isign < 0$.

scale
is the scaling constant *scale*. See "Function" on page 752 for its usage. Specified as: a number of the data type indicated in Table 126 on page 749, where $scale > 0.0$ or $scale < 0.0$

aux1
is the working storage for this subroutine, where:

If $init \neq 0$, the working storage is computed.

If $init = 0$, the working storage is used in the computation of the Fourier transforms.

Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1
is the number of doublewords in the working storage specified in *aux1*. Specified as: a fullword integer; $naux1 > 7$ and $naux1 \geq$ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For values between 7 and the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 31.

aux2
has the following meaning:

If $naux2 = 0$ and error 2015 is unrecoverable, *aux2* is ignored.

Otherwise, it is the working storage used by this subroutine, which is available for use by the calling program between calls to this subroutine.

Specified as: an area of storage, containing *naux2* long-precision real numbers. On output, the contents are overwritten.

naux2
is the number of doublewords in the working storage specified in *aux2*. Specified as: a fullword integer, where:

If $naux2 = 0$ and error 2015 is unrecoverable, SCFT and DCFT dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux2* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 31.

On Return

y

has the following meaning, where:

If *init* ≠ 0, this argument is not used, and its contents remain unchanged.

If *init* = 0, this is array Y, consisting of the results of the *m* discrete Fourier transforms, each of length *n*.

Returned as: an array of (at least) length 1+(*n*−1)*inc1y*+(*m*−1)*inc2y*, containing numbers of the data type indicated in Table 126 on page 749. This array must be aligned on a doubleword boundary.

aux1

is the working storage for this subroutine, where:

If *init* ≠ 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If *init* = 0, its contents are unchanged.

Returned as: the contents are not relevant.

Notes

1. *aux1* should **not** be used by the calling program between calls to this subroutine with *init* ≠ 0 and *init* = 0. However, it can be reused after intervening calls to this subroutine with different arguments.
2. For optimal performance, the preferred value for *inc1x* and *inc1y* is 1. This implies that the sequences are stored with stride 1. The preferred value for *inc2x* and *inc2y* is *n*. This implies that sequences are stored one after another without any gap.

   It is possible to specify sequences in the transposed form—that is, as rows of a two-dimensional array. In this case, *inc2x* (or *inc2y*) = 1 and *inc1x* (or *inc1y*) is equal to the leading dimension of the array. One can specify either input, output, or both in the transposed form by specifying appropriate values for the stride parameters. For selecting optimal values of *inc1x* and *inc1y* for _CFT, you should use "STRIDE—Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines" on page 977. Example 1 in the STRIDE subroutine description explains how it is used for _CFT.

   If you specify the same array for X and Y, then *inc1x* and *inc1y* must be equal, and *inc2x* and *inc2y* must be equal. In this case, output overwrites input. If *m* = 1, the *inc2x* and *inc2y* values are not used by the subroutine. If you specify different arrays for X and Y, they must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.

Processor-Independent Formulas for SCFT for NAUX1 and NAUX2

NAUX1 Formulas:

If *n* ≤ 8192, use *naux1* = 20000.
If *n* > 8192, use *naux1* = 20000+1.14*n*.

NAUX2 Formulas:

If $n \leq 8192$, use *naux2* = 20000.
If $n > 8192$, use *naux2* = 20000+1.14*n*.
For the transposed case, where *inc2x* = 1 or *inc2y* = 1, and where $n \geq 252$, add the following to the above storage requirements:
(*n*+256)(min(64, *m*)).

Processor-Independent Formulas for DCFT for NAUX1 and NAUX2

NAUX1 Formulas:

If $n \leq 2048$, use *naux1* = 20000.
If $n > 2048$, use *naux1* = 20000+2.28*n*.

NAUX2 Formulas:

If $n \leq 2048$, use *naux2* = 20000.
If $n > 2048$, use *naux2* = 20000+2.28*n*.
For the transposed case, where *inc2x* = 1 or *inc2y* = 1, and where $n \geq 252$, add the following to the above storage requirements:
(2*n*+256)(min(64, *m*)).

Function: The set of *m* complex discrete n-point Fourier transforms of complex data in array `X`, with results going into array `Y`, is expressed as follows:

$$y_{ki} = scale \sum_{j=0}^{n-1} x_{ji} W_n^{(Isign)jk}$$

for:

$k = 0, 1, ..., n-1$
$i = 1, 2, ..., m$

where:

$$W_n = e^{-2\pi\left(\sqrt{-1}\right)/n}$$

and where:

x_{ji} are elements of the sequences in array `X`.
y_{ki} are elements of the sequences in array `Y`.
Isign is + or − (determined by argument *isign*).
scale is a scalar value.

For *scale* = 1.0 and *isign* being positive, you obtain the discrete Fourier transform, a function of frequency. The inverse Fourier transform is obtained with *scale* = 1.0/*n* and *isign* being negative. See references [1], [3], [4], [19], and [20].

Two invocations of this subroutine are necessary:

1. With *init* ≠ 0, the subroutine tests and initializes arguments of the program, setting up the *aux1* working storage.
2. With *init* = 0, the subroutine checks that the initialization arguments in the *aux1* working storage correspond to the present arguments, and if so, performs the calculation of the Fourier transforms.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux2* = 0, and unable to allocate work area.

Computational Errors: None

Input-Argument Errors

1. $n > 37748736$
2. *inc1x*, *inc2x*, *inc1y*, or *inc2y* ≤ 0
3. $m \leq 0$
4. *isign* = 0
5. *scale* = 0.0
6. The subroutine has not been initialized with the present arguments.
7. The length of the transform in *n* is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
8. *naux1* ≤ 7
9. *naux1* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
10. Error 2015 is recoverable or *naux2*≠0, and *naux2* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example 1: This example shows an input array `X` with a set of four short-precision complex sequences:

$$e^{2\pi\left(\sqrt{-1}\right)jk/n}$$

for $j = 0, 1, ..., n-1$ with $n = 8$, and the single frequencies $k = 0$, 1, 2, and 3. The arrays are declared as follows:

```
COMPLEX*8  X(0:1023),Y(0:1023)
REAL*8     AUX1(1693),AUX2(4096)
```

First, initialize `AUX1` using the calling sequence shown below with `INIT` ≠ 0. Then use the same calling sequence with `INIT = 0` to do the calculation.

Call Statement and Input

```
          INIT  X INC1X INC2X Y INC1Y INC2Y N   M ISIGN SCALE  AUX1   NAUX1  AUX2   NAUX2
           |    |   |     |   |   |     |   |   |   |     |     |       |     |       |
CALL SCFT(INIT, X , 1  ,  8 , Y , 1  ,  8 , 8 , 4 , 1 , SCALE, AUX1 , 1693 , AUX2 , 4096)
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
SCALE    =  1.0
```

X contains the following four sequences:

```
(1.0000, 0.0000)   (1.0000,  0.0000)   (1.0000,  0.0000)   (1.0000,  0.0000)
(1.0000, 0.0000)   (0.7071,  0.7071)   (0.0000,  1.0000)  (-0.7071,  0.7071)
(1.0000, 0.0000)   (0.0000,  1.0000)  (-1.0000,  0.0000)   (0.0000, -1.0000)
(1.0000, 0.0000)  (-0.7071,  0.7071)   (0.0000, -1.0000)   (0.7071,  0.7071)
(1.0000, 0.0000)  (-1.0000,  0.0000)   (1.0000,  0.0000)  (-1.0000,  0.0000)
(1.0000, 0.0000)  (-0.7071, -0.7071)   (0.0000,  1.0000)   (0.7071, -0.7071)
(1.0000, 0.0000)   (0.0000, -1.0000)  (-1.0000,  0.0000)   (0.0000,  1.0000)
(1.0000, 0.0000)   (0.7071, -0.7071)   (0.0000, -1.0000)  (-0.7071, -0.7071)
```

Output: Y contains the following four sequences:

```
(8.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (8.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (8.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (8.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
```

Example 2: This example shows an input array X with a set of four input spike sequences equal to the output of Example 1. This shows how you can compute the inverse of the transform in Example 1 by using a negative *isign*, giving as output the four sequences listed in the input for Example 1. First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Call Statement and Input

```
          INIT  X INC1X INC2X Y INC1Y INC2Y N   M  ISIGN SCALE   AUX1   NAUX1  AUX2   NAUX2
           |    |   |     |   |   |     |   |   |    |     |      |       |     |       |
CALL SCFT(INIT, X , 1  ,  8 , Y , 1  ,  8 , 8 , 4 , -1 , SCALE , AUX1 , 1693 , AUX2 , 4096)
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
SCALE    =  0.125
X        =(same as output Y in Example 1)
```

Output

```
Y        =(same as input X in Example 1)
```

Example 3: This example shows an input array X with a set of four short-precision complex sequences

$$e^{2\pi\left(\sqrt{-1}\right)jk/n}$$

for $j = 0, 1, ..., n-1$ with $n = 12$, and the single frequencies $k = 0, 1, 2$, and 3. Also, $inc1x = inc1y = m$ and $inc2x = inc2y = 1$ to show how the input and output arrays can be stored in the transposed form. The arrays are declared as follows:

```
COMPLEX*8  X (4,0:11),Y(4,0:11)
REAL*8     AUX1(10000),AUX2(10000)
```

First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Call Statement and Input

```
          INIT  X INC1X INC2X Y INC1Y INC2Y  N   M ISIGN SCALE  AUX1   NAUX1   AUX2   NAUX2
           |    |   |     |   |   |     |    |   |   |     |     |       |      |       |
CALL SCFT(INIT, X , 4  ,  1 , Y , 4  ,  1 , 12 , 4 , 1 , SCALE, AUX1 , 10000 , AUX2 , 10000)
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
SCALE    =  1.0
```

X contains the following four sequences:

```
(1.0000, 0.0000)   (1.0000, 0.0000)   (1.0000, 0.0000)   (1.0000, 0.0000)
(1.0000, 0.0000)   (0.8660, 0.5000)   (0.5000, 0.8660)   (0.0000, 1.0000)
(1.0000, 0.0000)   (0.5000, 0.8660)  (-0.5000, 0.8660)  (-1.0000, 0.0000)
(1.0000, 0.0000)   (0.0000, 1.0000)  (-1.0000, 0.0000)  (0.0000, -1.0000)
(1.0000, 0.0000)  (-0.5000, 0.8660) (-0.5000, -0.8660)   (1.0000, 0.0000)
(1.0000, 0.0000)  (-0.8660, 0.5000)  (0.5000, -0.8660)   (0.0000, 1.0000)
(1.0000, 0.0000)  (-1.0000, 0.0000)   (1.0000, 0.0000)  (-1.0000, 0.0000)
(1.0000, 0.0000) (-0.8660, -0.5000)   (0.5000, 0.8660)  (0.0000, -1.0000)
(1.0000, 0.0000) (-0.5000, -0.8660)  (-0.5000, 0.8660)   (1.0000, 0.0000)
(1.0000, 0.0000)  (0.0000, -1.0000)  (-1.0000, 0.0000)   (0.0000, 1.0000)
(1.0000, 0.0000)  (0.5000, -0.8660) (-0.5000, -0.8660) (-1.0000,  0.0000)
(1.0000, 0.0000)  (0.8660, -0.5000)  (0.5000, -0.8660)  (0.0000, -1.0000)
```

Output: Y contains the following four sequences:

```
(12.0000, 0.0000)   (0.0000, 0.0000)   (0.0000, 0.0000)   (0.0000, 0.0000)
 (0.0000, 0.0000)  (12.0000, 0.0000)   (0.0000, 0.0000)   (0.0000, 0.0000)
 (0.0000, 0.0000)   (0.0000, 0.0000)  (12.0000, 0.0000)   (0.0000, 0.0000)
 (0.0000, 0.0000)   (0.0000, 0.0000)   (0.0000, 0.0000)  (12.0000, 0.0000)
 (0.0000, 0.0000)   (0.0000, 0.0000)   (0.0000, 0.0000)   (0.0000, 0.0000)
 (0.0000, 0.0000)   (0.0000, 0.0000)   (0.0000, 0.0000)   (0.0000, 0.0000)
 (0.0000, 0.0000)   (0.0000, 0.0000)   (0.0000, 0.0000)   (0.0000, 0.0000)
 (0.0000, 0.0000)   (0.0000, 0.0000)   (0.0000, 0.0000)   (0.0000, 0.0000)
 (0.0000, 0.0000)   (0.0000, 0.0000)   (0.0000, 0.0000)   (0.0000, 0.0000)
 (0.0000, 0.0000)   (0.0000, 0.0000)   (0.0000, 0.0000)   (0.0000, 0.0000)
 (0.0000, 0.0000)   (0.0000, 0.0000)   (0.0000, 0.0000)   (0.0000, 0.0000)
```

Example 4: This example shows an input array X with a set of four input spike sequences exactly equal to the output of Example 3. This shows how you can compute the inverse of the transform in Example 3 by using a negative *isign*, giving as output the four sequences listed in the input for Example 3. First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Call Statement and Input

```
          INIT  X INC1X INC2X Y INC1Y INC2Y  N   M  ISIGN SCALE   AUX1  NAUX1  AUX2  NAUX2
           |    |   |     |   |   |     |    |   |    |     |      |      |     |      |
CALL SCFT(INIT, X , 4  ,  1 , Y , 4  ,  1 , 12 , 4 , -1 , SCALE , AUX1, 10000, AUX2, 10000)
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
SCALE    =  1.0/12.0
X        =(same as output Y in Example 3)
```

Output

```
Y        =(same as input X in Example 3)
```

Example 5: This example shows how to compute a transform of a single long-precision complex sequence. It uses *isign* = 1 and *scale* = 1.0. The arrays are declared as follows:

```
COMPLEX*16  X(0:7),Y(0:7)
REAL*8      AUX1(26),AUX2(12)
```

The input in X is an impulse at zero, and the output in Y is constant for all frequencies. First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Call Statement and Input

```
          INIT  X INC1X INC2X Y INC1Y INC2Y N   M ISIGN SCALE   AUX1  NAUX1  AUX2  NAUX2
           |    |   |     |   |   |     |   |   |   |     |      |      |     |      |
CALL DCFT(INIT, X , 1  ,  0 , Y , 1  ,  0 , 8 , 1 , 1 , SCALE , AUX1 , 26  , AUX2 , 12)
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
SCALE    =  1.0
```

X contains the following sequence:

```
(1.0000, 0.0000)
(0.0000, 0.0000)
(0.0000, 0.0000)
(0.0000, 0.0000)
(0.0000, 0.0000)
(0.0000, 0.0000)
(0.0000, 0.0000)
(0.0000, 0.0000)
```

Output:

```
(1.0000, 0.0000)
(1.0000, 0.0000)
(1.0000, 0.0000)
(1.0000, 0.0000)
(1.0000, 0.0000)
(1.0000, 0.0000)
(1.0000, 0.0000)
(1.0000, 0.0000)
```

SRCFT and DRCFT—Real-to-Complex Fourier Transform

These subroutines compute a set of *m* complex discrete *n*-point Fourier transforms of real data.

Table 127. Data Types

X, *scale*	Y	**Subroutine**
Short-precision real	Short-precision complex	SRCFT
Long-precision real	Long-precision complex	DRCFT

Note: Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.

Syntax

Fortran	CALL SRCFT (*init*, *x*, *inc2x*, *y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*, *aux3*, *naux3*) CALL DRCFT (*init*, *x*, *inc2x*, *y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	srcft (*init*, *x*, *inc2x*, *y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*, *aux3*, *naux3*); drcft (*init*, *x*, *inc2x*, *y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);
PL/I	CALL SRCFT (*init*, *x*, *inc2x*, *y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*, *aux3*, *naux3*); CALL DRCFT (*init*, *x*, *inc2x*, *y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init
is a flag, where:

If *init* ≠ 0, trigonometric functions and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*. The contents of *x* and *y* are not used or changed.

If *init* = 0, the discrete Fourier transforms of the given sequences are computed. The only arguments that may change after initialization are *x*, *y*, and *aux2*. All scalar arguments must be the same as when the subroutine was called for initialization with *init* ≠ 0.

Specified as: a fullword integer. It can have any value.

x
is the array X, consisting of *m* sequences of length *n*, which are to be transformed. The sequences are assumed to be stored with stride 1. Specified as: an array of (at least) length *n*+(*m*−1)*inc2x*, containing numbers of the data type indicated in Table 127. See "Notes" on page 760 for more details. (It can be declared as X(*inc2x*,*m*).)

inc2x
is the stride between the first elements of the sequences in array X. (If *m* = 1, this argument is ignored.) Specified as: a fullword integer; *inc2x* ≥ *n*.

y
See On Return.

inc2y
is the stride between the first elements of the sequences in array Y. (If $m = 1$, this argument is ignored.) Specified as: a fullword integer; $inc2y \geq (n/2)+1$.

n
is the length of each sequence to be transformed. Specified as: a fullword integer; $n \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 741. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 38.

m
is the number of sequences to be transformed. Specified as: a fullword integer; $m > 0$.

isign
controls the direction of the transform, determining the sign *Isign* of the exponent of W_n, where:

If *isign* = positive value, *Isign* = + (transforming time to frequency).

If *isign* = negative value, *Isign* = − (transforming frequency to time).

Specified as: a fullword integer; $isign > 0$ or $isign < 0$.

scale
is the scaling constant *scale*. See "Function" on page 761 for its usage. Specified as: a number of the data type indicated in Table 127 on page 758, where $scale > 0.0$ or $scale < 0.0$.

aux1
is the working storage for this subroutine, where:

If $init \neq 0$, the working storage is computed.

If $init = 0$, the working storage is used in the computation of the Fourier transforms.

Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1
is the number of doublewords in the working storage specified in *aux1*. Specified as: a fullword integer; $naux1 > 14$ and $naux1 \geq$ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For values between 14 and the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 31.

aux2
has the following meaning:

If $naux2 = 0$ and error 2015 is unrecoverable, *aux2* is ignored.

Otherwise, it is the working storage used by this subroutine, which is available for use by the calling program between calls to this subroutine.

Specified as: an area of storage, containing *naux2* long-precision real numbers. On output, the contents are overwritten.

naux2
is the number of doublewords in the working storage specified in *aux2*. Specified as: a fullword integer, where:

If $naux2 = 0$ and error 2015 is unrecoverable, SRCFT and DRCFT dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux2* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 31.

aux3
this argument is provided for migration purposes only and is ignored.

Specified as: an area of storage, containing *naux3* long-precision real numbers.

naux3
this argument is provided for migration purposes only and is ignored.

Specified as: a fullword integer.

On Return

y
has the following meaning, where:

If *init* ≠ 0, this argument is not used, and its contents remain unchanged.

If *init* = 0, this is array Y, consisting of the results of the *m* complex discrete Fourier transforms, each of length *n*. The sequences are stored with the stride 1. Due to complex conjugate symmetry, only the first (n/2) + 1 elements of each sequence are given in the output—that is, y_{ki}, $k = 0, 1, ..., n/2$, $i = 1, 2, ..., m$.

Returned as: an array of (at least) length $n/2+1+(m-1)inc2y$, containing numbers of the data type indicated in Table 127 on page 758. This array must be aligned on a doubleword boundary. (It can be declared as Y(*inc2y*,*m*).)

aux1
is the working storage for this subroutine, where:

If *init* ≠ 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If *init* = 0, its contents are unchanged.

Returned as: the contents are not relevant.

Notes

1. *aux1* should **not** be used by the calling program between calls to this subroutine with *init* ≠ 0 and *init* = 0. However, it can be reused after intervening calls to this subroutine with different arguments.
2. In these subroutines, the elements in each sequence in ***x*** and ***y*** are assumed to be stored in contiguous storage locations, using a stride of 1; therefore, *inc1x* and *inc1y* values are not a part of the argument list. For optimal performance, the *inc2x* and *inc2y* values should be close to their respective minimum values, which are given below:

   $\min(inc2x) = n$
   $\min(inc2y) = n/2+1$

   If you specify the same array for X and Y, then *inc2x* must equal 2(*inc2y*). In this case, output overwrites input. If $m = 1$, the *inc2x* and *inc2y* values are not used by the subroutine. If you specify different arrays for X and Y, they must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.

3. Be sure to align array X on a doubleword boundary, and specify an even number for *inc2x*, if possible.

Processor-Independent Formulas for SRCFT for NAUX1 and NAUX2

NAUX1 Formulas

If $n \leq 16384$, use $naux1 = 25000$.
If $n > 16384$, use $naux1 = 20000+0.82n$.

NAUX2 Formulas

If $n \leq 16384$, use $naux2 = 20000$.
If $n > 16384$, use $naux2 = 20000+0.57n$.

Processor-Independent Formulas for DRCFT for NAUX1 and NAUX2

NAUX1 Formulas

If $n \leq 4096$, use $naux1 = 22000$.
If $n > 4096$, use $naux1 = 20000+1.64n$.

NAUX2 Formulas

If $n \leq 4096$, use $naux2 = 20000$.
If $n > 4096$, use $naux2 = 20000+1.14n$.

Function: The set of *m* complex conjugate even discrete n-point Fourier transforms of real data in array X, with results going into array Y, is expressed as follows:

$$y_{ki} = scale \sum_{j=0}^{n-1} x_{ji} W_n^{(Isign)jk}$$

for:

$k = 0, 1, ..., n-1$
$i = 1, 2, ..., m$

where:

$$W_n = e^{-2\pi\left(\sqrt{-1}\right)/n}$$

and where:

x_{ji} are elements of the sequences in array X.
y_{ki} are elements of the sequences in array Y.
Isign is + or − (determined by argument *isign*).
scale is a scalar value.

The output in array Y is complex. For $scale = 1.0$ and *isign* being positive, you obtain the discrete Fourier transform, a function of frequency. The inverse Fourier transform is

obtained with *scale* = 1.0/*n* and *isign* being negative. See references [1], [4], [19], and [20].

Two invocations of this subroutine are necessary:

1. With *init* ≠ 0, the subroutine tests and initializes arguments of the program, setting up the *aux1* working storage.
2. With *init* = 0, the subroutine checks that the initialization arguments in the *aux1* working storage correspond to the present arguments, and if so, performs the calculation of the Fourier transforms.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux2* = 0, and unable to allocate work area.

Computational Errors: None

Input-Argument Errors

1. $n > 37748736$
2. $m \leq 0$
3. $inc2x < n$
4. $inc2y < n/2+1$
5. $isign = 0$
6. $scale = 0.0$
7. The subroutine has not been initialized with the present arguments.
8. The length of the transform in *n* is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
9. $naux1 \leq 14$
10. *naux1* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
11. Error 2015 is recoverable or $naux2 \neq 0$, and *naux2* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example 1: This example shows an input array X with a set of *m* cosine sequences $\cos(2\pi jk/n)$, $j = 0, 1, ..., 15$ with the single frequencies $k = 0, 1, 2, 3$. The Fourier transform of the cosine sequence with frequency $k = 0$ or n/2 has 1.0 in the 0 or n/2 position, respectively, and zeros elsewhere. For all other *k*, the Fourier transform has 0.5 in the *k* position and zeros elsewhere. The arrays are declared as follows:

```
REAL*4     X(0:65535)
COMPLEX*8  Y(0:32768)
REAL*8     AUX1(41928), AUX2(35344), AUX3(1)
```

First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Call Statement and Input

```
           INIT  X  INC2X Y INC2Y  N   M ISIGN SCALE  AUX1   NAUX1   AUX2   NAUX2   AUX3  NAUX3
            |    |    |   |   |    |   |   |     |     |       |      |       |      |      |
CALL SRCFT(INIT, X , 16 , Y , 9 , 16 , 4 , 1 , SCALE, AUX1 , 41928 , AUX2 , 35344 , AUX3 ,  0 )
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
SCALE    =  1.0/16
```

X contains the following four sequences:

```
1.0000   1.0000   1.0000   1.0000
1.0000   0.9239   0.7071   0.3827
1.0000   0.7071   0.0000  -0.7071
1.0000   0.3827  -0.7071  -0.9239
1.0000   0.0000  -1.0000   0.0000
1.0000  -0.3827  -0.7071   0.9239
1.0000  -0.7071   0.0000   0.7071
1.0000  -0.9239   0.7071  -0.3827
1.0000  -1.0000   1.0000  -1.0000
1.0000  -0.9239   0.7071  -0.3827
1.0000  -0.7071   0.0000   0.7071
1.0000  -0.3827  -0.7071   0.9239
1.0000   0.0000  -1.0000   0.0000
1.0000   0.3827  -0.7071  -0.9239
1.0000   0.7071   0.0000  -0.7071
1.0000   0.9239   0.7071   0.3827
```

Output: Y contains the following four sequences:

```
(1.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.5000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.5000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.5000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
```

Example 2: This example shows another transform computation with different data using the same initialized array AUX1 as in Example 1. The input is also a set of four cosine sequences $\cos(2\pi jk/n)$, $j = 0, 1, ..., 15$ with the single frequencies $k = 8, 9, 10, 11$, thus including the middle frequency $k = 8$. The middle frequency has the value 1.0. For other frequencies, the transform has zeros, except for frequencies k and $n-k$. Only the values for $j = n-k$ are given in the output.

Call Statement and Input

```
           INIT X  INC2X Y INC2Y  N   M ISIGN SCALE  AUX1   NAUX1   AUX2   NAUX2   AUX3  NAUX3
            |   |    |   |   |    |   |   |     |     |       |      |       |      |
CALL SRCFT( 0 , X , 16 , Y , 9 , 16 , 4 , 1 , SCALE, AUX1 , 41928 , AUX2 , 35344 , AUX3 ,  0  )
```

```
SCALE    =  1.0/16
```

X contains the following four sequences:

```
 1.0000   1.0000   1.0000   1.0000
-1.0000  -0.9239  -0.7071  -0.3827
 1.0000   0.7071   0.0000  -0.7071
-1.0000  -0.3827   0.7071   0.9239
 1.0000   0.0000  -1.0000   0.0000
-1.0000   0.3827   0.7071  -0.9239
 1.0000  -0.7071   0.0000   0.7071
-1.0000   0.9239  -0.7071   0.3827
 1.0000  -1.0000   1.0000  -1.0000
-1.0000   0.9239  -0.7071   0.3827
 1.0000  -0.7071   0.0000   0.7071
-1.0000   0.3827   0.7071  -0.9239
 1.0000   0.0000  -1.0000   0.0000
-1.0000  -0.3827   0.7071   0.9239
 1.0000   0.7071   0.0000  -0.7071
-1.0000  -0.9239  -0.7071  -0.3827
```

Output: Y contains the following four sequences:

```
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.5000, 0.0000)
(0.0000, 0.0000)  (0.0000, 0.0000)  (0.5000, 0.0000)  (0.0000, 0.0000)
(0.0000, 0.0000)  (0.5000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
(1.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)  (0.0000, 0.0000)
```

Example 3: This example uses the mixed-radix capability. The arrays are declared as follows:

```
REAL*8     X(0:11)
COMPLEX*16 Y(0:6)
REAL*8     AUX1(50),AUX2(50)
```

Arrays X and Y are made equivalent by the following statement, making them occupy the same storage:

```
EQUIVALENCE (X,Y)
```

First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Call Statement and Input

```
          INIT  X INC2X Y INC2Y  N   M ISIGN SCALE   AUX1  NAUX1 AUX2  NAUX2
           |    |   |   |   |    |   |   |     |      |      |    |      |
CALL DRCFT(INIT, X , 0 , Y , 0 , 12 , 1 , 1 , SCALE , AUX1 , 50 , AUX2 , 50)
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
SCALE    =  1.0
X        =  (1.0000 , 1.0000 , 1.0000 , 1.0000 , 1.0000 , 1.0000 ,
             1.0000 , 1.0000 , 1.0000 , 1.0000 , 1.0000 , 1.0000)
```

Output: `Y` contains the following sequence:

```
(12.0000 , 0.0000)
 (0.0000 , 0.0000)
 (0.0000 , 0.0000)
 (0.0000 , 0.0000)
 (0.0000 , 0.0000)
 (0.0000 , 0.0000)
 (0.0000 , 0.0000)
```

SCRFT and DCRFT—Complex-to-Real Fourier Transform

These subroutines compute a set of *m* real discrete *n*-point Fourier transforms of complex conjugate even data.

Table 128. Data Types

X	Y, *scale*	**Subroutine**
Short-precision complex	Short-precision real	SCRFT
Long-precision complex	Long-precision real	DCRFT

Note: Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.

Syntax

Fortran	CALL SCRFT (*init*, *x*, *inc2x*, *y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*, *aux3*, *naux3*) CALL DCRFT (*init*, *x*, *inc2x*, *y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	scrft (*init*, *x*, *inc2x*, *y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*, *aux3*, *naux3*); dcrft (*init*, *x*, *inc2x*, *y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);
PL/I	CALL SCRFT (*init*, *x*, *inc2x*, *y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*, *aux3*, *naux3*); CALL DCRFT (*init*, *x*, *inc2x*, *y*, *inc2y*, *n*, *m*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init
is a flag, where:

If *init* ≠ 0, trigonometric functions and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*. The contents of *x* and *y* are not used or changed.

If *init* = 0, the discrete Fourier transforms of the given sequences are computed. The only arguments that may change after initialization are *x*, *y*, and *aux2*. All scalar arguments must be the same as when the subroutine was called for initialization with *init* ≠ 0.

Specified as: a fullword integer. It can have any value.

x
is the array X, consisting of *m* sequences. Due to complex conjugate symmetry, the input consists of only the first (*n*/2)+1 elements of each sequence; that is, x_{ji}, $j = 0, 1, ..., n/2$, $i = 1, 2, ..., m$. The sequences are assumed to be stored with stride 1.

Specified as: an array of (at least) length *n*/2+1+(*m*−1)*inc2x*, containing numbers of the data type indicated in Table 128. This array must be aligned on a doubleword boundary. (It can be declared as X(*inc2x*,*m*).)

inc2x
is the stride between the first elements of the sequences in array X. (If *m* = 1, this argument is ignored.) Specified as: a fullword integer; *inc2x* ≥ (*n*/2)+1.

y
See On Return.

inc2y
is the stride between the first elements of the sequences in array Y. (If $m = 1$, this argument is ignored.) Specified as: a fullword integer; $inc2y \geq n$.

n
is the length of each sequence to be transformed. Specified as: a fullword integer; $n \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 741. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 38.

m
is the number of sequences to be transformed. Specified as: a fullword integer; $m > 0$.

isign
controls the direction of the transform, determining the sign *Isign* of the exponent of W_n, where:

If *isign* = positive value, *Isign* = + (transforming time to frequency).

If *isign* = negative value, *Isign* = − (transforming frequency to time).

Specified as: a fullword integer; $isign > 0$ or $isign < 0$.

scale
is the scaling constant *scale*. See "Function" on page 769 for its usage. Specified as: a number of the data type indicated in Table 128 on page 766, where $scale > 0.0$ or $scale < 0.0$.

aux1
is the working storage for this subroutine, where:

If $init \neq 0$, the working storage is computed.

If $init = 0$, the working storage is used in the computation of the Fourier transforms.

Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1
is the number of doublewords in the working storage specified in *aux1*. Specified as: a fullword integer; $naux1 > 13$ and $naux1 \geq$ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For values between 13 and the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 31.

aux2
has the following meaning:

If $naux2 = 0$ and error 2015 is unrecoverable, *aux2* is ignored.

Otherwise, it is the working storage used by this subroutine that is available for use by the calling program between calls to this subroutine.

Specified as: an area of storage, containing *naux2* long-precision real numbers. On output, the contents are overwritten.

naux2
is the number of doublewords in the working storage specified in *aux2*. Specified as: a fullword integer, where:

If $naux2 = 0$ and error 2015 is unrecoverable, SCRFT and DCRFT dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux2* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 31.

aux3
this argument is provided for migration purposes only and is ignored.

Specified as: an area of storage, containing *naux3* long-precision real numbers.

naux3
this argument is provided for migration purposes only and is ignored.

Specified as: a fullword integer.

On Return

y
has the following meaning, where:

If *init* ≠ 0, this argument is not used, and its contents remain unchanged.

If *init* = 0, this is array Y, consisting of the results of the *m* discrete Fourier transforms of the complex conjugate even data, each of length *n*. The sequences are stored with stride 1.

Returned as: an array of (at least) length *n*+(*m*−1)*inc2y*, containing numbers of the data type indicated in Table 128 on page 766. See "Notes" for more details. (It can be declared as Y(*inc2y*,*m*).)

aux1
is the working storage for this subroutine, where:

If *init* ≠ 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If *init* = 0, its contents are unchanged.

Returned as: the contents are not relevant.

Notes

1. *aux1* should **not** be used by the calling program between calls to this subroutine with *init* ≠ 0 and *init* = 0. However, it can be reused after intervening calls to this subroutine with different arguments.
2. The elements in each sequence in ***x*** and ***y*** are assumed to be stored in contiguous storage locations—that is, with a stride of 1. Therefore, *inc1x* and *inc1y* values are not a part of the argument list. For optimal performance, the *inc2x* and *inc2y* values should be close to their respective minimum values, which are given below:

   $\min(inc2y) = n$
   $\min(inc2x) = n/2+1$

   If you specify the same array for X and Y, then *inc2y* must equal 2(*inc2x*). In this case, output overwrites input. If *m* = 1, the *inc2x* and *inc2y* values are not used by the subroutine. If you specify different arrays for X and Y, they must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.
3. Be sure to align array Y on a doubleword boundary, and specify an even number for *inc2y*, if possible.

Processor-Independent Formulas for SCRFT for NAUX1 and NAUX2

NAUX1 Formulas

If $n \leq 16384$, use *naux1* = 25000.
If $n > 16384$, use *naux1* = 20000+0.82*n*.

NAUX2 Formulas

If $n \leq 16384$, use *naux2* = 20000.
If $n > 16384$, use *naux2* = 20000+0.57*n*.

Processor-Independent Formulas for DCRFT for NAUX1 and NAUX2

NAUX1 Formulas

If $n \leq 4096$, use *naux1* = 22000.
If $n > 4096$, use *naux1* = 20000+1.64*n*.

NAUX2 Formulas

If $n \leq 4096$, use *naux2* = 20000.
If $n > 4096$, use *naux2* = 20000+1.14*n*.

Function: The set of *m* real discrete *n*-point Fourier transforms of complex conjugate even data in array X, with results going into array Y, is expressed as follows:

$$y_{ki} = scale \sum_{j=0}^{n-1} x_{ji} W_n^{(Isign)jk}$$

for:

$k = 0, 1, ..., n-1$
$i = 1, 2, ..., m$

where:

$$W_n = e^{-2\pi\left(\sqrt{-1}\right)/n}$$

and where:

x_{ji} are elements of the sequences in array X.
y_{ki} are elements of the sequences in array Y.
Isign is + or − (determined by argument *isign*).
scale is a scalar value.

Because of the symmetry, Y has real data. For *scale* = 1.0 and *isign* being positive, you obtain the discrete Fourier transform, a function of frequency. The inverse Fourier transform is obtained with *scale* = 1.0/*n* and *isign* being negative. See references [1], [4], [19], and [20].

Two invocations of this subroutine are necessary:

1. With *init* ≠ 0, the subroutine tests and initializes arguments of the program, setting up the *aux1* working storage.
2. With *init* = 0, the subroutine checks that the initialization arguments in the *aux1* working storage correspond to the present arguments, and if so, performs the calculation of the Fourier transforms.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux2* = 0, and unable to allocate work area.

Computational Errors: None

Input-Argument Errors

1. *n* > 37748736
2. *m* ≤ 0
3. *inc2x* < *n*/2+1
4. *inc2y* < *n*
5. *scale* = 0.0
6. *isign* = 0
7. The subroutine has not been initialized with the present arguments.
8. The length of the transform in *n* is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
9. *naux1* ≤ 13
10. *naux1* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
11. Error 2015 is recoverable or *naux2*≠0, and *naux2* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example 1: This example uses the mixed-radix capability and shows how to compute a single transform. The arrays are declared as follows:

```
COMPLEX*8  X(0:6)
REAL*8     AUX1(50), AUX2(50), AUX3(1)
REAL*4     Y(0:11)
```

First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Note: X shows the *n*/2+1 = 7 elements used in the computation.

Call Statement and Input

```
           INIT  X INC2X Y INC2Y  N   M ISIGN SCALE  AUX1  NAUX1 AUX2  NAUX2  AUX3  NAUX3
            |    |   |   |   |    |   |   |     |     |      |     |     |     |      |
CALL SCRFT(INIT, X , 0 , Y , 0 , 12 , 1 , 1 , SCALE, AUX1 , 50 , AUX2 , 50 ,  AUX3 ,  0  )
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
SCALE    =  1.0
```

X contains the following sequence:

```
(1.0,  0.0)
(0.0,  0.0)
(0.0,  0.0)
(0.0,  0.0)
(0.0,  0.0)
(0.0,  0.0)
(0.0,  0.0)
```

Output

```
Y        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
```

Example 2: This example shows another transform computation with different data using the same initialized array AUX1 as in Example 1.

Call Statement and Input

```
          INIT  X INC2X Y INC2Y  N   M ISIGN SCALE  AUX1 NAUX1  AUX2 NAUX2  AUX3  NAUX3
           |    |   |   |   |    |   |   |     |     |     |     |     |     |      |
CALL SCRFT( 0 , X , 0 , Y , 0 , 12 , 1 , 1 , SCALE, AUX1 , 50 , AUX2 , 50 , AUX3 ,  0 )
```

```
SCALE    =  1.0
```

X contains the following sequence:

```
(1.0, 0.0)
(1.0, 0.0)
(1.0, 0.0)
(1.0, 0.0)
(1.0, 0.0)
(1.0, 0.0)
(1.0, 0.0)
```

Output

```
Y        =  (12.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 ,
             0.0 , 0.0 , 0.0 , 0.0)
```

Example 3: This example shows how to compute many transforms simultaneously. The arrays are declared as follows:

```
COMPLEX*8  X(0:8,2)
REAL*8     AUX1(50), AUX2(16), AUX3(1)
REAL*4     Y(0:15,2)
```

First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Call Statement and Input

```
           INIT  X INC2X Y INC2Y   N   M ISIGN SCALE  AUX1  NAUX1 AUX2  NAUX2 AUX3  NAUX3
            |    |   |   |   |     |   |   |     |     |      |    |      |    |      |
CALL SCRFT(INIT, X , 9 , Y , 16 , 16 , 2 , 1 , SCALE, AUX1 , 50 , AUX2 , 16 , AUX3 ,  0 )
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
SCALE    =  1.0
```

X contains the following two sequences:

```
(1.0, 0.0)  (0.0, 0.0)
(1.0, 0.0)  (0.0, 0.0)
(1.0, 0.0)  (0.0, 0.0)
(1.0, 0.0)  (0.0, 0.0)
(1.0, 0.0)  (0.0, 0.0)
(1.0, 0.0)  (0.0, 0.0)
(1.0, 0.0)  (0.0, 0.0)
(1.0, 0.0)  (0.0, 0.0)
(1.0, 0.0)  (1.0, 0.0)
```

Output: Y contains the following two sequences:

```
16.0   1.0
 0.0  -1.0
 0.0   1.0
 0.0  -1.0
 0.0   1.0
 0.0  -1.0
 0.0   1.0
 0.0  -1.0
 0.0   1.0
 0.0  -1.0
 0.0   1.0
 0.0  -1.0
 0.0   1.0
 0.0  -1.0
 0.0   1.0
 0.0  -1.0
```

Example 4: This example shows the same array being used for input and output. The arrays are declared as follows:

```
COMPLEX*16 X(0:8,2)
REAL*8     AUX1(50), AUX2(16)
REAL*8     Y(0:17,2)
```

Arrays X and Y are made equivalent by the following statement, making them occupy the same storage:

```
EQUIVALENCE (X,Y)
```

This requires INC2Y = 2(INC2X). First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Call Statement and Input

```
          INIT  X INC2X Y  INC2Y  N   M  ISIGN SCALE  AUX1  NAUX1 AUX2  NAUX2
           |    |   |   |    |    |   |    |     |     |      |    |      |
CALL DCRFT(INIT, X , 9 , Y , 18 , 16 , 2 , -1 , SCALE, AUX1 , 50 , AUX2 , 16)
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
SCALE    =  0.0625
```

X contains the following two sequences:

```
 (1.0,  0.0)   (1.0,  0.0)
 (0.0,  1.0)   (0.0, -1.0)
(-1.0,  0.0)  (-1.0,  0.0)
 (0.0, -1.0)   (0.0,  1.0)
 (1.0,  0.0)   (1.0,  0.0)
 (0.0,  1.0)   (0.0, -1.0)
(-1.0,  0.0)  (-1.0,  0.0)
 (0.0, -1.0)   (0.0,  1.0)
 (1.0,  0.0)   (1.0,0.0)
```

Output: Y contains the following two sequences:

```
0.0  0.0
0.0  0.0
0.0  0.0
0.0  0.0
0.0  1.0
0.0  0.0
0.0  0.0
0.0  0.0
0.0  0.0
0.0  0.0
0.0  0.0
0.0  0.0
1.0  0.0
0.0  0.0
0.0  0.0
0.0  0.0
```

SCOSF and DCOSF—Cosine Transform

These subroutines compute a set of *m* real even discrete *n*-point Fourier transforms of cosine sequences of real even data.

Table 129. Data Types

X, Y, *scale*	**Subroutine**
Short-precision real	SCOSF
Long-precision real	DCOSF

Note: Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.

Syntax

Fortran	CALL SCOSF | DCOSF (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *n*, *m*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	scosf | dcosf (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *n*, *m*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);
PL/I	CALL SCOSF | DCOSF (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *n*, *m*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init
is a flag, where:

If *init* ≠ 0, trigonometric functions and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*. The contents of *x* and *y* are not used or changed.

If *init* = 0, the discrete Fourier transforms of the given sequences are computed. The only arguments that may change after initialization are *x*, *y*, and *aux2*. All scalar arguments must be the same as when the subroutine was called for initialization with *init* ≠ 0.

Specified as: a fullword integer. It can have any value.

x
is the array X, consisting of *m* sequences of length *n*/2+1. Specified as: an array of (at least) length 1+(*n*/2)*inc1x*+(*m*−1)*inc2x*, containing numbers of the data type indicated in Table 129.

inc1x
is the stride between the elements within each sequence in array X. Specified as: a fullword integer; *inc1x* > 0.

inc2x
is the stride between the first elements of the sequences in array X. (If *m* = 1, this argument is ignored.) Specified as: a fullword integer; *inc2x* > 0.

y
See On Return.

inc1y
is the stride between the elements within each sequence in array Y. Specified as: a fullword integer; *inc1y* > 0.

inc2y
is the stride between the first elements of the sequences in array Y. (If $m = 1$, this argument is ignored.) Specified as: a fullword integer; $inc2y > 0$.

n
is the transform length. However, due to symmetry, only the first $n/2+1$ values are given in the input and output. Specified as: a fullword integer; $n \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 741. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 38.

m
is the number of sequences to be transformed. Specified as: a fullword integer; $m > 0$.

scale
is the scaling constant *scale*. See "Function" on page 777 for its usage. Specified as: a number of the data type indicated in Table 129 on page 774, where $scale > 0.0$ or $scale < 0.0$.

aux1
is the working storage for this subroutine, where:

If $init \neq 0$, the working storage is computed.

If $init = 0$, the working storage is used in the computation of the Fourier transforms.

Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1
is the number of doublewords in the working storage specified in *aux1*. Specified as: a fullword integer; *naux1* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 31.

aux2
has the following meaning:

If $naux2 = 0$ and error 2015 is unrecoverable, *aux2* is ignored.

Otherwise, it is the working storage used by this subroutine, which is available for use by the calling program between calls to this subroutine.

Specified as: an area of storage, containing *naux2* long-precision real numbers. On output, the contents are overwritten.

naux2
is the number of doublewords in the working storage specified in *aux2*. Specified as: a fullword integer, where:

If $naux2 = 0$ and error 2015 is unrecoverable, SCOSF and DCOSF dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux2* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 31.

On Return

y
has the following meaning, where:

If *init* ≠ 0, this argument is not used, and its contents remain unchanged.

If *init* = 0, this is array Y, consisting of the results of the *m* discrete Fourier transforms, where each Fourier transform is real and of length *n*. However, due to symmetry, only the first *n*/2+1 values are given in the output—that is, y_{ki}, $k = 0, 1, ..., n/2$ for each $i = 1, 2, ..., m$.

Returned as: an array of (at least) length 1+(*n*/2)*inc1y*+(*m*−1)*inc2y*, containing numbers of the data type indicated in Table 129 on page 774.

aux1
is the working storage for this subroutine, where:

If *init* ≠ 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If *init* = 0, its contents are unchanged.

Returned as: the contents are not relevant.

Notes

1. *aux1* should **not** be used by the calling program between calls to this subroutine with *init* ≠ 0 and *init* = 0. However, it can be reused after intervening calls to this subroutine with different arguments.
2. For optimal performance, the preferred value for *inc1x* and *inc1y* is 1. This implies that the sequences are stored with stride 1. In addition, *inc2x* and *inc2y* should be close to *n*/2+1.

   It is possible to specify sequences in the transposed form—that is, as rows of a two-dimensional array. In this case, *inc2x* (or *inc2y*) = 1 and *inc1x* (or *inc1y*) is equal to the leading dimension of the array. One can specify either input, output, or both in the transposed form by specifying appropriate values for the stride parameters. For selecting optimal values of *inc1x* and *inc1y* for _COSF, you should use "STRIDE—Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines" on page 977. Example 2 in the STRIDE subroutine description explains how it is used for _COSF.

   If you specify the same array for X and Y, then *inc1x* and *inc1y* must be equal, and *inc2x* and *inc2y* must be equal. In this case, output overwrites input. If *m* = 1, the *inc2x* and *inc2y* values are not used by the subroutine. If you specify different arrays for X and Y, they must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.

Processor-Independent Formulas for SCOSF for NAUX1 and NAUX2

NAUX1 Formulas:

If $n \leq 16384$, use *naux1* = 40000.
If $n > 16384$, use *naux1* = 20000+.30*n*.

NAUX2 Formulas:

If $n \leq 16384$, use *naux2* = 25000.
If $n > 16384$, use *naux2* = 20000+.32*n*.

For the transposed case, where *inc2x* = 1 or *inc2y* = 1, and where $n \geq 252$, add the following to the above storage requirements:
(n/4+257)(min(128, m)).

Processor-Independent Formulas for DCOSF for NAUX1 and NAUX2

NAUX1 Formulas:

If $n \leq 16384$, use *naux1* = 35000.
If $n > 16384$, use *naux1* = 20000+.60n.

NAUX2 Formulas:

If $n \leq 16384$, use *naux2* = 20000.
If $n > 16384$, use *naux2* = 20000+.64n.
For the transposed case, where *inc2x* = 1 or *inc2y* = 1, and where $n \geq 252$, add the following to the above storage requirements:
(n/2+257)(min(128, m)).

Function: The set of m real even discrete n-point Fourier transforms of the cosine sequences of real data in array X, with results going into array Y, is expressed as follows:

$$y_{ki} = scale\left(.5x_{0,i} + .5(-1)^k x_{n/2,i} + \sum_{j=1}^{n/2-1} x_{ji}\ \cos\left(jk(2\pi/n)\right)\right)$$

for:

$k = 0, 1, ..., n/2$
$i = 1, 2, ..., m$

where:

x_{ji} are elements of the sequences in array X, where each sequence contains the n/2+1 real nonredundant data x_{ji}, $j = 0, 1, ..., n/2$.
y_{ki} are elements of the sequences in array Y, where each sequence contains the n/2+1 real nonredundant data y_{ki}, $k = 0, 1, ..., n/2$.
scale is a scalar value.

You can obtain the inverse cosine transform by specifying *scale* = 4.0/n. Thus, if an X input is used with *scale* = 1.0, and its output is used as input on a subsequent call with *scale* = 4.0/n, the original X is obtained. See references [1], [4], [19], and [20].

Two invocations of this subroutine are necessary:

1. With *init* ≠ 0, the subroutine tests and initializes arguments of the program, setting up the *aux1* working storage.
2. With *init* = 0, the subroutine checks that the initialization arguments in the *aux1* working storage correspond to the present arguments, and if so, performs the calculation of the Fourier transforms.

These subroutines use a Fourier transform method with a mixed-radix capability. This provides maximum performance for your application.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux2* = 0, and unable to allocate work area.

Computational Errors: None

Input-Argument Errors

1. $n > 37748736$
2. *inc1x* or *inc1y* ≤ 0
3. *inc2x* or *inc2y* ≤ 0
4. $m \leq 0$
5. *scale* = 0.0
6. The subroutine has not been initialized with the present arguments.
7. The length of the transform in *n* is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
8. *naux1* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
9. Error 2015 is recoverable or *naux2*≠0, and *naux2* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example 1: This example shows an input array X with a set of *m* cosine sequences of length *n*/2+1, cos(*jk*(2π/*n*)), *j* = 0, 1, ..., *n*/2, with the single frequencies *k* = 0, 1, 2, 3. The Fourier transform of the cosine sequence with frequency *k* = 0 or *n*/2 has *n*/2 in the 0-th or *n*/2-th position, respectively, and zeros elsewhere. For all other *k*, the Fourier transform has *n*/4 in position *k* and zeros elsewhere. The arrays are declared as follows:

```
REAL*4     X(0:71),Y(0:71)
REAL*8     AUX1(414),AUX2(8960)
```

First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Call Statement and Input

```
          INIT  X INC1X INC2X Y INC1Y INC2Y  N   M   SCALE  AUX1  NAUX1  AUX2  NAUX2
           |    |   |     |   |   |     |    |   |    |      |      |     |      |
CALL SCOSF(INIT, X , 1  , 18 , Y , 1  , 18 , 32 , 4 , SCALE, AUX1 , 414 , AUX2 , 8960)
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
SCALE    =  1.0
```

X contains the following four sequences:

```
1.0000   1.0000   1.0000   1.0000
1.0000   0.9808   0.9239   0.8315
1.0000   0.9239   0.7071   0.3827
1.0000   0.8315   0.3827  -0.1951
1.0000   0.7071   0.0000  -0.7071
1.0000   0.5556  -0.3827  -0.9808
1.0000   0.3827  -0.7071  -0.9239
1.0000   0.1951  -0.9239  -0.5556
1.0000   0.0000  -1.0000   0.0000
1.0000  -0.1951  -0.9239   0.5556
1.0000  -0.3827  -0.7071   0.9239
1.0000  -0.5556  -0.3827   0.9808
1.0000  -0.7071   0.0000   0.7071
1.0000  -0.8315   0.3827   0.1951
1.0000  -0.9239   0.7071  -0.3827
1.0000  -0.9808   0.9239  -0.8315
1.0000  -1.0000   1.0000  -1.0000
 .        .        .        .
```

Output: Y contains the following four sequences:

```
16.0000  0.0000  0.0000  0.0000
 0.0000  8.0000  0.0000  0.0000
 0.0000  0.0000  8.0000  0.0000
 0.0000  0.0000  0.0000  8.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
  .       .       .       .
```

Example 2: This example shows an input array X with a set of four input spike sequences equal to the output of Example 1. This shows how you can compute the inverse of the transform in Example 1 by using *scale* = 4.0/*n*, giving as output the four sequences listed in the input for Example 1. First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Call Statement and Input

```
            INIT  X INC1X INC2X Y INC1Y INC2Y  N   M   SCALE  AUX1  NAUX1  AUX2  NAUX2
             |    |   |     |   |   |     |    |   |     |     |      |     |      |
CALL SCOSF(INIT, X , 1  , 18 , Y , 1  , 18 , 32 , 4 , SCALE, AUX1 , 414 , AUX2 , 8960)
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
```

```
SCALE    =  4.0/32
X        =(same sequences as in output Y in Example 1)
```

Output

```
Y        =(same sequences as in output X in Example 1)
```

Example 3: This example shows another computation using the same arguments initialized in Example 1 and using different input sequence data. The data for this example has frequencies $k = 14, 15, 16, 17$. Because only the sequence data has changed, initialization does not have to be done again.

Call Statement and Input

```
          INIT  X INC1X INC2X Y INC1Y INC2Y N   M   SCALE  AUX1  NAUX1  AUX2  NAUX2
           |   |   |     |   |   |     |    |   |     |     |      |     |      |
CALL SCOSF( 0 , X , 1  , 18 , Y , 1  , 18 , 32 , 4 , SCALE, AUX1 , 414 , AUX2 , 8960)
```

```
SCALE    =  1.0
```

X contains the following four sequences:

```
 1.0000   1.0000   1.0000   1.0000
-0.9239  -0.9808  -1.0000  -0.9808
 0.7071   0.9239   1.0000   0.9239
-0.3827  -0.8315  -1.0000  -0.8315
 0.0000   0.7071   1.0000   0.7071
 0.3827  -0.5556  -1.0000  -0.5556
-0.7071   0.3827   1.0000   0.3827
 0.9239  -0.1951  -1.0000  -0.1951
-1.0000   0.0000   1.0000   0.0000
 0.9239   0.1951  -1.0000   0.1951
-0.7071  -0.3827   1.0000  -0.3827
 0.3827   0.5556  -1.0000   0.5556
 0.0000  -0.7071   1.0000  -0.7071
-0.3827   0.8315  -1.0000   0.8315
 0.7071  -0.9239   1.0000  -0.9239
-0.9239   0.9808  -1.0000   0.9808
 1.0000  -1.0000   1.0000  -1.0000
   .        .        .        .
```

Output: Y contains the following four sequences:

```
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
0.0000  0.0000   0.0000  0.0000
8.0000  0.0000   0.0000  0.0000
0.0000  8.0000   0.0000  8.0000
0.0000  0.0000  16.0000  0.0000
 .       .        .       .
```

SSINF and DSINF—Sine Transform

These subroutines compute a set of *m* real even discrete *n*-point Fourier transforms of sine sequences of real even data.

Table 130. Data Types	
X, Y, *scale*	**Subroutine**
Short-precision real	SSINF
Long-precision real	DSINF

Note: Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.

Syntax

Fortran	CALL SSINF | DSINF (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *n*, *m*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	ssinf | dsinf (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *n*, *m*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);
PL/I	CALL SSINF | DSINF (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *n*, *m*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init
is a flag, where:

If *init* ≠ 0, trigonometric functions and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*. The contents of *x* and *y* are not used or changed.

If *init* = 0, the discrete Fourier transforms of the given sequences are computed. The only arguments that may change after initialization are *x*, *y*, and *aux2*. All scalar arguments must be the same as when the subroutine was called for initialization with *init* ≠ 0.

Specified as: a fullword integer. It can have any value.

x
is the array X, consisting of *m* sequences of length *n*/2. Specified as: an array of (at least) length 1+(*n* / 2−1)*inc1x*+(*m*−1)*inc2x*, containing numbers of the data type indicated in Table 130. The first element in X must have a value of 0.0 (otherwise, incorrect results may occur).

inc1x
is the stride between the elements within each sequence in array X. Specified as: a fullword integer; *inc1x* > 0.

inc2x
is the stride between the first elements of the sequences in array X. (If *m* = 1, this argument is ignored.) Specified as: a fullword integer; *inc2x* > 0.

y
See On Return.

inc1y
is the stride between the elements within each sequence in array Y. Specified as: a fullword integer; *inc1y* > 0.

inc2y
is the stride between the first elements of the sequences in array Y. (If $m = 1$, this argument is ignored.) Specified as: a fullword integer; $inc2y > 0$.

n
is the transform length. However, due to symmetry, only the first $n/2$ values are given in the input and output. Specified as: a fullword integer; $n \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 741. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 38.

m
is the number of sequences to be transformed. Specified as: a fullword integer; $m > 0$.

scale
is the scaling constant *scale*. See "Function" on page 785 for its usage. Specified as: a number of the data type indicated in Table 130 on page 782, where $scale > 0.0$ or $scale < 0.0$.

aux1
is the working storage for this subroutine, where:

If $init \neq 0$, the working storage is computed.

If $init = 0$, the working storage is used in the computation of the Fourier transforms.

Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1
is the number of doublewords in the working storage specified in *aux1*. Specified as: a fullword integer; *naux1* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 31.

aux2
has the following meaning:

If $naux2 = 0$ and error 2015 is unrecoverable, *aux2* is ignored.

Otherwise, it is the working storage used by this subroutine, which is available for use by the calling program between calls to this subroutine.

Specified as: an area of storage, containing *naux2* long-precision real numbers. On output, the contents are overwritten.

naux2
is the number of doublewords in the working storage specified in *aux2*. Specified as: a fullword integer, where:

If $naux2 = 0$ and error 2015 is unrecoverable, SSINF and DSINF dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux2* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 31.

On Return

y
has the following meaning, where:

If *init* ≠ 0, this argument is not used, and its contents remain unchanged.

If *init* = 0, this is array Y, consisting of the results of the *m* discrete Fourier transforms, where each Fourier transform is real and of length *n*. However, due to symmetry, only the first *n*/2 values are given in the output—that is, y_{ki}, $k = 0, 1, ..., n/2-1$ for each $i = 1, 2, ..., m$.

Returned as: an array of (at least) length $1+(n / 2-1)inc1y+(m-1)inc2y$, containing numbers of the data type indicated in Table 130 on page 782.

aux1
is the working storage for this subroutine, where:

If *init* ≠ 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If *init* = 0, its contents are unchanged.

Returned as: the contents are not relevant.

Notes

1. *aux1* should **not** be used by the calling program between calls to this subroutine with *init* ≠ 0 and *init* = 0. However, it can be reused after intervening calls to this subroutine with different arguments.
2. For optimal performance, the preferred value for *inc1x* and *inc1y* is 1. This implies that the sequences are stored with stride 1. In addition, *inc2x* and *inc2y* should be close to *n*/2.

   It is possible to specify sequences in the transposed form—that is, as rows of a two-dimensional array. In this case, *inc2x* (or *inc2y*) = 1 and *inc1x* (or *inc1y*) is equal to the leading dimension of the array. One can specify either input, output, or both in the transposed form by specifying appropriate values for the stride parameters. For selecting optimal values of *inc1x* and *inc1y* for _SINF, you should use "STRIDE—Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines" on page 977. Example 3 in the STRIDE subroutine description explains how it is used for _SINF.

   If you specify the same array for X and Y, then *inc1x* and *inc1y* must be equal, and *inc2x* and *inc2y* must be equal. In this case, output overwrites input. If *m* = 1, the *inc2x* and *inc2y* values are not used by the subroutine. If you specify different arrays for X and Y, they must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.

Processor-Independent Formulas for SSINF for NAUX1 and NAUX2

NAUX1 Formulas:

If $n \leq 16384$, use $naux1 = 60000$.
If $n > 16384$, use $naux1 = 20000+.30n$.

NAUX2 Formulas:

If $n \leq 16384$, use $naux2 = 25000$.
If $n > 16384$, use $naux2 = 20000+.32n$.

For the transposed case, where *inc2x* = 1 or *inc2y* = 1, and where $n \geq 252$, add the following to the above storage requirements:
(n/4+257)(min(128, m)).

Processor-Independent Formulas for DSINF for NAUX1 and NAUX2

NAUX1 Formulas:

If $n \leq 16384$, use *naux1* = 50000.
If $n > 16384$, use *naux1* = 20000+.60n.

NAUX2 Formulas:

If $n \leq 16384$, use *naux2* = 20000.
If $n > 16384$, use *naux2* = 20000+.64n.
For the transposed case, where *inc2x* = 1 or *inc2y* = 1, and where $n \geq 252$, add the following to the above storage requirements:
(n/2+257)(min(128, m)).

Function: The set of m real even discrete n-point Fourier transforms of the sine sequences of real data in array X, with results going into array Y, is expressed as follows:

$$y_{ki} = scale \sum_{j=0}^{n/2-1} x_{ji} \sin\left(jk\left(2\pi / n\right)\right)$$

for:

$k = 0, 1, ..., n/2-1$
$i = 1, 2, ..., m$

where:

$x_{0i} = 0.0$
x_{ji} are elements of the sequences in array X, where each sequence contains the n/2 real nonredundant data x_{ji}, $j = 0, 1, ..., n/2-1$.
y_{ki} are elements of the sequences in array Y, where each sequence contains the n/2 real nonredundant data y_{ki}, $k = 0, 1, ..., n/2-1$.
scale is a scalar value.

You can obtain the inverse sine transform by specifying *scale* = 4.0/n. Thus, if an X input is used with *scale* = 1.0, and its output is used as input on a subsequent call with *scale* = 4.0/n, the original X is obtained. See references [1], [4], [19], and [20].

Two invocations of this subroutine are necessary:

1. With *init* ≠ 0, the subroutine tests and initializes arguments of the program, setting up the *aux1* working storage.
2. With *init* = 0, the subroutine checks that the initialization arguments in the *aux1* working storage correspond to the present arguments, and if so, performs the calculation of the Fourier transforms.

These subroutines use a Fourier transform method with a mixed-radix capability. This provides maximum performance for your application.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux2* = 0, and unable to allocate work area.

Computational Errors: None

Input-Argument Errors

1. $n > 37748736$
2. *inc1x* or *inc1y* ≤ 0
3. *inc2x* or *inc2y* ≤ 0
4. $m \leq 0$
5. *scale* = 0.0
6. The subroutine has not been initialized with the present arguments.
7. The length of the transform in *n* is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
8. *naux1* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
9. Error 2015 is recoverable or *naux2*≠0, and *naux2* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example 1: This example shows an input array X with a set of *m* sine sequences of length *n*/2, $\sin(jk(2\pi/n))$, $j = 0, 1, ..., n/2-1$, with the single frequencies $k = 1, 2, 3$. The Fourier transform of the sine sequence has *n*/4 in position *k* and zeros elsewhere. The arrays are declared as follows:

```
REAL*4     X(0:53),Y(0:53)
REAL*8     AUX1(414),AUX2(8960)
```

First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Call Statement and Input

```
          INIT  X INC1X INC2X Y INC1Y INC2Y  N   M   SCALE  AUX1  NAUX1  AUX2   NAUX2
           |    |   |     |   |   |     |    |   |     |     |      |     |      |
CALL SSINF(INIT, X , 1  , 18 , Y , 1  , 18 , 32 , 3 , SCALE, AUX1 , 414 , AUX2 , 8960)
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
SCALE    =  1.0
```

X contains the following three sequences:

```
0.0000   0.0000   0.0000
0.1951   0.3827   0.5556
0.3827   0.7071   0.9239
0.5556   0.9239   0.9808
0.7071   1.0000   0.7071
0.8315   0.9239   0.1951
0.9239   0.7071  -0.3827
0.9808   0.3827  -0.8315
1.0000   0.0000  -1.0000
0.9808  -0.3827  -0.8315
0.9239  -0.7071  -0.3827
0.8315  -0.9239   0.1951
0.7071  -1.0000   0.7071
0.5556  -0.9239   0.9808
0.3827  -0.7071   0.9239
0.1951  -0.3827   0.5556
 .        .        .
 .        .        .
```

Output: Y contains the following three sequences:

```
0.0000  0.0000  0.0000
8.0000  0.0000  0.0000
0.0000  8.0000  0.0000
0.0000  0.0000  8.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
 .       .       .
 .       .       .
```

Example 2: This example shows an input array X with a set of three input spike sequences equal to the output of Example 1. This shows how you can compute the inverse of the transform in Example 1 by using *scale* = 4.0/*n*, giving as output the three sequences listed in the input for Example 1. First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Call Statement and Input

```
           INIT  X INC1X INC2X Y INC1Y INC2Y  N   M   SCALE  AUX1  NAUX1  AUX2  NAUX2
            |    |   |     |   |   |     |    |   |     |     |      |     |      |
CALL SSINF(INIT, X , 1  , 18 , Y , 1  , 18 , 32 , 3 , SCALE, AUX1 , 414 , AUX2 , 8960)
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
```

```
SCALE    =  4.0/32
X        =(same sequences as in output Y in Example 1)
```

Output

```
Y        =(same sequences as in output X in Example 1)
```

Example 3: This example shows another computation using the same arguments initialized in Example 1 and using different input sequence data. The data for this example has frequencies $k = 14, 15, 17$. Because only the sequence data has changed, initialization does not have to be done again.

Call Statement and Input

```
          INIT X INC1X INC2X Y INC1Y INC2Y  N   M   SCALE  AUX1  NAUX1  AUX2  NAUX2
           |   |   |     |   |   |     |    |   |     |     |      |     |       |
CALL SSINF( 0 , X , 1  , 18 , Y , 1  , 18 , 32 , 3 , SCALE, AUX1 , 414 , AUX2 , 8960)
```

```
SCALE    =  1.0
```

X contains the following three sequences:

```
 0.0000   0.0000   0.0000
 0.3827   0.1951  -0.1951
-0.7071  -0.3827   0.3827
 0.9239   0.5556  -0.5556
-1.0000  -0.7071   0.7071
 0.9239   0.8315  -0.8315
-0.7071  -0.9239   0.9239
 0.3827   0.9808  -0.9808
 0.8573  -1.0000   1.0000
-0.3827   0.9808  -0.9808
 0.7071  -0.9239   0.9239
-0.9239   0.8315  -0.8315
 1.0000  -0.7071   0.7071
-0.9239   0.5556  -0.5556
 0.7071  -0.3827   0.3827
-0.3827   0.1951  -0.1951
   .        .        .
   .        .        .
```

Output: Y contains the following three sequences:

```
0.0000  0.0000   0.0000
0.0000  0.0000   0.0000
0.0000  0.0000   0.0000
0.0000  0.0000   0.0000
0.0000  0.0000   0.0000
0.0000  0.0000   0.0000
0.0000  0.0000   0.0000
0.0000  0.0000   0.0000
0.0000  0.0000   0.0000
0.0000  0.0000   0.0000
0.0000  0.0000   0.0000
0.0000  0.0000   0.0000
0.0000  0.0000   0.0000
8.0000  0.0000   0.0000
0.0000  8.0000  -8.0000
0.0000  0.0000   0.0000
 .       .        .
 .       .        .
```

SCFT2 and DCFT2—Complex Fourier Transform in Two Dimensions

These subroutines compute the two-dimensional discrete Fourier transform of complex data.

Table 131. Data Types

X, Y	*scale*	**Subroutine**
Short-precision complex	Short-precision real	SCFT2
Long-precision complex	Long-precision real	DCFT2

Note: Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.

Syntax

Fortran	CALL SCFT2 | DCFT2 (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	scft2 | dcft2 (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);
PL/I	CALL SCFT2 | DCFT2 (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init
is a flag, where:

If *init* ≠ 0, trigonometric functions and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*. The contents of *x* and *y* are not used or changed.

If *init* = 0, the discrete Fourier transform of the given array is computed. The only arguments that may change after initialization are *x*, *y*, and *aux2*. All scalar arguments must be the same as when the subroutine was called for initialization with *init* ≠ 0.

Specified as: a fullword integer. It can have any value.

x
is the array X, containing the two-dimensional data to be transformed, where each element $x_{j1,j2}$, using zero-based indexing, is stored in X(*j1*(*inc1x*)+*j2*(*inc2x*)) for *j1* = 0, 1, ..., *n1*−1 and *j2* = 0, 1, ..., *n2*−1.

Specified as: an array of (at least) length 1+(*n1*−1)*inc1x*+(*n2*−1)*inc2x*, containing numbers of the data type indicated in Table 131. This array must be aligned on a doubleword boundary, and:

If *inc1x* = 1, the input array is stored in normal form, and *inc2x* ≥ *n1*.

If *inc2x* = 1, the input array is stored in transposed form, and *inc1x* ≥ *n2*.

See “Notes” on page 793 for more details.

inc1x
is the stride between the elements in array X for the first dimension.

If the array is stored in the normal form, *inc1x* = 1.

If the array is stored in the transposed form, *inc1x* is the leading dimension of the array and *inc1x* ≥ *n2*.

Specified as: a fullword integer; *inc1x* > 0. If *inc2x* = 1, then *inc1x* ≥ *n2*.

inc2x
is the stride between the elements in array X for the second dimension.

If the array is stored in the transposed form, *inc2x* = 1.

If the array is stored in the normal form, *inc2x* is the leading dimension of the array and *inc2x* ≥ *n1*.

Specified as: a fullword integer; *inc2x* > 0. If *inc1x* = 1, then *inc2x* ≥ *n1*.

y
See On Return.

inc1y
is the stride between the elements in array Y for the first dimension.

If the array is stored in the normal form, *inc1y* = 1.

If the array is stored in the transposed form, *inc1y* is the leading dimension of the array and *inc1y* ≥ *n2*.

Specified as: a fullword integer; *inc1y* > 0. If *inc2y* = 1, then *inc1y* ≥ *n2*.

inc2y
is the stride between the elements in array Y for the second dimension.

If the array is stored in the transposed form, *inc2y* = 1.

If the array is stored in the normal form, *inc2y* is the leading dimension of the array and *inc2y* ≥ *n1*.

Specified as: a fullword integer; *inc2y* > 0. If *inc1y* = 1, then *inc2y* ≥ *n1*.

n1
is the length of the first dimension of the two-dimensional data in the array to be transformed. Specified as: a fullword integer; *n1* ≤ 37748736 and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 741. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 38.

n2
is the length of the second dimension of the two-dimensional data in the array to be transformed. Specified as: a fullword integer; *n2* ≤ 37748736 and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 741. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 38.

isign
controls the direction of the transform, determining the sign *Isign* of the exponents of W_{n1} and W_{n2}, where:

If *isign* = positive value, *Isign* = + (transforming time to frequency).

If *isign* = negative value, *Isign* = − (transforming frequency to time).

Specified as: a fullword integer; *isign* > 0 or *isign* < 0.

scale
is the scaling constant *scale*. See "Function" on page 794 for its usage. Specified as: a number of the data type indicated in Table 131 on page 790, where *scale* > 0.0 or *scale* < 0.0.

aux1
is the working storage for this subroutine, where:

If *init* ≠ 0, the working storage is computed.

If *init* = 0, the working storage is used in the computation of the Fourier transforms.

Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1
is the number of doublewords in the working storage specified in *aux1*. Specified as: a fullword integer; *naux1* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 31.

aux2
has the following meaning:

If *naux2* = 0 and error 2015 is unrecoverable, *aux2* is ignored.

Otherwise, it is the working storage used by this subroutine, which is available for use by the calling program between calls to this subroutine.

Specified as: an area of storage, containing *naux2* long-precision real numbers. On output, the contents are overwritten.

naux2
is the number of doublewords in the working storage specified in *aux2*. Specified as: a fullword integer, where:

If *naux2* = 0 and error 2015 is unrecoverable, SCFT2 and DCFT2 dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux2* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 31.

On Return

y
has the following meaning, where:

If *init* ≠ 0, this argument is not used, and its contents remain unchanged.

If *init* = 0, this is array Y, containing the elements resulting from the two-dimensional discrete Fourier transform of the data in X. Each element $y_{k1,k2}$, using zero-based indexing, is stored in Y(*k1*(*inc1y*)+*k2*(*inc2y*)) for *k1* = 0, 1, ..., *n1*−1 and *k2* = 0, 1, ..., *n2*−1.

Returned as: an array of (at least) length 1+(*n1*−1)*inc1y*+(*n2*−1)*inc2y*, containing numbers of the data type indicated in Table 131 on page 790. This array must be aligned on a doubleword boundary, and:

If *inc1y* = 1, the output array is stored in normal form, and *inc2y* ≥ *n1*.

If *inc2y* = 1, the output array is stored in transposed form, and *inc1y* ≥ *n2*.

See "Notes" for more details.

aux1

is the working storage for this subroutine, where:

If *init* ≠ 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If *init* = 0, its contents are unchanged.

Returned as: the contents are not relevant.

Notes

1. *aux1* should **not** be used by the calling program between program calls to this subroutine with *init* ≠ 0 and *init* = 0. However, it can be reused after intervening calls to this subroutine with different arguments.
2. If you specify the same array for X and Y, then *inc1x* must equal *inc1y*, and *inc2x* must equal *inc2y*. In this case, output overwrites input. If you specify different arrays X and Y, they must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.
3. By appropriately specifying the *inc* arguments, this subroutine allows you to specify that it should use one of two forms of its arrays, the normal untransposed form or the transposed form. As a result, you **do not have to move any data**. Instead, the subroutine performs the adjustments for you. Also, either the input array or the output array can be in transposed form. The FFT computation is symmetrical with respect to *n1* and *n2*. They can be interchanged without the loss of generality. If they are interchanged, an array that is stored in the normal form appears as an array stored in the transposed form and vise versa. If, for performance reasons, the forms of the input and output arrays are different, then the input array should be specified in the normal form, and the output array should be specified in the transposed form. This can always be done by interchanging *n1* and *n2*.
4. Although the *inc* arguments for each array can be arbitrary, in most cases, one of the *inc* arguments is 1 for each array. If *inc1* = 1, the array is stored in normal form; that is, the first dimension of the array is along the columns. In this case, *inc2* is the leading dimension of the array and must be at least *n1*. Conversely, if *inc2* = 1, the array is stored in the transposed form; that is, the first dimension of the array is along the rows. In this case, *inc1* is the leading dimension of the array and must be at least *n2*. The rows of the arrays are accessed with a stride that equals the leading dimension of the array. To minimize cache interference in accessing a row, an optimal value should be used for the leading dimension of the array. You should use "STRIDE—Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines" on page 977 to determine this optimal value. Example 4 in the STRIDE subroutine description explains how it is used to find either *inc1* or *inc2*.

Processor-Independent Formulas for SCFT2 for NAUX1 and NAUX2: The required values of *naux1* and *naux2* depend on *n1* and *n2*.

NAUX1 Formulas

If max(*n1*, *n2*) ≤ 8192, use *naux1* = 40000.
If max(*n1*, *n2*) > 8192, use *naux1* = 40000+1.14(*n1*+*n2*).

NAUX2 Formulas

If max(*n1*, *n2*) < 252, use *naux2* = 20000.
If max(*n1*, *n2*) ≥ 252, use *naux2* = 20000+(*r*+256)(*s*+1.14), where *r* = max(*n1*, *n2*) and *s* = min(64, *n1*, *n2*).

Processor-Independent Formulas for DCFT2 for NAUX1 and NAUX2: The required values of *naux1* and *naux2* depend on *n1* and *n2*.

NAUX1 Formulas

If max(*n1*, *n2*) ≤ 2048, use *naux1* = 40000.
If max(*n1*, *n2*) > 2048, use *naux1* = 40000+2.28(*n1*+*n2*).

NAUX2 Formulas

If max(*n1*, *n2*) < 252, use *naux2* = 20000.
If max(*n1*, *n2*) ≥ 252, use *naux2* = 20000+(2*r*+256)(*s*+2.28), where *r* = max(*n1*, *n2*) and *s* = min(64, *n1*, *n2*).

Function: The two-dimensional discrete Fourier transform of complex data in array X, with results going into array Y, is expressed as follows:

$$y_{k1,k2} = scale \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} x_{j1,j2}\ W_{n1}^{(Isign)j1k1}\ W_{n2}^{(Isign)j2k2}$$

for:

k1 = 0, 1, ..., *n1*−1
k2 = 0, 1, ..., *n2*−1

where:

$$W_{n1} = e^{-2\pi\left(\sqrt{-1}\right)/n1}$$
$$W_{n2} = e^{-2\pi\left(\sqrt{-1}\right)/n2}$$

and where:

$x_{j1,j2}$ are elements of array X.
$y_{k1,k2}$ are elements of array Y.
Isign is + or − (determined by argument *isign*).
scale is a scalar value.

For *scale* = 1.0 and *isign* being positive, you obtain the discrete Fourier transform, a function of frequency. The inverse Fourier transform is obtained with *scale* = 1.0/((*n1*)(*n2*)) and *isign* being negative. See references [1], [4], and [20].

Two invocations of this subroutine are necessary:

1. With *init* ≠ 0, the subroutine tests and initializes arguments of the program, setting up the *aux1* working storage.

2. With *init* = 0, the subroutine checks that the initialization arguments in the *aux1* working storage correspond to the present arguments, and if so, performs the calculation of the Fourier transform.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux2* = 0, and unable to allocate work area.

Computational Errors: None

Input-Argument Errors

1. *n1* > 37748736
2. *n2* > 37748736
3. *inc1x*|*inc2x*|*inc1y*|*inc2y* ≤ 0
4. *scale* = 0.0
5. *isign* = 0
6. The subroutine has not been initialized with the present arguments.
7. The length of one of the transforms in *n1* or *n2* is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
8. *naux1* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
9. Error 2015 is recoverable or *naux2*≠0, and *naux2* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example 1: This example shows how to compute a two-dimensional transform where both input and output are stored in normal form (*inc1x* = *inc1y* = 1). Also, *inc2x* = *inc2y* so the same array can be used for both input and output. The arrays are declared as follows:

```
COMPLEX*8  X(6,8),Y(6,8)
REAL*8     AUX1(20000), AUX2(10000)
```

Arrays `X` and `Y` are made equivalent by the following statement, making them occupy the same storage: `EQUIVALENCE (X,Y)`. First, initialize `AUX1` using the calling sequence shown below with `INIT` ≠ 0. Then use the same calling sequence with `INIT` = 0 to do the calculation.

Call Statement and Input

```
           INIT  X INC1X INC2X Y INC1Y INC2Y N1  N2 ISIGN SCALE  AUX1  NAUX1   AUX2  NAUX2
            |    |   |     |   |   |     |   |   |    |     |      |     |      |      |
CALL SCFT2(INIT, X , 1  ,  6 , Y , 1  ,  6 , 6 , 8 ,  1 , SCALE, AUX1, 20000 , AUX2, 10000)
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
SCALE    =  1.0
```

`X` is an array with 6 rows and 8 columns with (1.0, 0.0) in all locations.

Output: `Y` is an array with 6 rows and 8 columns having (48.0, 0.0) in location `Y(1,1)` and (0.0, 0.0) in all others.

Example 2: This example shows how to compute a two-dimensional inverse Fourier transform. For this example, X is stored in normal untransposed form (*inc1x* = 1), and Y is stored in transposed form (*inc2y* = 1). The arrays are declared as follows:

```
COMPLEX*16  X(6,8),Y(8,6)
REAL*8      AUX1(20000), AUX2(10000)
```

First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Call Statement and Input

```
          INIT  X INC1X INC2X Y INC1Y INC2Y N1  N2 ISIGN SCALE  AUX1   NAUX1   AUX2   NAUX2
           |    |   |     |   |   |     |   |   |    |     |     |       |      |       |
CALL DCFT2(INIT, X , 1  ,  6 , Y , 8  ,  1 , 6 , 8 , -1 , SCALE, AUX1 , 20000 , AUX2 , 10000)
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
SCALE    =  1.0/48.0
X        =(same as output Y in Example 1)
```

Output: Y is an array with 8 rows and 6 columns with (1.0, 0.0) in all locations.

SRCFT2 and DRCFT2—Real-to-Complex Fourier Transform in Two Dimensions

These subroutines compute the two-dimensional discrete Fourier transform of real data in a two-dimensional array.

Table 132. Data Types

X, *scale*	Y	**Subroutine**
Short-precision real	Short-precision complex	SRCFT2
Long-precision real	Long-precision complex	DRCFT2

Note: Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.

Syntax

Fortran	CALL SRCFT2 (*init*, *x*, *inc2x*, *y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*, *aux3*, *naux3*) CALL DRCFT2 (*init*, *x*, *inc2x*, *y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	srcft2 (*init*, *x*, *inc2x*, *y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*, *aux3*, *naux3*); drcft2 (*init*, *x*, *inc2x*, *y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);
PL/I	CALL SRCFT2 (*init*, *x*, *inc2x*, *y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*, *aux3*, *naux3*); CALL DRCFT2 (*init*, *x*, *inc2x*, *y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init
is a flag, where:

If *init* ≠ 0, trigonometric functions and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*. The contents of *x* and *y* are not used or changed.

If *init* = 0, the discrete Fourier transform of the given array is computed. The only arguments that may change after initialization are *x*, *y*, and *aux2*. All scalar arguments must be the same as when the subroutine was called for initialization with *init* ≠ 0.

Specified as: a fullword integer. It can have any value.

x
is the array X, containing *n1* rows and *n2* columns of data to be transformed. The data in each column is stored with stride 1. Specified as: an *inc2x* by (at least) *n2* array, containing numbers of the data type indicated in Table 132. See "Notes" on page 799 for more details.

inc2x
is the leading dimension (stride between columns) of array X. Specified as: a fullword integer; *inc2x* ≥ *n1*.

y
See On Return.

inc2y
is the leading dimension (stride between columns) of array Y. Specified as: a fullword integer; *inc2y* ≥ ((*n1*)/2)+1.

n1
is the number of rows of data—that is, the length of the columns in array X involved in the computation. The length of the columns in array Y are (*n1*)/2+1. Specified as: a fullword integer; *n1* ≤ 37748736 and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 741. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 38.

n2
is the number of columns of data—that is, the length of the rows in arrays X and Y involved in the computation. Specified as: a fullword integer; *n2* ≤ 37748736 and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 741. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 38.

isign
controls the direction of the transform, determining the sign *Isign* of the exponents of W_{n1} and W_{n2}, where:

If *isign* = positive value, *Isign* = + (transforming time to frequency).

If *isign* = negative value, *Isign* = − (transforming frequency to time).

Specified as: a fullword integer; *isign* > 0 or *isign* < 0.

scale
is the scaling constant *scale*. See "Function" on page 800 for its usage. Specified as: a number of the data type indicated in Table 132 on page 797, where *scale* > 0.0 or *scale* < 0.0.

aux1
is the working storage for this subroutine, where:

If *init* ≠ 0, the working storage is computed.

If *init* = 0, the working storage is used in the computation of the Fourier transforms.

Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1
is the number of doublewords in the working storage specified in *aux1*. Specified as: a fullword integer; *naux1* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 31.

aux2
has the following meaning:

If *naux2* = 0 and error 2015 is unrecoverable, *aux2* is ignored.

Otherwise, it is the working storage used by this subroutine, which is available for use by the calling program between calls to this subroutine.

Specified as: an area of storage, containing *naux2* long-precision real numbers. On output, the contents are overwritten.

naux2
is the number of doublewords in the working storage specified in *aux2*. Specified as: a fullword integer, where:

If *naux2* = 0 and error 2015 is unrecoverable, SRCFT2 and DRCFT2 dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux2* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 31.

aux3
this argument is provided for migration purposes only and is ignored.

Specified as: an area of storage containing *naux3* long-precision real numbers.

naux3
this argument is provided for migration purposes only and is ignored.

Specified as: a fullword integer.

On Return

y
has the following meaning, where:

If *init* ≠ 0, this argument is not used, and its contents remain unchanged.

If *init* = 0, this is array Y, containing the results of the complex discrete Fourier transform of X. The output consists of *n2* columns of data. The data in each column is stored with stride 1. Due to complex conjugate symmetry, the output consists of only the first ((*n1*)/2)+1 rows of the array—that is, $y_{k1,k2}$, where *k1* = 0, 1, ..., (*n1*)/2 and *k2* = 0, 1, ..., *n2*−1.

Returned as: an *inc2y* by (at least) *n2* array, containing numbers of the data type indicated in Table 132 on page 797. This array must be aligned on a doubleword boundary.

aux1
is the working storage for this subroutine, where:

If *init* ≠ 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If *init* = 0, its contents are unchanged.

Returned as: the contents are not relevant.

Notes

1. *aux1* should **not** be used by the calling program between calls to this subroutine with *init* ≠ 0 and *init* = 0. However, it can be reused after intervening calls to this subroutine with different arguments.
2. If you specify the same array for X and Y, then *inc2x* must equal (2)(*inc2y*). In this case, output overwrites input. If you specify different arrays X and Y, they must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.

3. For selecting optimal strides (or leading dimensions *inc2x* and *inc2y*) for your input and output arrays, you should use "STRIDE—Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines" on page 977. Example 5 in the STRIDE subroutine description explains how it is used for these subroutines.
4. Be sure to align array X on a doubleword boundary, and specify an even number for *inc2x*, if possible.

Processor-Independent Formulas for SRCFT2 for NAUX1 and NAUX2: The required values of *naux1* and *naux2* depend on *n1* and *n2*.

NAUX1 Formulas

If max(*n1*/2, *n2*) ≤ 8192, use *naux1* = 45000.
If max(*n1*/2, *n2*) > 8192, use *naux1* = 40000+0.82*n1*+1.14*n2*.

NAUX2 Formulas

If *n1* ≤ 16384 and *n2* < 252, use *naux2* = 20000.
If *n1* > 16384 and *n2* < 252, use *naux2* = 20000+0.57*n1*.
If *n2* ≥ 252, add the following to the above storage requirements:
(*n2*+256)(1.14+*s*)
where *s* = min(64, 1+*n1*/2).

Processor-Independent Formulas for DRCFT2 for NAUX1 and NAUX2: The required values of *naux1* and *naux2* depend on *n1* and *n2*.

NAUX1 Formulas

If *n* ≤ 2048, use *naux1* = 42000.
If *n* > 2048, use *naux1* = 40000+1.64*n1*+2.28*n2*,
where *n* = max(*n1*/2, *n2*).

NAUX2 Formulas

If *n1* ≤ 4096 and *n2* < 252, use *naux2* = 20000.
If *n1* > 4096 and *n2* < 252, use *naux2* = 20000+1.14*n1*.
If *n2* ≥ 252, add the following to the above storage requirements:
((2)*n2*+256) (2.28+*s*)
where *s* = min(64, 1+*n1*/2).

Function: The two-dimensional complex conjugate even discrete Fourier transform of real data in array X, with results going into array Y, is expressed as follows:

$$y_{k1,k2} = scale \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} x_{j1,j2}\, W_{n1}^{(Isign)j1k1}\, W_{n2}^{(Isign)j2k2}$$

for:

k1 = 0, 1, ..., *n1*−1
k2 = 0, 1, ..., *n2*−1

where:

$$W_{n1} = e^{-2\pi\left(\sqrt{-1}\right)/n1}$$
$$W_{n2} = e^{-2\pi\left(\sqrt{-1}\right)/n2}$$

and where:

$x_{j1,j2}$ are elements of array X.
$y_{k1,k2}$ are elements of array Y.
Isign is + or − (determined by argument *isign*).
scale is a scalar value.

The output in array Y is complex. For *scale* = 1.0 and *isign* being positive, you obtain the discrete Fourier transform, a function of frequency. The inverse Fourier transform is obtained with *scale* = 1.0/((*n1*)(*n2*)) and *isign* being negative. See references [1], [4], [19], and [20].

Two invocations of this subroutine are necessary:

1. With *init* ≠ 0, the subroutine tests and initializes arguments of the program, setting up the *aux1* working storage.
2. With *init* = 0, the subroutine checks that the initialization arguments in the *aux1* working storage correspond to the present arguments, and if so, performs the calculation of the Fourier transform.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux2* = 0, and unable to allocate work area.

Computational Errors: None

Input-Argument Errors

1. *n1* > 37748736
2. *n2* > 37748736
3. *inc2x* < *n1*
4. *inc2y* < (*n1*)/2+1
5. *scale* = 0.0
6. *isign* = 0
7. The subroutine has not been initialized with the present arguments.
8. The length of one of the transforms in *n1* or *n2* is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
9. *naux1* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
10. Error 2015 is recoverable or *naux2*≠0, and *naux2* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example 1: This example shows how to compute a two-dimensional transform. The arrays are declared as follows:

```
COMPLEX*8  Y(0:6,0:7)
REAL*4     X(0:11,0:7)
REAL*8     AUX1(1000), AUX2(1000), AUX3(1)
```

First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Call Statement and Input

```
            INIT  X  INC2X Y INC2Y N1  N2 ISIGN SCALE  AUX1   NAUX1  AUX2   NAUX2  AUX3  NAUX3
             |    |    |   |   |    |   |   |     |      |       |     |       |     |      |
CALL SRCFT2(INIT, X , 12 , Y , 7 , 12 , 8 , 1 , SCALE, AUX1 , 1000 , AUX2 , 1000 , AUX3 ,  0 )
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
SCALE    =  1.0
```

X is an array with 12 rows and 8 columns having 1.0 in location X(0,0) and 0.0 in all others.

Output: Y is an array with 7 rows and 8 columns with (1.0, 0.0) in all locations.

Example 2: This example shows another transform computation with different data using the same initialized array AUX1 in Example 1.

Call Statement and Input

```
            INIT X  INC2X Y INC2Y N1   N2 ISIGN SCALE  AUX1  NAUX1  AUX2  NAUX2  AUX3  NAUX3
             |   |    |   |   |   |    |    |     |     |      |     |      |     |      |
CALL SRCFT2( 0 , X , 12 , Y , 7 , 12 , 8 ,  1 , SCALE, AUX1, 1000 , AUX2, 1000 , AUX3 ,  0 )
```

```
SCALE    =  1.0
```

X is an array with 12 rows and 8 columns with 1.0 in all locations.

Output: Y is an array with 7 rows and 8 columns having (96.0, 0.0) in location Y(0,0) and (0.0, 0.0) in all others.

Example 3: This example shows the same array being used for input and output, where *isign* = −1 and *scale* = 1/((N1)(N2)). The arrays are declared as follows:

```
COMPLEX*16  Y(0:8,0:7)
REAL*8      X(0:19,0:7)
REAL*8      AUX1(1000), AUX2(1000), AUX3(1)
```

Arrays X and Y are made equivalent by the following statement, making them occupy the same storage.

```
EQUIVALENCE (X,Y)
```

This requires *inc2x* ≥ 2(*inc2y*). First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Call Statement and Input

```
            INIT  X  INC2X Y INC2Y N1   N2 ISIGN SCALE  AUX1   NAUX1  AUX2   NAUX2  AUX3  NAUX3
             |    |    |   |   |   |    |   |     |      |       |     |       |     |      |
CALL DRCFT2(INIT, X , 20 , Y , 9 , 16 , 8 , -1 , SCALE, AUX1 , 1000 , AUX2 , 1000 , AUX3 ,  0 )
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
SCALE    =  1.0/128.0
```

```
        ⎡                                                                ⎤
        ⎢  2.0   2.0  -2.0  -2.0   2.0   2.0  -2.0  -2.0 ⎥
        ⎢  2.0  -2.0  -2.0   2.0   2.0  -2.0  -2.0   2.0 ⎥
        ⎢ -2.0  -2.0   2.0   2.0  -2.0  -2.0   2.0   2.0 ⎥
        ⎢ -2.0   2.0   2.0  -2.0  -2.0   2.0   2.0  -2.0 ⎥
        ⎢  2.0   2.0  -2.0  -2.0   2.0   2.0  -2.0  -2.0 ⎥
        ⎢  2.0  -2.0  -2.0   2.0   2.0  -2.0  -2.0   2.0 ⎥
        ⎢ -2.0  -2.0   2.0   2.0  -2.0  -2.0   2.0   2.0 ⎥
        ⎢ -2.0   2.0   2.0  -2.0  -2.0   2.0   2.0  -2.0 ⎥
        ⎢  2.0   2.0  -2.0  -2.0   2.0   2.0  -2.0  -2.0 ⎥
X    =  ⎢  2.0  -2.0  -2.0   2.0   2.0  -2.0  -2.0   2.0 ⎥
        ⎢ -2.0  -2.0   2.0   2.0  -2.0  -2.0   2.0   2.0 ⎥
        ⎢ -2.0   2.0   2.0  -2.0  -2.0   2.0   2.0  -2.0 ⎥
        ⎢  2.0   2.0  -2.0  -2.0   2.0   2.0  -2.0  -2.0 ⎥
        ⎢  2.0  -2.0  -2.0   2.0   2.0  -2.0  -2.0   2.0 ⎥
        ⎢ -2.0  -2.0   2.0   2.0  -2.0  -2.0   2.0   2.0 ⎥
        ⎢ -2.0   2.0   2.0  -2.0  -2.0   2.0   2.0  -2.0 ⎥
        ⎢   .     .     .     .     .     .     .     .  ⎥
        ⎢   .     .     .     .     .     .     .     .  ⎥
        ⎢   .     .     .     .     .     .     .     .  ⎥
        ⎢   .     .     .     .     .     .     .     .  ⎥
        ⎣                                                                ⎦
```

Output: `Y` is an array with 9 rows and 8 columns having (1.0, 1.0) in location `Y(4,2)` and (0.0, 0.0) in all others.

SCRFT2 and DCRFT2—Complex-to-Real Fourier Transform in Two Dimensions

These subroutines compute the two-dimensional discrete Fourier transform of complex conjugate even data in a two-dimensional array.

Table 133. Data Types

X	Y, *scale*	**Subroutine**
Short-precision complex	Short-precision real	SCRFT2
Long-precision complex	Long-precision real	DCRFT2

Note: Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.

Syntax

Fortran	CALL SCRFT2 (*init*, *x*, *inc2x*, *y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*, *aux3*, *naux3*) CALL DCRFT2 (*init*, *x*, *inc2x*, *y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	scrft2 (*init*, *x*, *inc2x*, *y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*, *aux3*, *naux3*); dcrft2 (*init*, *x*, *inc2x*, *y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);
PL/I	CALL SCRFT2 (*init*, *x*, *inc2x*, *y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*, *aux3*, *naux3*); CALL DCRFT2 (*init*, *x*, *inc2x*, *y*, *inc2y*, *n1*, *n2*, *isign*, *scale*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init
is a flag, where:

If *init* ≠ 0, trigonometric functions and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*. The contents of *x* and *y* are not used or changed.

If *init* = 0, the discrete Fourier transform of the given array is computed. The only arguments that may change after initialization are *x*, *y*, and *aux2*. All scalar arguments must be the same as when the subroutine was called for initialization with *init* ≠ 0.

Specified as: a fullword integer. It can have any value.

x
is the array X, containing *n2* columns of data to be transformed. Due to complex conjugate symmetry, the input consists of only the first ((*n1*)/2)+1 rows of the array—that is, $x_{j1,j2}$, *j1* = 0, 1, ..., (*n1*)/2, *j2* = 0, 1, ..., *n2*−1. The data in each column is stored with stride 1.

Specified as: an *inc2x* by (at least) *n2* array, containing numbers of the data type indicated in Table 133. This array must be aligned on a doubleword boundary.

inc2x
is the leading dimension (stride between columns) of array X. Specified as: a fullword integer; *inc2x* ≥ ((*n1*)/2)+1.

y
See On Return.

inc2y
is the leading dimension (stride between the columns) of array Y. Specified as: a fullword integer; *inc2y* ≥ *n1*+2.

n1
is the number of rows of data—that is, the length of the columns in array Y involved in the computation. The length of the columns in array X are (*n1*)/2+1. Specified as: a fullword integer; *n1* ≤ 37748736 and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 741. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 38.

n2
is the number of columns of data—that is, the length of the rows in arrays X and Y involved in the computation. Specified as: a fullword integer; *n2* ≤ 37748736 and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 741. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 38.

isign
controls the direction of the transform, determining the sign *Isign* of the exponents of W_{n1} and W_{n2}, where:

If *isign* = positive value, *Isign* = + (transforming time to frequency).

If *isign* = negative value, *Isign* = − (transforming frequency to time).

Specified as: a fullword integer; *isign* > 0 or *isign* < 0.

scale
is the scaling constant *scale*. See "Function" on page 807 for its usage. Specified as: a number of the data type indicated in Table 133 on page 804, where *scale* > 0.0 or *scale* < 0.0.

aux1
is the working storage for this subroutine, where:

If *init* ≠ 0, the working storage is computed.

If *init* = 0, the working storage is used in the computation of the Fourier transforms.

Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1
is the number of doublewords in the working storage specified in *aux*. Specified as: a fullword integer; *naux1* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 31.

aux2
has the following meaning:

If *naux2* = 0 and error 2015 is unrecoverable, *aux2* is ignored.

Otherwise, it is the working storage used by this subroutine, which is available for use by the calling program between calls to this subroutine.

Specified as: an area of storage, containing *naux2* long-precision real numbers. On output, the contents are overwritten.

naux2
is the number of doublewords in the working storage specified in *aux2*. Specified as: a fullword integer, where:

If *naux2* = 0 and error 2015 is unrecoverable, SCRFT2 and DCRFT2 dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux2* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 31.

aux3
this argument is provided for migration purposes only and is ignored.

Specified as: an area of storage, containing *naux3* long-precision real numbers.

naux3
this argument is provided for migration purposes only and is ignored.

Specified as: a fullword integer.

On Return

y
has the following meaning, where:

If *init* ≠ 0, this argument is not used, and its contents remain unchanged.

If *init* = 0, this is the array Y, containing *n1* rows and *n2* columns of results of the real discrete Fourier transform of X. The data in each column of Y is stored with stride 1.

Returned as: an *inc2y* by (at least) *n2* array, containing numbers of the data type indicated in Table 133 on page 804. See "Notes" for more details.

aux1
is the working storage for this subroutine, where:

If *init* ≠ 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If *init* = 0, its contents are unchanged.

Returned as: the contents are not relevant.

Notes

1. *aux1* should **not** be used by the calling program between program calls to this subroutine with *init* ≠ 0 and *init* = 0. However, it can be reused after intervening calls to this subroutine with different arguments.
2. If you specify the same array for X and Y, then (2)(*inc2x*) must equal *inc2y*. In this case, output overwrites input. If you specify different arrays X and Y, they must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.
3. For selecting optimal strides (or leading dimensions *inc2x* and *inc2y*) for your input and output arrays, you should use "STRIDE—Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines" on page 977. Example 6 in the STRIDE subroutine description explains how it is used for these subroutines.

4. Be sure to align array Y on a doubleword boundary, and specify an even number for *inc2y*, if possible.

Processor-Independent Formulas for SCRFT2 for NAUX1 and NAUX2: The required values of *naux1* and *naux2* depend on *n1* and *n2*.

NAUX1 Formulas

If max(*n1*/2, *n2*) ≤ 8192, use *naux1* = 45000.
If max(*n1*/2, *n2*) > 8192, use *naux1* = 40000+0.82*n1*+1.14*n2*.

NAUX2 Formulas

If *n1* ≤ 16384 and *n2* < 252, use *naux2* = 20000.
If *n1* > 16384 and *n2* < 252, use *naux2* = 20000+0.57*n1*.
If *n2* ≥ 252, add the following to the above storage requirements:
(*n2*+256)(1.14+*s*)
where *s* = min(64, 1+*n1*/2).

Processor-Independent Formulas for DCRFT2 for NAUX1 and NAUX2: The required values of *naux1* and *naux2* depend on *n1* and *n2*.

NAUX1 Formulas

If *n* ≤ 2048, use *naux1* = 42000.
If *n* > 2048, use *naux1* = 40000+1.64*n1*+2.28*n2*,
where *n* = max(*n1*/2, *n2*).

NAUX2 Formulas

If *n1* ≤ 4096 and *n2* < 252, use *naux2* = 20000.
If *n1* > 4096 and *n2* < 252, use *naux2* = 20000+1.14*n1*.
If *n2* ≥ 252, add the following to the above storage requirements:
((2)*n2*+256) (2.28+*s*)
where *s* = min(64, 1+*n1*/2).

Function: The two-dimensional discrete Fourier transform of complex conjugate even data in array X, with results going into array Y, is expressed as follows:

$$y_{k1,k2} = scale \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} x_{j1,j2} \; W_{n1}^{(Isign)j1k1} \; W_{n2}^{(Isign)j2k2}$$

for:

k1 = 0, 1, ..., *n1*−1
k2 = 0, 1, ..., *n2*−1

where:

$$W_{n1} = e^{-2\pi\left(\sqrt{-1}\right)/n1}$$
$$W_{n2} = e^{-2\pi\left(\sqrt{-1}\right)/n2}$$

and where:

$x_{j1,j2}$ are elements of array X.
$y_{k1,k2}$ are elements of array Y.
Isign is + or − (determined by argument *isign*).
scale is a scalar value.

Because of the complex conjugate symmetry, the output in array Y is real. For *scale* = 1.0 and *isign* being positive, you obtain the discrete Fourier transform, a function of frequency. The inverse Fourier transform is obtained with *scale* = 1.0/((*n1*)(*n2*)) and *isign* being negative. See references [1], [4], and [20].

Two invocations of this subroutine are necessary:

1. With *init* ≠ 0, the subroutine tests and initializes arguments of the program, setting up the *aux1* working storage.
2. With *init* = 0, the subroutine checks that the initialization arguments in the *aux1* working storage correspond to the present arguments, and if so, performs the calculation of the Fourier transform.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux2* = 0, and unable to allocate work area.

Computational Errors: None

Input-Argument Errors

1. *n1* > 37748736
2. *n2* > 37748736
3. *inc2x* < (*n1*)/2+1
4. *inc2y* < *n1*+2
5. *scale* = 0.0
6. *isign* = 0
7. The subroutine has not been initialized with the present arguments.
8. The length of one of the transforms in *n1* or *n2* is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
9. *naux1* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
10. Error 2015 is recoverable or *naux2*≠0, and *naux2* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example 1: This example shows how to compute a two-dimensional transform. The arrays are declared as follows:

```
REAL*4     Y(0:13,0:7)
COMPLEX*8  X(0:6,0:7)
REAL*8     AUX1(1000), AUX2(1000), AUX3(1)
```

First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Call Statement and Input

```
            INIT  X INC2X Y  INC2Y N1   N2 ISIGN SCALE   AUX1  NAUX1   AUX2  NAUX2   AUX3  NAUX3
             |    |   |   |    |   |    |    |     |      |      |      |      |      |      |
CALL SCRFT2(INIT, X , 7 , Y , 14 , 12 , 8 , -1 , SCALE , AUX1 , 1000 , AUX2 , 1000 , AUX3 ,  0 )
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
SCALE    =  1.0/96.0
```

X is an array with 7 rows and 8 columns with (1.0, 0.0) in all locations.

Output

```
         ┌                                                   ┐
         │ 1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 │
         │ 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 │
         │ 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 │
         │ 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 │
         │ 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 │
         │ 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 │
Y    =   │ 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 │
         │ 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 │
         │ 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 │
         │ 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 │
         │ 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 │
         │ 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0 │
         │  .    .    .    .    .    .    .    .  │
         │  .    .    .    .    .    .    .    .  │
         └                                                   ┘
```

Example 2: This example shows another transform computation with different data using the same initialized array AUX1 in Example 1.

Call Statement and Input

```
            INIT X INC2X Y  INC2Y N1   N2 ISIGN SCALE   AUX1   NAUX1  AUX2   NAUX2  AUX3  NAUX3
             |   |   |   |    |   |    |    |     |      |      |     |      |     |      |
CALL SCRFT2( 0 , X , 7 , Y , 14 , 12 , 8 , -1 , SCALE , AUX1 , 1000 , AUX2 , 1000 , AUX3 ,  0 )
```

```
SCALE    =  1.0/96.0
```

X is an array with 7 rows and 8 columns having (96.0, 0.0) in location X(0,0) and (0.0, 0.0) in all others.

Output

```
                ┌                                       ┐
                │ 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 │
                │ 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 │
                │ 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 │
                │ 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 │
                │ 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 │
                │ 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 │
Y    =          │ 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 │
                │ 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 │
                │ 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 │
                │ 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 │
                │ 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 │
                │ 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 │
                │  .    .    .    .    .    .    .    .  │
                │  .    .    .    .    .    .    .    .  │
                └                                       ┘
```

Example 3: This example shows the same array being used for input and output. The arrays are declared as follows:

```
REAL*8      Y(0:17,0:7)
COMPLEX*16  X(0:8,0:7)
REAL*8      AUX1(1000), AUX2(1000), AUX3(1)
```

Arrays X and Y are made equivalent by the following statement, making them occupy the same storage.

```
EQUIVALENCE (X,Y)
```

This requires *inc2y* = 2(*inc2x*). First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Call Statement and Input

```
            INIT  X INC2X Y  INC2Y N1  N2 ISIGN SCALE   AUX1   NAUX1  AUX2   NAUX2  AUX3  NAUX3
             |    |   |   |    |    |   |   |     |      |       |      |      |     |      |
CALL DCRFT2(INIT, X , 9 , Y , 18 , 16 , 8 , 1 , SCALE , AUX1 , 1000 , AUX2 , 1000 , AUX3 ,  0 )
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
SCALE    =  1.0
```

X is an array with 9 rows and 8 columns having (1.0, 1.0) in location X(4,2) and (0.0, 0.0) in all others.

Output

$$
Y = \begin{bmatrix}
2.0 & 2.0 & -2.0 & -2.0 & 2.0 & 2.0 & -2.0 & -2.0 \\
2.0 & -2.0 & -2.0 & 2.0 & 2.0 & -2.0 & -2.0 & 2.0 \\
-2.0 & -2.0 & 2.0 & 2.0 & -2.0 & -2.0 & 2.0 & 2.0 \\
-2.0 & 2.0 & 2.0 & -2.0 & -2.0 & 2.0 & 2.0 & -2.0 \\
2.0 & 2.0 & -2.0 & -2.0 & 2.0 & 2.0 & -2.0 & -2.0 \\
2.0 & -2.0 & -2.0 & 2.0 & 2.0 & -2.0 & -2.0 & 2.0 \\
-2.0 & -2.0 & 2.0 & 2.0 & -2.0 & -2.0 & 2.0 & 2.0 \\
-2.0 & 2.0 & 2.0 & -2.0 & -2.0 & 2.0 & 2.0 & -2.0 \\
2.0 & 2.0 & -2.0 & -2.0 & 2.0 & 2.0 & -2.0 & -2.0 \\
2.0 & -2.0 & -2.0 & 2.0 & 2.0 & -2.0 & -2.0 & 2.0 \\
-2.0 & -2.0 & 2.0 & 2.0 & -2.0 & -2.0 & 2.0 & 2.0 \\
-2.0 & 2.0 & 2.0 & -2.0 & -2.0 & 2.0 & 2.0 & -2.0 \\
2.0 & 2.0 & -2.0 & -2.0 & 2.0 & 2.0 & -2.0 & -2.0 \\
2.0 & -2.0 & -2.0 & 2.0 & 2.0 & -2.0 & -2.0 & 2.0 \\
-2.0 & -2.0 & 2.0 & 2.0 & -2.0 & -2.0 & 2.0 & 2.0 \\
-2.0 & 2.0 & 2.0 & -2.0 & -2.0 & 2.0 & 2.0 & -2.0 \\
. & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & .
\end{bmatrix}
$$

SCFT3 and DCFT3—Complex Fourier Transform in Three Dimensions

These subroutines compute the three-dimensional discrete Fourier transform of complex data.

Table 134. Data Types

X, Y	*scale*	**Subroutine**
Short-precision complex	Short-precision real	SCFT3
Long-precision complex	Long-precision real	DCFT3

Note: For each use, only one invocation of this subroutine is necessary. The initialization phase, preparing the working storage, is a relatively small part of the total computation, so it is performed on each invocation.

Syntax

Fortran	CALL SCFT3 | DCFT3 (*x*, *inc2x*, *inc3x*, *y*, *inc2y*, *inc3y*, *n1*, *n2*, *n3*, *isign*, *scale*, *aux*, *naux*)
C and C++	scft3 | dcft3 (*x*, *inc2x*, *inc3x*, *y*, *inc2y*, *inc3y*, *n1*, *n2*, *n3*, *isign*, *scale*, *aux*, *naux*);
PL/I	CALL SCFT3 | DCFT3 (*x*, *inc2x*, *inc3x*, *y*, *inc2y*, *inc3y*, *n1*, *n2*, *n3*, *isign*, *scale*, *aux*, *naux*);

On Entry

x
is the array X, containing the three-dimensional data to be transformed, where each element $x_{j1,j2,j3}$, using zero-based indexing, is stored in X(*j1*+*j2*(*inc2x*)+*j3*(*inc3x*)) for *j1* = 0, 1, ..., *n1*−1, *j2* = 0, 1, ..., *n2*−1, and *j3* = 0, 1, ..., *n3*−1. The strides for the elements in the first, second, and third dimensions are assumed to be 1, *inc2x*(≥ *n1*), and *inc3x*(≥ (*n2*)(*inc2x*)), respectively.

Specified as: an array, containing numbers of the data type indicated in Table 134. This array must be aligned on a doubleword boundary. If the array is dimensioned X(LDA1,LDA2,LDA3), then LDA1 = *inc2x*, (LDA1)(LDA2) = *inc3x*, and LDA3 ≥ *n3*. For information on how to set up this array, see "Setting Up Your Data" on page 744. For more details, see "Notes" on page 814.

inc2x
is the stride between the elements in array X for the second dimension. Specified as: a fullword integer; *inc2x* ≥ *n1*.

inc3x
is the stride between the elements in array X for the third dimension. Specified as: a fullword integer; *inc3x* ≥ (*n2*)(*inc2x*).

y
See On Return.

inc2y
is the stride between the elements in array Y for the second dimension. Specified as: a fullword integer; *inc2y* ≥ *n1*.

inc3y
is the stride between the elements in array Y for the third dimension. Specified as: a fullword integer; *inc3y* ≥ (*n2*)(*inc2y*).

n1
is the length of the first dimension of the three-dimensional data in the array to be transformed. Specified as: a fullword integer; $n1 \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 741. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 38.

n2
is the length of the second dimension of the three-dimensional data in the array to be transformed. Specified as: a fullword integer; $n2 \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 741. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 38.

n3
is the length of the third dimension of the three-dimensional data in the array to be transformed. Specified as: a fullword integer; $n3 \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 741. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 38.

isign
controls the direction of the transform, determining the sign *Isign* of the exponents of W_{n1}, W_{n2}, and W_{n3}, where:

If $isign = $ positive value, $Isign = +$ (transforming time to frequency).

If $isign = $ negative value, $Isign = -$ (transforming frequency to time).

Specified as: a fullword integer; $isign > 0$ or $isign < 0$.

scale
is the scaling constant *scale*. See "Function" on page 815 for its usage. Specified as: a number of the data type indicated in Table 134 on page 812, where $scale > 0.0$ or $scale < 0.0$.

aux
has the following meaning:

If $naux = 0$ and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is a storage work area used by this subroutine.

Specified as: an area of storage, containing *naux* long-precision real numbers. On output, the contents are overwritten.

naux
is the number of doublewords in the working storage specified in *aux*. Specified as: a fullword integer, where:

If $naux = 0$ and error 2015 is unrecoverable, SCFT3 and DCFT3 dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, $naux \geq$ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 31.

On Return

y

is the array Y, containing the elements resulting from the three-dimensional discrete Fourier transform of the data in X. Each element $y_{k1,k2,k3}$, using zero-based indexing, is stored in Y(*k1*+*k2*(*inc2y*)+*k3*(*inc3y*)) for *k1* = 0, 1, ..., *n1*−1, *k2* = 0, 1, ..., *n2*−1, and *k3* = 0, 1, ..., *n3*−1. The strides for the elements in the first, second, and third dimensions are assumed to be 1, *inc2y*(≥ *n1*), and *inc3y*(≥ (*n2*)(*inc2y*)), respectively.

Returned as: an array, containing numbers of the data type indicated in Table 134 on page 812. This array must be aligned on a doubleword boundary. If the array is dimensioned Y(LDA1,LDA2,LDA3), then LDA1 = *inc2y*, (LDA1)(LDA2) = *inc3y*, and LDA3 ≥ *n3*. For information on how to set up this array, see "Setting Up Your Data" on page 744. For more details, see "Notes."

Notes

1. If you specify the same array for X and Y, then *inc2x* must be greater than or equal to *inc2y*, and *inc3x* must be greater than or equal to *inc3y*. In this case, output overwrites input. When using the ESSL SMP library in a multithreaded environment, if *inc2x* > *inc2y* or *inc3x* > *inc3y*, these subroutines run on a single thread and issue an attention message.

   If you specify different arrays X and Y, they must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.

2. You should use "STRIDE—Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines" on page 977 to determine the optimal values for the strides *inc2y* and *inc3y* for your output array. The strides for your input array do not affect performance. Example 7 in the STRIDE subroutine description explains how it is used for these subroutines. For additional information on how to set up your data, see "Setting Up Your Data" on page 744.

Processor-Independent Formulas for SCFT3 for NAUX: Use the following formulas for calculating *naux*:

1. If max(*n2*, *n3*) < 252 and:

   If *n1* ≤ 8192, use *naux* = 60000.
   If *n1* > 8192, use *naux* = 60000+2.28*n1*.

2. If *n2* ≥ 252, *n3* < 252, and:

   If *n1* ≤ 8192, use *naux* = 60000+λ.
   If *n1* > 8192, use *naux* = 60000+2.28*n1*+λ,
   where λ = (*n2*+256)(*s*+2.28)
   and *s* = min(64, *n1*).

3. If *n2* < 252, *n3* ≥ 252, and:

   If *n1* ≤ 8192, use *naux* = 60000+ψ.
   If *n1* > 8192, use *naux* = 60000+2.28*n1*+ψ,
   where ψ = (*n3*+256)(*s*+2.28)
   and *s* = min(64, (*n1*)(*n2*)).

4. If *n2* ≥ 252 and *n3* ≥ 252, use the larger of the values calculated for cases 2 and 3 above.

Processor-Independent Formulas for DCFT3 for NAUX: Use the following formulas for calculating *naux*:

1. If max(*n2*, *n3*) < 252 and:

   If *n1* ≤ 2048, use *naux* = 60000.
   If *n1* > 2048, use *naux* = 60000+4.56*n1*.

2. If *n2* ≥ 252, *n3* < 252, and:

   If *n1* ≤ 2048, use *naux* = 60000+λ.
   If *n1* > 2048, use *naux* = 60000+4.56*n1*+λ,
   where λ = ((2)*n2*+256)(*s*+4.56)
   and *s* = min(64, *n1*).

3. If *n2* < 252, *n3* ≥ 252, and:

   If *n1* ≤ 2048, use *naux* = 60000+ψ.
   If *n1* > 2048, use *naux* = 60000+4.56*n1*+ψ,
   where ψ = ((2)*n3*+256)(*s*+4.56)
   and *s* = min(64, (*n1*)(*n2*)).

4. If *n2* ≥ 252 and *n3* ≥ 252, use the larger of the values calculated for cases 2 and 3 above.

Function: The three-dimensional discrete Fourier transform of complex data in array X, with results going into array Y, is expressed as follows:

$$y_{k1,k2,k3} = scale \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} \sum_{j3=0}^{n3-1} x_{j1,j2,j3}\ W_{n1}^{(Isign)j1k1}\ W_{n2}^{(Isign)j2k2}\ W_{n3}^{(Isign)j3k3}$$

for:

k1 = 0, 1, ..., *n1*−1
k2 = 0, 1, ..., *n2*−1
k3 = 0, 1, ..., *n3*−1

where:

$$W_{n1} = e^{-2\pi\left(\sqrt{-1}\right)/n1}$$
$$W_{n2} = e^{-2\pi\left(\sqrt{-1}\right)/n2}$$
$$W_{n3} = e^{-2\pi\left(\sqrt{-1}\right)/n3}$$

and where:

$x_{j1,j2,j3}$ are elements of array X.
$y_{k1,k2,k3}$ are elements of array Y.
Isign is + or − (determined by argument *isign*).
scale is a scalar value.

For *scale* = 1.0 and *isign* being positive, you obtain the discrete Fourier transform, a function of frequency. The inverse Fourier transform is obtained with

scale = 1.0/((*n1*)(*n2*)(*n3*)) and *isign* being negative. See references [1], [4], [5], [19], and [20].

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors: None

Input-Argument Errors

1. *n1* > 37748736
2. *n2* > 37748736
3. *n3* > 37748736
4. *inc2x* < *n1*
5. *inc3x* < (*n2*)(*inc2x*)
6. *inc2y* < *n1*
7. *inc3y* < (*n2*)(*inc2y*)
8. *scale* = 0.0
9. *isign* = 0
10. The length of one of the transforms in *n1*, *n2*, or *n3* is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
11. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example: This example shows how to compute a three-dimensional transform. In this example, INC2X ≥ INC2Y and INC3X ≥ INC3Y, so that the same array can be used for both input and output. The STRIDE subroutine is called to select good values for the INC2Y and INC3Y strides. (As explained below, STRIDE is not called for INC2X and INC3X.) Using the transform lengths (N1 = 32, N2 = 64, and N3 = 40) along with the output data type (short-precision complex: 'C'), STRIDE is called once for each stride needed. First, it is called for INC2Y:

```
CALL STRIDE (N2,N1,INC2Y,'C',0)
```

The output value returned for INC2Y is 32. Then STRIDE is called again for INC3Y:

```
CALL STRIDE (N3,N2*INC2Y,INC3Y,'C',0)
```

The output value returned for INC3Y is 2056. Because INC3Y is not a multiple of INC2Y, Y is not declared as a three-dimensional array. It is declared as a two-dimensional array, Y(INC3Y,N3).

To equivalence the X and Y arrays requires INC2X ≥ INC2Y and INC3X ≥ INC3Y. Therefore, INC2X is set equal to INC2Y(= 32). Also, to declare the X array as a three-dimensional array, INC3X must be a multiple of INC2X. Therefore, its value is set as INC3X = (65)(INC2X) = 2080.

The arrays are declared as follows:

```
COMPLEX*8  X(32,65,40),Y(2056,40)
REAL*8     AUX(30000)
```

Arrays X and Y are made equivalent by the following statement, making them occupy the same storage:

```
EQUIVALENCE (X,Y)
```

Call Statement and Input

```
            X  INC2X INC3X  Y  INC2Y INC3Y  N1   N2   N3 ISIGN SCALE   AUX   NAUX
            |    |     |    |    |     |    |    |    |   |     |      |      |
CALL SCFT3( X , 32 , 2080 , Y , 32 , 2056 , 32 , 64 , 40 , 1 , SCALE , AUX , 30000)
```

```
SCALE    =  1.0
```

X has (1.0,2.0) in location `X(1,1,1)` and (0.0,0.0) in all other locations.

Output: `Y` has (1.0,2.0) in locations `Y`(*ij*,*k*), where *ij* = 1, 2048 and *j* = 1, 40. It remains unchanged elsewhere.

SRCFT3 and DRCFT3—Real-to-Complex Fourier Transform in Three Dimensions

These subroutines compute the three-dimensional discrete Fourier transform of real data in a three-dimensional array.

Table 135. Data Types

X, *scale*	Y	**Subroutine**
Short-precision real	Short-precision complex	SRCFT3
Long-precision real	Long-precision complex	DRCFT3

Note: For each use, only one invocation of this subroutine is necessary. The initialization phase, preparing the working storage, is a relatively small part of the total computation, so it is performed on each invocation.

Syntax

Fortran	CALL SRCFT3 | DRCFT3 (*x*, *inc2x*, *inc3x*, *y*, *inc2y*, *inc3y*, *n1*, *n2*, *n3*, *isign*, *scale*, *aux*, *naux*)
C and C++	srcft3 | drcft3 (*x*, *inc2x*, *inc3x*, *y*, *inc2y*, *inc3y*, *n1*, *n2*, *n3*, *isign*, *scale*, *aux*, *naux*);
PL/I	CALL SRCFT3 | DRCFT3 (*x*, *inc2x*, *inc3x*, *y*, *inc2y*, *inc3y*, *n1*, *n2*, *n3*, *isign*, *scale*, *aux*, *naux*);

On Entry

x

is the array X, containing the three-dimensional data to be transformed, where each element $x_{j1,j2,j3}$, using zero-based indexing, is stored in X(*j1*+*j2*(*inc2x*)+*j3*(*inc3x*)) for *j1* = 0, 1, ..., *n1*−1, *j2* = 0, 1, ..., *n2*−1, and *j3* = 0, 1, ..., *n3*−1. The strides for the elements in the first, second, and third dimensions are assumed to be 1, *inc2x*(≥ *n1*), and *inc3x*(≥ (*n2*)(*inc2x*)), respectively.

Specified as: an array, containing numbers of the data type indicated in Table 135. If the array is dimensioned X(LDA1,LDA2,LDA3), then LDA1 = *inc2x*, (LDA1)(LDA2) = *inc3x*, and LDA3 ≥ *n3*. For information on how to set up this array, see "Setting Up Your Data" on page 744. For more details, see "Notes" on page 820.

inc2x

is the stride between the elements in array X for the second dimension. Specified as: a fullword integer; *inc2x* ≥ *n1*.

inc3x

is the stride between the elements in array X for the third dimension. Specified as: a fullword integer; *inc3x* ≥ (*n2*)(*inc2x*).

y

See On Return.

inc2y

is the stride between the elements in array Y for the second dimension. Specified as: a fullword integer; *inc2y* ≥ *n1*/2+1.

inc3y

is the stride between the elements in array Y for the third dimension. Specified as: a fullword integer; *inc3y* ≥ (*n2*)(*inc2y*).

n1
is the length of the first dimension of the three-dimensional data in the array to be transformed. Specified as: a fullword integer; $n1 \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 741. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 38.

n2
is the length of the second dimension of the three-dimensional data in the array to be transformed. Specified as: a fullword integer; $n2 \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 741. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 38.

n3
is the length of the third dimension of the three-dimensional data in the array to be transformed. Specified as: a fullword integer; $n3 \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 741. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 38.

isign
controls the direction of the transform, determining the sign *Isign* of the exponents of W_{n1}, W_{n2}, and W_{n3}, where:

If $isign = $ positive value, $Isign = +$ (transforming time to frequency).

If $isign = $ negative value, $Isign = -$ (transforming frequency to time).

Specified as: a fullword integer; $isign > 0$ or $isign < 0$.

scale
is the scaling constant *scale*. See "Function" on page 821 for its usage. Specified as: a number of the data type indicated in Table 135 on page 818, where $scale > 0.0$ or $scale < 0.0$.

aux
has the following meaning:

If $naux = 0$ and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is a storage work area used by this subroutine.

Specified as: an area of storage, containing *naux* long-precision real numbers. On output, the contents are overwritten.

naux
is the number of doublewords in the working storage specified in *aux*. Specified as: a fullword integer, where:

If $naux = 0$ and error 2015 is unrecoverable, SRCFT3 and DRCFT3 dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, $naux \geq$ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 31.

On Return

y

is the array Y, containing the elements resulting from the three-dimensional discrete Fourier transform of the data in X. Each element $y_{k1,k2,k3}$, using zero-based indexing, is stored in Y(*k1*+*k2*(*inc2y*)+*k3*(*inc3y*)) for *k1* = 0, 1, ..., *n1*/2, *k2* = 0, 1, ..., *n2*−1, and *k3* = 0, 1, ..., *n3*−1. Due to complex conjugate symmetry, the output consists of only the first *n1*/2+1 values along the first dimension of the array, for *k1* = 0, 1, ..., *n1*/2. The strides for the elements in the first, second, and third dimensions are assumed to be 1, *inc2y*(≥ *n1*/2+1), and *inc3y*(≥ (*n2*)(*inc2y*)), respectively.

Returned as: an array, containing numbers of the data type indicated in Table 135 on page 818. This array must be aligned on a doubleword boundary. If the array is dimensioned Y(LDA1,LDA2,LDA3), then LDA1 = *inc2y*, (LDA1)(LDA2) = *inc3y*, and LDA3 ≥ *n3*. For information on how to set up this array, see "Setting Up Your Data" on page 744. For more details, see "Notes."

Notes

1. If you specify the same array for X and Y, then *inc2x* must be greater than or equal to (2)(*inc2y*), and *inc3x* must be greater than or equal to (2)(*inc3y*). In this case, output overwrites input. When using the ESSL SMP library in a multithreaded environment, if *inc2x* > (2)(*inc2y*) or *inc3x* > (2)(*inc3y*), these subroutines run on a single thread and issue an attention message.

   If you specify different arrays X and Y, they must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.

2. To achieve the best performance, you should align array X on a doubleword boundary, and *inc2x* and *inc3x* should be even numbers. The strides for your input array do not affect performance as long as they are even numbers. In addition, you should use "STRIDE—Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines" on page 977 to determine the optimal values for the strides *inc2y* and *inc3y* for your output array. Example 8 in the STRIDE subroutine description explains how it is used for these subroutines. For additional information on how to set up your data, see "Setting Up Your Data" on page 744.

Processor-Independent Formulas for SRCFT3 for NAUX: Use the following formulas for calculating *naux*:

1. If max(*n2*, *n3*) < 252 and:

   If *n1* ≤ 16384, use *naux* = 65000.
   If *n1* > 16384, use *naux* = 60000+1.39*n1*.

2. If *n2* ≥ 252, *n3* < 252, and:

   If *n1* ≤ 16384, use *naux* = 65000+λ.
   If *n1* > 16384, use *naux* = 60000+1.39*n1*+λ,
   where λ = (*n2*+256)(*s*+2.28) and *s* = min(64, 1+*n1*/2).

3. If *n2* < 252, *n3* ≥ 252, and:

   If *n1* ≤ 16384, use *naux* = 65000+ψ.
   If *n1* > 16384, use *naux* = 60000+1.39*n1*+ψ,
   where ψ = (*n3*+256)(*s*+2.28) and *s* = min(64, (*n2*)(1+*n1*/2)).

4. If *n2* ≥ 252 and *n3* ≥ 252, use the larger of the values calculated for cases 2 and 3 above.

If *inc2x* or *inc3x* is an odd number, or if array X is not aligned on a doubleword boundary, you should add the following amount to all the formulas given above:

n2(1+*n1*/2)

Processor-Independent Formulas for DRCFT3 for NAUX: Use the following formulas for calculating *naux*:

1. If max(*n2*, *n3*) < 252 and:

   If *n1* ≤ 4096, use *naux* = 62000.
   If *n1* > 4096, use *naux* = 60000+2.78*n1*.

2. If *n2* ≥ 252, *n3* < 252, and:

   If *n1* ≤ 4096, use *naux* = 62000+λ.
   If *n1* > 4096, use *naux* = 60000+2.78*n1*+λ,
   where λ = ((2)*n2*+256)(*s*+4.56)
   and *s* = min(64, *n1*/2).

3. If *n2* < 252, *n3* ≥ 252, and:

   If *n1* ≤ 4096, use *naux* = 62000+ψ.
   If *n1* > 4096, use *naux* = 60000+2.78*n1*+ψ,
   where ψ = ((2)*n3*+256)(*s*+4.56)
   and *s* = min(64, *n2*(1+*n1*/2)).

4. If *n2* ≥ 252 and *n3* ≥ 252, use the larger of the values calculated for cases 2 and 3 above.

Function: The three-dimensional complex conjugate even discrete Fourier transform of real data in array X, with results going into array Y, is expressed as follows:

$$y_{k1,k2,k3} = scale \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} \sum_{j3=0}^{n3-1} x_{j1,j2,j3} \; W_{n1}^{(Isign)j1k1} \; W_{n2}^{(Isign)j2k2} \; W_{n3}^{(Isign)j3k3}$$

for:

k1 = 0, 1, ..., *n1*−1
k2 = 0, 1, ..., *n2*−1
k3 = 0, 1, ..., *n3*−1

where:

$$W_{n1} = e^{-2\pi\left(\sqrt{-1}\right)/n1}$$
$$W_{n2} = e^{-2\pi\left(\sqrt{-1}\right)/n2}$$
$$W_{n3} = e^{-2\pi\left(\sqrt{-1}\right)/n3}$$

and where:

$x_{j1,j2,j3}$ are elements of array X.
$y_{k1,k2,k3}$ are elements of array Y.
Isign is + or − (determined by argument *isign*).

scale is a scalar value.

The output in array Y is complex. For *scale* = 1.0 and *isign* being positive, you obtain the discrete Fourier transform, a function of frequency. The inverse Fourier transform is obtained with *scale* = 1.0/((*n1*)(*n2*)(*n3*)) and *isign* being negative. See references [1], [4], [5], [19], and [20].

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors: None

Input-Argument Errors

1. *n1* > 37748736
2. *n2* > 37748736
3. *n3* > 37748736
4. *inc2x* < *n1*
5. *inc3x* < (*n2*)(*inc2x*)
6. *inc2y* < *n1*/2+1
7. *inc3y* < (*n2*)(*inc2y*)
8. *scale* = 0.0
9. *isign* = 0
10. The length of one of the transforms in *n1*, *n2*, or *n3* is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
11. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example: This example shows how to compute a three-dimensional transform. In this example, INC2X ≥ (2)(INC2Y) and INC3X ≥ (2)(INC3Y), so that the same array can be used for both input and output. The STRIDE subroutine is called to select good values for the INC2Y and INC3Y strides. Using the transform lengths (N1 = 32, N2 = 64, and N2 = 40) along with the output data type (short-precision complex: 'C'), STRIDE is called once for each stride needed. First, it is called for INC2Y:

```
CALL STRIDE (N2,N1/2+1,INC2Y,'C',0)
```

The output value returned for INC2Y is 17. (This value is equal to N1/2+1.) Then STRIDE is called again for INC3Y:

```
CALL STRIDE (N3,N2*INC2Y,INC3Y,'C',0)
```

The output value returned for INC3Y is 1088. Because INC3Y is a multiple of INC2Y—that is, INC3Y = (N2)(INC2Y)—Y is declared as a three-dimensional array, Y(17,64,40). (In general, for larger arrays, these types of values for INC2Y and INC3Y are not returned by STRIDE, and you are probably not able to declare Y as a three-dimensional array.)

To equivalence the X and Y arrays requires INC2X ≥ (2)(INC2Y) and INC3X ≥ (2)(INC3Y). Therefore, the values INC2X = (2)(INC2Y) = 34 and INC3X = (2)(INC3Y) = 2176 are set, and X is declared as a three-dimensional array, X(34,64,40).

The arrays are declared as follows:

```
REAL*4     X(34,64,40)
COMPLEX*8  Y(17,64,40)
REAL*8     AUX(32000)
```

Arrays X and Y are made equivalent by the following statement, making them occupy the same storage:

```
EQUIVALENCE (X,Y)
```

Call Statement and Input

```
             X  INC2X INC3X  Y  INC2Y INC3Y  N1   N2   N3 ISIGN SCALE   AUX    NAUX
             |    |     |    |    |     |    |    |    |   |     |      |      |
CALL SRCFT3( X , 34 , 2176 , Y , 17 , 1088 , 32 , 64 , 40 , 1 , SCALE , AUX , 32000)
```

```
SCALE    =  1.0
```

X has 1.0 in location X(1,1,1) and 0.0 in all other locations.

Output: Y has (1.0,0.0) in all locations.

SCRFT3 and DCRFT3—Complex-to-Real Fourier Transform in Three Dimensions

These subroutines compute the three-dimensional discrete Fourier transform of complex conjugate even data in a three-dimensional array.

Table 136. Data Types

X	Y, *scale*	**Subroutine**
Short-precision complex	Short-precision real	SCRFT2
Long-precision complex	Long-precision real	DCRFT2

Note: For each use, only one invocation of this subroutine is necessary. The initialization phase, preparing the working storage, is a relatively small part of the total computation, so it is performed on each invocation.

Syntax

Fortran	CALL SCRFT3 | DCRFT3 (*x*, *inc2x*, *inc3x*, *y*, *inc2y*, *inc3y*, *n1*, *n2*, *n3*, *isign*, *scale*, *aux*, *naux*)
C and C++	scrft3 | dcrft3 (*x*, *inc2x*, *inc3x*, *y*, *inc2y*, *inc3y*, *n1*, *n2*, *n3*, *isign*, *scale*, *aux*, *naux*);
PL/I	CALL SCRFT3 | DCRFT3 (*x*, *inc2x*, *inc3x*, *y*, *inc2y*, *inc3y*, *n1*, *n2*, *n3*, *isign*, *scale*, *aux*, *naux*);

On Entry

x
is the array X, containing the three-dimensional data to be transformed, where each element $x_{j1,j2,j3}$, using zero-based indexing, is stored in X(*j1*+*j2*(*inc2x*)+*j3*(*inc3x*)) for *j1* = 0, 1, ..., *n1*/2, *j2* = 0, 1, ..., *n2*−1, and *j3* = 0, 1, ..., *n3*−1. Due to complex conjugate symmetry, the input consists of only the first *n1*/2+1 values along the first dimension of the array, for *j1* = 0, 1, ..., *n1*/2. The strides for the elements in the first, second, and third dimensions are assumed to be 1, *inc2x*(≥ *n1*/2+1), and *inc3x*(≥ (*n2*)(*inc2x*)), respectively.

Specified as: an array, containing numbers of the data type indicated in Table 136. This array must be aligned on a doubleword boundary. If the array is dimensioned X(LDA1,LDA2,LDA3), then LDA1 = *inc2x*, (LDA1)(LDA2) = *inc3x*, and LDA3 ≥ *n3*. For information on how to set up this array, see "Setting Up Your Data" on page 744. For more details, see "Notes" on page 826.

inc2x
is the stride between the elements in array X for the second dimension. Specified as: a fullword integer; *inc2x* ≥ *n1*/2+1.

inc3x
is the stride between the elements in array X for the third dimension. Specified as: a fullword integer; *inc3x* ≥ (*n2*)(*inc2x*).

y
See On Return.

inc2y
is the stride between the elements in array Y for the second dimension. Specified as: a fullword integer; *inc2y* ≥ *n1*+2.

inc3y
is the stride between the elements in array Y for the third dimension. Specified as: a fullword integer; *inc3y* ≥ (*n2*)(*inc2y*).

n1
is the length of the first dimension of the three-dimensional data in the array to be transformed. Specified as: a fullword integer; $n1 \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 741. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 38.

n2
is the length of the second dimension of the three-dimensional data in the array to be transformed. Specified as: a fullword integer; $n2 \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 741. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 38.

n3
is the length of the third dimension of the three-dimensional data in the array to be transformed. Specified as: a fullword integer; $n3 \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 741. For all other values specified less than 37748736, you have the option of having the next larger acceptable value returned in this argument. For details, see "Providing a Correct Transform Length to ESSL" on page 38.

isign
controls the direction of the transform, determining the sign *Isign* of the exponents of W_{n1}, W_{n2}, and W_{n3}, where:

If $isign = $ positive value, $Isign = +$ (transforming time to frequency).

If $isign = $ negative value, $Isign = -$ (transforming frequency to time).

Specified as: a fullword integer; $isign > 0$ or $isign < 0$.

scale
is the scaling constant *scale*. See "Function" on page 827 for its usage. Specified as: a number of the data type indicated in Table 136 on page 824, where $scale > 0.0$ or $scale < 0.0$.

aux
has the following meaning:

If $naux = 0$ and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is a storage work area used by this subroutine. Specified as: an area of storage, containing *naux* long-precision real numbers. On output, the contents are overwritten.

naux
is the number of doublewords in the working storage specified in *aux*. Specified as: a fullword integer, where:

If $naux = 0$ and error 2015 is unrecoverable, SCRFT3 and DCRFT3 dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, $naux \geq$ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 31.

On Return

y

is the array Y, containing the elements resulting from the three-dimensional discrete Fourier transform of the data in X. Each element $y_{k1,k2,k3}$, using zero-based indexing, is stored in Y(*k1*+*k2*(*inc2y*)+*k3*(*inc3y*)) for *k1* = 0, 1, ..., *n1*−1, *k2* = 0, 1, ..., *n2*−1, and *k3* = 0, 1, ..., *n3*−1. The strides for the elements in the first, second, and third dimensions are assumed to be 1, *inc2y*(≥ *n1*+2), and *inc3y*(≥ (*n2*)(*inc2y*)), respectively.

Returned as: an array, containing numbers of the data type indicated in Table 136 on page 824. If the array is dimensioned Y(LDA1,LDA2,LDA3), then LDA1 = *inc2y*, (LDA1)(LDA2) = *inc3y*, and LDA3 ≥ *n3*. For information on how to set up this array, see "Setting Up Your Data" on page 744. For more details, see "Notes."

Notes

1. If you specify the same array for X and Y, then *inc2y* must equal (2)(*inc2x*) and *inc3y* must equal (2)(*inc3x*). In this case, output overwrites input. If you specify different arrays X and Y, they must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.
2. To achieve the best performance, you should align array Y on a doubleword boundary, and *inc2y* and *inc3y* should be even numbers. In addition, you should use "STRIDE—Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines" on page 977 to determine the optimal values for the strides *inc2y* and *inc3y* for your output array. To obtain the best performance, you should use *inc2x* = *inc2y*/2 and *inc3x* = *inc3y*/2. Example 9 in the STRIDE subroutine description explains how it is used for these subroutines. For additional information on how to set up your data, see "Setting Up Your Data" on page 744.

Processor-Independent Formulas for SCRFT3 for Calculating NAUX: Use the following formulas for calculating *naux*:

1. If max(*n2*, *n3*) < 252 and:

   If *n1* ≤ 16384, use *naux* = 65000.
   If *n1* > 16384, use *naux* = 60000+1.39*n1*.

2. If *n2* ≥ 252, *n3* < 252, and:

   If *n1* ≤ 16384, use *naux* = 65000+λ.
   If *n1* > 16384, use *naux* = 60000+1.39*n1*+λ,
   where λ = (*n2*+256)(*s*+2.28)
   and *s* = min(64, 1+*n1*/2).

3. If *n2* < 252, *n3* ≥ 252, and:

   If *n1* ≤ 16384, use *naux* = 65000+ψ.
   If *n1* > 16384, use *naux* = 60000+1.39*n1*+ψ,
   where ψ = (*n3*+256)(*s*+2.28)
   and *s* = min(64, (*n2*)(1+*n1*/2)).

4. If *n2* ≥ 252 and *n3* ≥ 252, use the larger of the values calculated for cases 2 and 3 above.

If *inc2y* or *inc3y* is an odd number, or if array Y is not aligned on a doubleword boundary, you should add the following amount to all the formulas given above:

(1+*n1*/2)(max(*n2*, *n3*))

Processor-Independent Formulas for DCRFT3 for NAUX: Use the following formulas for calculating *naux*:

1. If max(*n2*, *n3*) < 252 and:

   If *n1* ≤ 4096, use *naux* = 62000.
   If *n1* > 4096, use *naux* = 60000+2.78*n1*.

2. If *n2* ≥ 252, *n3* < 252, and:

   If *n1* ≤ 4096, use *naux* = 62000+λ.
   If *n1* > 4096, use *naux* = 60000+2.78*n1*+λ,
   where λ = ((2)*n2*+256)(*s*+4.56)
   and *s* = min(64, *n1*/2).

3. If *n2* < 252, *n3* ≥ 252, and:

   If *n1* ≤ 4096, use *naux* = 62000+ψ.
   If *n1* > 4096, use *naux* = 60000+2.78*n1*+ψ,
   where ψ = ((2)*n3*+256)(*s*+4.56)
   and *s* = min(64, *n2*(1+*n1*/2)).

4. If *n2* ≥ 252 and *n3* ≥ 252, use the larger of the values calculated for cases 2 and 3 above.

Function: The three-dimensional discrete Fourier transform of complex conjugate even data in array X, with results going into array Y, is expressed as follows:

$$y_{k1,k2,k3} = scale \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} \sum_{j3=0}^{n3-1} x_{j1,j2,j3} \; W_{n1}^{(Isign)j1k1} \; W_{n2}^{(Isign)j2k2} \; W_{n3}^{(Isign)j3k3}$$

for:

k1 = 0, 1, ..., *n1*–1
k2 = 0, 1, ..., *n2*–1
k3 = 0, 1, ..., *n3*–1

where:

$$W_{n1} = e^{-2\pi\left(\sqrt{-1}\right)/n1}$$
$$W_{n2} = e^{-2\pi\left(\sqrt{-1}\right)/n2}$$
$$W_{n3} = e^{-2\pi\left(\sqrt{-1}\right)/n3}$$

and where:

$x_{j1,j2,j3}$ are elements of array X.
$y_{k1,k2,k3}$ are elements of array Y.
Isign is + or – (determined by argument *isign*).
scale is a scalar value.

Because of the complex conjugate symmetry, the output in array Y is real. For *scale* = 1.0 and *isign* being positive, you obtain the discrete Fourier transform, a function of frequency. The inverse Fourier transform is obtained with *scale* = 1.0/((*n1*)(*n2*)(*n3*)) and *isign* being negative. See references [1], [4], [5], [19], and [20].

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors: None

Input-Argument Errors

1. *n1* > 37748736
2. *n2* > 37748736
3. *n3* > 37748736
4. *inc2x* < *n1*/2+1
5. *inc3x* < (*n2*)(*inc2x*)
6. *inc2y* < *n1*+2
7. *inc3y* < (*n2*)(*inc2y*)
8. *scale* = 0.0
9. *isign* = 0
10. The length of one of the transforms in *n1*, *n2*, or *n3* is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
11. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example: This example shows how to compute a three-dimensional transform. In this example, INC2Y = (2)(INC2X) and INC3Y = (2)(INC3X), so that the same array can be used for both input and output. The STRIDE subroutine is called to select good values for the INC2Y and INC3Y strides. (As explained below, STRIDE is not called for INC2X and INC3X.) Using the transform lengths (N1 = 32, N2 = 64, and N3 = 40) along with the output data type (short-precision real: 'S'), STRIDE is called once for each stride needed. First, it is called for INC2Y:

```
CALL STRIDE (N2,N1+2,INC2Y,'S',0)
```

The output value returned for INC2Y is 34. (This value is equal to N1+2.) Then STRIDE is called again for INC3Y:

```
CALL STRIDE (N3,N2*INC2Y,INC3Y,'S',0)
```

The output value returned for INC3Y is 2176. Because INC3Y is a multiple of INC2Y—that is, INC3Y = (N2)(INC2Y)—Y is declared as a three-dimensional array, Y(34,64,40). (In general, for larger arrays, these types of values for INC2Y and INC3Y are not returned by STRIDE, and you are probably not able to declare Y as a three-dimensional array.)

A good stride value for INC2X is INC2Y/2, and a good stride value for INC3X is INC3Y/2. Also, to equivalence the X and Y arrays requires INC2Y = (2)(INC2X) and INC3Y = (2)(INC3X). Therefore, the values INC2X = INC2Y/2 = 17 and INC3X = INC3Y/2 = 1088 are set, and X is declared as a three-dimensional array, X(17,64,40).

The arrays are declared as follows:

```
COMPLEX*8  X(17,64,40)
REAL*4     Y(34,64,40)
REAL*8     AUX(32000)
```

Arrays X and Y are made equivalent by the following statement, making them occupy the same storage:

```
EQUIVALENCE (X,Y)
```

Call Statement and Input

```
             X  INC2X INC3X  Y  INC2Y INC3Y  N1   N2   N3 ISIGN SCALE   AUX    NAUX
             |    |     |    |    |     |    |    |    |   |     |      |      |
CALL SCRFT3( X , 17 , 1088 , Y , 34 , 2176 , 32 , 64 , 40 , 1 , SCALE , AUX , 32000)
```

```
SCALE    =  1.0
```

X has (1.0,0.0) in location X(1,1,1) and (0.0,0.0) in all other locations.

Output: Y has 1.0 in all locations.

Convolution and Correlation Subroutines

This section contains the convolution and correlation subroutine descriptions.

SCON and SCOR—Convolution or Correlation of One Sequence with One or More Sequences

These subroutines compute the convolutions and correlations of a sequence with one or more sequences using a direct method. The input and output sequences contain short-precision real numbers.

Note: These subroutines are considered obsolete. They are provided in ESSL only for compatibility with earlier releases. You should use SCOND, SCORD, SDCON, SDCOR, SCONF, and SCORF instead, because they provide **better performance**. For further details, see "Convolution and Correlation Considerations" on page 745.

Syntax

Fortran	CALL SCON | SCOR (*init*, *h*, *inc1h*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *nh*, *nx*, *m*, *iy0*, *ny*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	scon | scor (*init*, *h*, *inc1h*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *nh*, *nx*, *m*, *iy0*, *ny*, *aux1*, *naux1*, *aux2*, *naux2*);
PL/I	CALL SCON | SCOR (*init*, *h*, *inc1h*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *nh*, *nx*, *m*, *iy0*, *ny*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init
is a flag, where:

If *init* ≠ 0, no computation is performed, error checking is performed, and the subroutine exits back to the calling program.

If *init* = 0, the convolutions or correlations of the sequence in *h* with the sequences in *x* are computed.

Specified as: a fullword integer. It can have any value.

h
is the array H, consisting of the sequence of length N_h to be convolved or correlated with the sequences in array X. Specified as: an array of (at least) length $1+(N_h-1)|inc1h|$, containing short-precision real numbers.

inc1h
is the stride between the elements within the sequence in array H. Specified as: a fullword integer; *inc1h* > 0.

x
is the array X, consisting of *m* input sequences of length N_x, each to be convolved or correlated with the sequence in array H. Specified as: an array of (at least) length $1 + (m-1)inc2x + (N_x-1)inc1x$, containing short-precision real numbers.

inc1x
is the stride between the elements within each sequence in array X. Specified as: a fullword integer; *inc1x* > 0.

inc2x
is the stride between the first elements of the sequences in array X. Specified as: a fullword integer; *inc2x* > 0.

y
See On Return.

inc1y
is the stride between the elements within each sequence in output array Y. Specified as: a fullword integer; *inc1y* > 0.

inc2y
is the stride between the first elements of each sequence in output array Y. Specified as: a fullword integer; *inc2y* > 0.

nh
is the number of elements, N_h, in the sequence in array H. Specified as: a fullword integer; $N_h > 0$.

nx
is the number of elements, N_x, in each sequence in array X. Specified as: a fullword integer; $N_x > 0$.

m
is the number of sequences in array X to be convolved or correlated. Specified as: a fullword integer; $m > 0$.

iy0
is the convolution or correlation index of the element to be stored in the first position of each sequence in array Y. Specified as: a fullword integer. It can have any value.

ny
is the number of elements, N_y, in each sequence in array Y. Specified as: a fullword integer; $N_y > 0$ for SCON and $N_y \geq -N_h+1$ for SCOR.

aux1
is no longer used in the computation, but must still be specified as a dummy argument (for migration purposes from Version 1 of ESSL). It can have any value.

naux1
is no longer used in the computation, but must still be specified as a dummy argument (for migration purposes from Version 1 of ESSL). It can have any value.

aux2
is no longer used in the computation, but must still be specified as a dummy argument (for migration purposes from Version 1 of ESSL). It can have any value.

naux2
is no longer used in the computation, but must still be specified as a dummy argument (for migration purposes from Version 1 of ESSL). It can have any value.

On Return

y
is array Y, consisting of *m* output sequences of length N_y that are the result of the convolutions or correlations of the sequence in array H with the sequences in array X. Returned as: an array of (at least) length $1 + (m-1)inc2y + (N_y-1)inc1y$, containing short-precision real numbers.

Notes

1. Output should not overwrite input; that is, input arrays X and H must have no common elements with output array Y. Otherwise, results are unpredictable. See "Concepts" on page 55.
2. Auxiliary storage is not needed, but the arguments *aux1*, *naux1*, *aux2*, and *naux2* must still be specified. You can assign any values to these arguments.

Function: The convolutions and correlations of a sequence in array H with one or more sequences in array X are expressed as follows:

Convolutions for SCON:

$$y_{ki} = \sum_{j=\max(0,k-N_x+1)}^{\min(N_h-1,k)} h_j x_{k-j,i}$$

Correlations for SCOR:

$$y_{ki} = \sum_{j=\max(0,-k)}^{\min(N_h-1,\ N_x-1-k)} h_j x_{k+j,i}$$

for:

$k = iy0, iy0+1, ..., iy0+N_y-1$
$i = 1, 2, ..., m$

where:

y_{ki} are elements of the m sequences of length N_y in array Y.
x_{ki} are elements of the m sequences of length N_x in array X.
h_j are elements of the sequence of length N_h in array H.
iy0 is the convolution or correlation index of the element to be stored in the first position of each sequence in array Y.
min and max select the minimum and maximum values, respectively.

It is assumed that elements outside the range of definition are zero. See references [17] and [78].

Only one invocation of this subroutine is needed:

1. You do not need to invoke the subroutine with *init* ≠ 0. If you do, however, the subroutine performs error checking, exits back to the calling program, and no computation is performed.
2. With *init* = 0, the subroutine performs the calculation of the convolutions or correlations.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. *nh*, *nx*, *ny*, or $m \le 0$
2. *inc1h*, *inc1x*, *inc2x*, *inc1y*, or *inc2y* ≤ 0

Example 1: This example shows how to compute a convolution of a sequence in H, which is a ramp function, and three sequences in X, a triangular function and its cyclic translates. It computes the full range of nonzero values of the convolution plus two extra points, which are set to 0. The arrays are declared as follows:

```
REAL*4   H(0:4999), X(0:49999), Y(0:49999)
REAL*8   AUX1, AUX2
```

Call Statement and Input

```
          INIT  H INC1H X INC1X INC2X Y INC1Y INC2Y NH NX  M IY0 NY  AUX1 NAUX1 AUX2 NAUX2
           |    |   |   |   |     |   |   |     |    | |  |  |   |    |     |    |     |
CALL SCON(INIT, H , 1 , X , 1 ,  10 , Y , 1 ,  15 , 4, 10, 3, 0, 15, AUX1 , 0 , AUX2 , 0)
```

```
INIT     =  0(for computation)
H        =  (1.0, 2.0, 3.0, 4.0)
```

X contains the following three sequences:

```
1.0  2.0  3.0
2.0  1.0  2.0
3.0  2.0  1.0
4.0  3.0  2.0
5.0  4.0  3.0
6.0  5.0  4.0
5.0  6.0  5.0
4.0  5.0  6.0
3.0  4.0  5.0
2.0  3.0  4.0
```

Output: Y contains the following three sequences:

```
 1.0   2.0   3.0
 4.0   5.0   8.0
10.0  10.0  14.0
20.0  18.0  22.0
30.0  20.0  18.0
40.0  30.0  20.0
48.0  40.0  30.0
52.0  48.0  40.0
50.0  52.0  48.0
40.0  50.0  52.0
29.0  38.0  47.0
18.0  25.0  32.0
 8.0  12.0  16.0
 0.0   0.0   0.0
 0.0   0.0   0.0
```

Example 2: This example shows how the output from Example 1 differs when the values for NY and *inc2y* are 10 rather than 15. The output is the same except that it consists of only the first 10 values produced in Example 1.

Output: Y contains the following three sequences:

```
 1.0   2.0   3.0
 4.0   5.0   8.0
10.0  10.0  14.0
20.0  18.0  22.0
30.0  20.0  18.0
40.0  30.0  20.0
48.0  40.0  30.0
52.0  48.0  40.0
50.0  52.0  48.0
40.0  50.0  52.0
```

Example 3: This example shows how the output from Example 2 differs if the value for IY0 is 3 rather than 0. The output is the same except it starts at element 3 of the convolution sequences rather than element 0.

Output: Y contains the following three sequences:

```
20.0  18.0  22.0
30.0  20.0  18.0
40.0  30.0  20.0
48.0  40.0  30.0
52.0  48.0  40.0
50.0  52.0  48.0
40.0  50.0  52.0
29.0  38.0  47.0
18.0  25.0  32.0
 8.0  12.0  16.0
```

Example 4: This example shows how to compute a correlation of a sequence in H, which is a ramp function, and three sequences in X, a triangular function and its cyclic translates. It computes the full range of nonzero values of the correlation plus two extra points, which are set to 0. The arrays are declared as follows:

```
REAL*4   H(0:4999), X(0:49999), Y(0:49999)
REAL*8   AUX1, AUX2
```

Call Statement and Input

```
          INIT  H INC1H X INC1X INC2X Y INC1Y INC2Y NH NX  M IY0  NY  AUX1  NAUX1 AUX2 NAUX2
           |    |   |   |   |     |   |   |     |   | |   |   |   |   |      |    |     |
CALL SCOR(INIT, H , 1 , X , 1  , 10 , Y , 1  , 15 , 4, 10, 3, -3, 15, AUX1 ,  0 , AUX2 , 0)
```

```
INIT     =  0(for computation)
H        =  (1.0, 2.0, 3.0, 4.0)
```

X contains the following three sequences:

```
1.0  2.0  3.0
2.0  1.0  2.0
3.0  2.0  1.0
4.0  3.0  2.0
5.0  4.0  3.0
6.0  5.0  4.0
5.0  6.0  5.0
4.0  5.0  6.0
3.0  4.0  5.0
2.0  3.0  4.0
```

Output: Y contains the following three sequences:

```
 4.0   8.0  12.0
11.0  10.0  17.0
20.0  15.0  16.0
30.0  22.0  18.0
40.0  30.0  22.0
50.0  40.0  30.0
52.0  50.0  40.0
48.0  52.0  50.0
40.0  48.0  52.0
30.0  40.0  48.0
16.0  22.0  28.0
 7.0  10.0  13.0
 2.0   3.0   4.0
 0.0   0.0   0.0
 0.0   0.0   0.0
```

Example 5: This example shows how the output from Example 4 differs when the values for NY and INC2Y are 10 rather than 15. The output is the same except that it consists of only the first 10 values produced in Example 4.

Output: Y contains the following three sequences:

```
 4.0   8.0  12.0
11.0  10.0  17.0
20.0  15.0  16.0
30.0  22.0  18.0
40.0  30.0  22.0
50.0  40.0  30.0
52.0  50.0  40.0
48.0  52.0  50.0
40.0  48.0  52.0
30.0  40.0  48.0
```

Example 6: This example shows how the output from Example 5 differs if the value for IY0 is 0 rather than –3. The output is the same except it starts at element 0 of the correlation sequences rather than element –3.

Output: Y contains the following three sequences:

```
30.0  22.0  18.0
40.0  30.0  22.0
50.0  40.0  30.0
52.0  50.0  40.0
48.0  52.0  50.0
40.0  48.0  52.0
30.0  40.0  48.0
16.0  22.0  28.0
 7.0  10.0  13.0
 2.0   3.0   4.0
```

SCOND and SCORD—Convolution or Correlation of One Sequence with Another Sequence Using a Direct Method

These subroutines compute the convolution and correlation of a sequence with another sequence using a direct method. The input and output sequences contain short-precision real numbers.

Notes:

1. These subroutines compute the convolution and correlation using direct methods. In most cases, these subroutines provide **better performance** than using SCON or SCOR, if you determine that SCON or SCOR would have used a direct method for its computation. For information on how to make this determination, see reference [4].
2. For long-precision data, you should use DDCON or DDCOR with the decimation rate, *id*, equal to 1.

Syntax

Fortran	CALL SCOND | SCORD (*h*, *inch*, *x*, *incx*, *y*, *incy*, *nh*, *nx*, *iy0*, *ny*)
C and C++	scond | scord (*h*, *inch*, *x*, *incx*, *y*, *incy*, *nh*, *nx*, *iy0*, *ny*);
PL/I	CALL SCOND | SCORD (*h*, *inch*, *x*, *incx*, *y*, *incy*, *nh*, *nx*, *iy0*, *ny*);

On Entry

h
is the array H, consisting of the sequence of length N_h to be convolved or correlated with the sequence in array X. Specified as: an array of (at least) length $1+(N_h-1)|inch|$, containing short-precision real numbers.

inch
is the stride between the elements within the sequence in array H. Specified as: a fullword integer; $inch > 0$ or $inch < 0$.

x
is the array X, consisting of the input sequence of length N_x, to be convolved or correlated with the sequence in array H. Specified as: an array of (at least) length $1+(N_x-1)|incx|$, containing short-precision real numbers.

incx
is the stride between the elements within the sequence in array X. Specified as: a fullword integer; $incx > 0$ or $incx < 0$.

y
See On Return.

incy
is the stride between the elements within the sequence in output array Y. Specified as: a fullword integer; $incy > 0$ or $incy < 0$.

nh
is the number of elements, N_h, in the sequence in array H. Specified as: a fullword integer; $N_h > 0$.

nx
is the number of elements, N_x, in the sequence in array X. Specified as: a fullword integer; $N_x > 0$.

iy0
is the convolution or correlation index of the element to be stored in the first position of the sequence in array Y. Specified as: a fullword integer. It can have any value.

ny
is the number of elements, N_y, in the sequence in array Y. Specified as: a fullword integer; $N_y > 0$.

On Return

y
is the array Y of length N_y, consisting of the output sequence that is the result of the convolution or correlation of the sequence in array H with the sequence in array X. Returned as: an array of (at least) length $1+(N_y-1)|incy|$, containing short-precision real numbers.

Notes

1. Output should not overwrite input—that is, input arrays X and H must have no common elements with output array Y. Otherwise, results are unpredictable. See "Concepts" on page 55.
2. If *iy0* and *ny* are such that output outside the basic range is needed, where the basic range is $0 \le k \le (nh+nx-2)$ for SCOND and $(-nh+1) \le k \le (nx-1)$ for SCORD, the subroutine stores zeros using scalar code. It is not efficient to store many zeros in this manner. It is more efficient to set *iy0* and *ny* so that the output is produced within the above range of *k* values.

Function: The convolution and correlation of a sequence in array H with a sequence in array X are expressed as follows:

Convolution for SCOND:

$$y_k = \sum_{j=\max(0,k-N_x+1)}^{\min(N_h-1,k)} h_j x_{k-j}$$

Correlation for SCORD:

$$y_k = \sum_{j=\max(0,-k)}^{\min(N_h-1,\ N_x-1-k)} h_j x_{k+j}$$

for $k = iy0, iy0+1, ..., iy0+N_y-1$

where:

y_k are elements of the sequence of length N_y in array Y.
x_k are elements of the sequence of length N_x in array X.
h_j are elements of the sequence of length N_h in array H.
iy0 is the convolution or correlation index of the element to be stored in the first position of each sequence in array Y.
min and max select the minimum and maximum values, respectively.

It is assumed that elements outside the range of definition are zero. See reference [4].

Special Usage: SCORD can also perform the functions of SCON and SACOR; that is, it can compute convolutions and autocorrelations. To compute a convolution, you must specify a negative stride for H (see Example 9). To compute the autocorrelation, you must specify the two input sequences to be the same (see Example 10). In fact, you can also compute the autoconvolution by using both of these techniques together, letting the two input sequences be the same, and specifying a negative stride for the first input sequence.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. *nh*, *nx*, or *ny* ≤ 0
2. *inch*, *incx*, or *incy* = 0

Example 1: This example shows how to compute a convolution of a sequence in H with a sequence in X, where both sequences are ramp functions.

Call Statement and Input

```
            H  INCH  X  INCX  Y  INCY  NH  NX IY0  NY
            |   |    |   |    |   |    |   |   |   |
CALL SCOND( H , 1  , X , 1  , Y , 1  , 4 , 8 , 0 , 11 )

H        =  (1.0, 2.0, 3.0, 4.0)
X        =  (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
```

Output

```
Y        =  (11.0, 34.0, 70.0, 120.0, 130.0, 140.0, 150.0, 160.0,
             151.0, 122.0, 72.0)
```

Example 2: This example shows how the output from Example 1 differs when the value for IY0 is –2 rather than 0, and NY is 15 rather than 11. The output has two zeros at the beginning and end of the sequence, for points outside the range of nonzero output.

Call Statement and Input

```
            H  INCH  X  INCX  Y  INCY  NH  NX  IY0  NY
            |   |    |   |    |   |    |   |    |   |
CALL SCOND( H , 1  , X , 1  , Y , 1  , 4 , 8 , -2 , 15 )

H        =  (1.0, 2.0, 3.0, 4.0)
X        =  (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
```

Output

```
Y        =  (0.0, 0.0, 11.0, 34.0, 70.0, 120.0, 130.0, 140.0, 150.0,
             160.0, 151.0, 122.0, 72.0, 0.0, 0.0)
```

Example 3: This example shows how the same output as Example 1 can be obtained when H and X are interchanged, because the convolution is symmetric in H and X. (The arguments are switched in the calling sequence.)

Call Statement and Input

```
            H  INCH  X  INCX  Y  INCY  NH  NX IY0  NY
            |   |    |   |    |   |    |   |   |   |
CALL SCOND( X , 1  , H , 1  , Y , 1  , 4 , 8 , 0 , 11 )

H        =  (1.0, 2.0, 3.0, 4.0)
X        =  (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
```

Output

```
Y        =  (11.0, 34.0, 70.0, 120.0, 130.0, 140.0, 150.0, 160.0,
             151.0, 122.0, 72.0)
```

Example 4: This example shows how the output from Example 1 differs when a negative stride is specified for the sequence in H. By reversing the H sequence, the correlation is computed.

Call Statement and Input

```
            H  INCH  X  INCX  Y  INCY  NH  NX IY0  NY
            |    |   |   |    |   |    |   |   |   |
CALL SCOND( H , -1 , X , 1  , Y , 1  , 4 , 8 , 0 , 11 )

H        =  (1.0, 2.0, 3.0, 4.0)
X        =  (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
```

Output

```
Y        =  (44.0, 81.0, 110.0, 130.0, 140.0, 150.0, 160.0, 170.0,
             104.0, 53.0, 18.0)
```

Example 5: This example shows how to compute the autoconvolution of a sequence by letting the two input sequences for H and X be the same. (X is specified for both arguments in the calling sequence.)

Call Statement and Input

```
            H  INCH  X  INCX  Y  INCY NH  NX IY0  NY
            |   |   |   |    |   |    |   |   |   |
CALL SCOND( X , 1 , X , 1  , Y , 1  , 4 , 4 , 0 , 7 )

X        =  (11.0, 12.0, 13.0, 14.0)
```

Output

```
Y        =  (121.0, 264.0, 430.0, 620.0, 505.0, 364.0, 196.0)
```

Example 6: This example shows how to compute a correlation of a sequence in H with a sequence in X, where both sequences are ramp functions.

Call Statement and Input

```
            H  INCH  X  INCX  Y  INCY NH  NX   IY0  NY
            |   |    |   |    |   |    |   |    |   |
CALL SCORD( H , 1  , X , 1  , Y , 1  , 4 , 8 , -3 , 11 )

H        =  (1.0, 2.0, 3.0, 4.0)
X        =  (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
```

Output

```
Y       =  (44.0, 81.0, 110.0, 130.0, 140.0, 150.0, 160.0, 170.0,
            104.0, 53.0, 18.0)
```

Example 7: This example shows how the output from Example 6 differs when the value for IY0 is –5 rather than –3 and NY is 15 rather than 11. The output has two zeros at the beginning and end of the sequence, for points outside the range of nonzero output.

Call Statement and Input

```
           H  INCH  X  INCX  Y  INCY NH  NX   IY0  NY
           |   |    |   |    |   |    |   |    |   |
CALL SCORD( H , 1  , X , 1  , Y , 1  , 4 , 8 , -5 , 15 )

H       =  (1.0, 2.0, 3.0, 4.0)
X       =  (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
```

Output

```
Y       =  (0.0, 0.0, 44.0, 81.0, 110.0, 130.0, 140.0, 150.0, 160.0,
            170.0, 104.0, 53.0, 18.0, 0.0, 0.0)
```

Example 8: This example shows how the output from Example 6 differs when H and X are interchanged (in the calling sequence). The output sequence is the reverse of that in Example 6. To get the full range of output, IY0 is set to –NX+1.

Call Statement and Input

```
           H  INCH  X  INCX  Y  INCY NH  NX   IY0  NY
           |   |    |   |    |   |    |   |    |   |
CALL SCORD( X , 1  , H , 1  , Y , 1  , 4 , 8 , -7 , 11 )

H       =  (1.0, 2.0, 3.0, 4.0)
X       =  (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
```

Output

```
Y       =  (18.0, 53.0, 104.0, 170.0, 160.0, 150.0, 140.0, 130.0,
            110.0, 81.0, 44.0)
```

Example 9: This example shows how the output from Example 6 differs when a negative stride is specified for the sequence in H. By reversing the H sequence, the convolution is computed.

Call Statement and Input

```
           H  INCH  X  INCX  Y  INCY  NH  NX  IY0  NY
           |    |   |   |    |   |    |   |   |   |
CALL SCORD( H , -1 , X , 1  , Y , 1  , 4 , 8 , -3 , 11 )

H       =  (1.0, 2.0, 3.0, 4.0)
X       =  (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
```

Output

```
Y       =  (11.0, 34.0, 70.0, 120.0, 130.0, 140.0, 150.0, 160.0,
            151.0, 122.0, 72.0)
```

Example 10: This example shows how to compute the autocorrelation of a sequence by letting the two input sequences for H and X be the same. (X is specified for both arguments in the calling sequence.)

Call Statement and Input

```
            H  INCH  X  INCX  Y  INCY  NH  NX  IY0  NY
            |   |   |   |   |   |   |   |   |   |
CALL SCORD( X , 1  , X , 1  , Y , 1  , 4 , 4 , -3 , 7 )

X        =  (11.0, 12.0, 13.0, 14.0)
```

Output

```
Y        =  (154.0, 311.0, 470.0, 630.0, 470.0, 311.0, 154.0)
```

SCONF and SCORF—Convolution or Correlation of One Sequence with One or More Sequences Using the Mixed-Radix Fourier Method

These subroutines compute the convolutions and correlations, respectively, of a sequence with one or more sequences using the mixed-radix Fourier method. The input and output sequences contain short-precision real numbers.

Note: Two invocations of these subroutines are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.

Syntax

Fortran	CALL SCONF | SCORF (*init*, *h*, *inc1h*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *nh*, *nx*, *m*, *iy0*, *ny*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	sconf | scorf (*init*, *h*, *inc1h*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *nh*, *nx*, *m*, *iy0*, *ny*, *aux1*, *naux1*, *aux2*, *naux2*);
PL/I	CALL SCONF | SCORF (*init*, *h*, *inc1h*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *nh*, *nx*, *m*, *iy0*, *ny*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init
is a flag, where:

If *init* ≠ 0, trigonometric functions, the transform of the sequence in *h*, and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*. The contents of *x* and *y* are not used or changed.

If *init* = 0, the convolutions or correlations of the sequence that was in *h* at initialization with the sequences in *x* are computed. *h* is not used or changed. The only arguments that may change after initialization are *x*, *y*, and *aux2*. All scalar arguments must be the same as when the subroutine was called for initialization with *init* ≠ 0.

Specified as: a fullword integer. It can have any value.

h
is the array H, consisting of the sequence of length N_h to be convolved or correlated with the sequences in array X. Specified as: an array of (at least) length $1+(N_h-1)|inc1h|$, containing short-precision real numbers.

inc1h
is the stride between the elements within the sequence in array H. Specified as: a fullword integer; *inc1h* > 0.

x
is the array X, consisting of *m* input sequences of length N_x, each to be convolved or correlated with the sequence in array H. Specified as: an array of (at least) length $1+(N_x-1)inc1x+(m-1)inc2x$, containing short-precision real numbers.

inc1x
is the stride between the elements within each sequence in array X. Specified as: a fullword integer; *inc1x* > 0.

inc2x
is the stride between the first elements of the sequences in array X. Specified as: a fullword integer; *inc2x* > 0.

y
See On Return.

inc1y
is the stride between the elements within each sequence in output array Y. Specified as: a fullword integer; *inc1y* > 0.

inc2y
is the stride between the first elements of each sequence in output array Y. Specified as: a fullword integer; *inc2y* > 0.

nh
is the number of elements, N_h, in the sequence in array H. Specified as: a fullword integer; $N_h > 0$.

nx
is the number of elements, N_x, in each sequence in array X. Specified as: a fullword integer; $N_x > 0$.

m
is the number of sequences in array X to be convolved or correlated. Specified as: a fullword integer; $m > 0$.

iy0
is the convolution or correlation index of the element to be stored in the first position of each sequence in array Y. Specified as: a fullword integer. It can have any value.

ny
is the number of elements, N_y, in each sequence in array Y. Specified as: a fullword integer; $N_y > 0$.

aux1
is the working storage for this subroutine, where:

If *init* ≠ 0, the working storage is computed.

If *init* = 0, the working storage is used in the computation of the convolutions.

Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1
is the number of doublewords in the working storage specified in *aux1*. Specified as: a fullword integer; *naux1* > 23 and *naux1* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For values between 23 and the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 31.

aux2
has the following meaning:

If *naux2* = 0 and error 2015 is unrecoverable, *aux2* is ignored.

Otherwise, it is the working storage used by this subroutine, which is available for use by the calling program between calls to this subroutine.

Specified as: an area of storage, containing *naux2* long-precision real numbers. On output, the contents are overwritten.

naux2
is the number of doublewords in the working storage specified in *aux2*. Specified as: a fullword integer, where:

If *naux2* = 0 and error 2015 is unrecoverable, SCONF and SCORF dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux2* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other

values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 31.

On Return

y

has the following meaning, where:

If *init* ≠ 0, this argument is not used, and its contents remain unchanged.

If *init* = 0, this is array Y, consisting of *m* output sequences of length N_y that are the result of the convolutions or correlations of the sequence in array H with the sequences in array X.

Returned as: an array of (at least) length $1+(N_y-1)inc1y+(m-1)inc2y$, containing short-precision real numbers.

aux1

is the working storage for this subroutine, where:

If *init* ≠ 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If *init* = 0, its contents are unchanged.

Returned as: the contents are not relevant.

Notes

1. *aux1* should **not** be used by the calling program between calls to this subroutine with *init* ≠ 0 and *init* = 0. However, it can be reused after intervening calls to this subroutine with different arguments.
2. If you specify the same array for X and Y, then *inc1x* and *inc1y* must be equal, and *inc2x* and *inc2y* must be equal. In this case, output overwrites input.
3. If you specify different arrays for X and Y, they must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.
4. If *iy0* and *ny* are such that output outside the basic range is needed, the subroutine stores zeros. These ranges are: $0 \le k \le N_x+N_h-2$ for SCONF and $1-N_h \le k \le N_x-1$ for SCORF.

Formulas for the Length of the Fourier Transform: Before calculating the necessary sizes of *naux1* and *naux2*, you must determine the length *n* of the Fourier transform. The value of *n* is based on *nf*. You can use one of two techniques to determine *nf*:

- Use the simple overestimate of $nf = nx+nh-1$. (If $iy0 = 0$ and $ny > nh+nx$, this is the actual value, not an overestimate.)
- Use the values of the arguments *iy0*, *nh*, *nx*, and *ny* inserted into the following formulas to get a value for the variable *nf*:

  $iy0p = \max(iy0, 0)$
  $ix0 = \max((iy0p+1)-nh, 0)$
  $ih0 = \max((iy0p+1)-nx, 0)$
  $nd = ix0+ih0$
  $n1 = iy0+ny$
  $nxx = \min(n1, nx)-ix0$
  $nhh = \min(n1, nh)-ih0$

ntt = *nxx*+*nhh*−1
nn1 = *n1*−*nd*
iyy0 = *iy0p*−*nd*
nzleft = max(0, *nhh*−*iyy0*−1)
nzrt = min(*nn1*, *ntt*)−*nxx*
nf = max(12, *nxx*+max(*nzleft*, *nzrt*))

After calculating the value for *nf*, using one of these two techniques, refer to the formula or table of allowable values of *n* in "Acceptable Lengths for the Transforms" on page 741, selecting the value equal to or greater than *nf*.

Processor-Independent Formulas for NAUX1 and NAUX2: The required values of *naux1* and *naux2* depend on the value determined for *n* in "Formulas for the Length of the Fourier Transform" on page 845.

NAUX1 Formulas

If $n \le 16384$, use *naux1* = 58000.
If $n > 16384$, use *naux1* = 40000+2.14*n*.

NAUX2 Formulas

If $n \le 16384$, use *naux2* = 30000.
If $n > 16384$, use *naux2* = 20000+1.07*n*.

Function: The convolutions and correlations of a sequence in array H with one or more sequences in array X are expressed as follows.

Convolutions for SCONF:

$$y_{ki} = \sum_{j=\max(0,k-N_x+1)}^{\min(N_h-1,k)} h_j x_{k-j,i}$$

Correlations for SCORF:

$$y_{ki} = \sum_{j=\max(0,-k)}^{\min(N_h-1,\ N_x-1-k)} h_j x_{k+j,i}$$

for:

k = *iy0*, *iy0*+1, ..., *iy0*+N_y−1
i = 1, 2, ..., *m*

where:

y_{ki} are elements of the *m* sequences of length N_y in array Y.
x_{ki} are elements of the *m* sequences of length N_x in array X.
h_j are elements of the sequence of length N_h in array H.
iy0 is the convolution or correlation index of the element to be stored in the first position of each sequence in array Y.

min and max select the minimum and maximum values, respectively.

These subroutines use a Fourier transform method with a mixed-radix capability. This provides maximum performance for your application. The length of the transform, *n*, that you must calculate to determine the correct sizes for *naux1* and *naux2* is the same length used by the Fourier transform subroutines called by this subroutine. It is assumed that elements outside the range of definition are zero. See references [17] and [78].

Two invocations of this subroutine are necessary:

1. With *init* ≠ 0, the subroutine tests and initializes arguments of the program, setting up the *aux1* working storage.
2. With *init* = 0, the subroutine checks that the initialization arguments in the *aux1* working storage correspond to the present arguments, and if so, performs the calculation of the convolutions.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux2* = 0, and unable to allocate work area.

Computational Errors: None

Input-Argument Errors

1. *nh*, *nx*, *ny*, or *m* ≤ 0
2. *inc1h*, *inc1x*, *inc2x*, *inc1y*, or *inc2y* ≤ 0
3. The resulting internal Fourier transform length *n*, is too large. See "Convolutions and Correlations by Fourier Methods" on page 747.
4. The subroutine has not been initialized with the present arguments.
5. *naux1* ≤ 23
6. *naux1* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
7. Error 2015 is recoverable or *naux2*≠0, and *naux2* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example 1: This example shows how to compute a convolution of a sequence in H, where H and X are ramp functions. It calculates all nonzero values of the convolution of the sequences in H and X. The arrays are declared as follows:

```
REAL*4   H(8), X(10,1), Y(17)
```

Because this convolution is symmetric in H and X, you can interchange the H and X sequences, leaving all other arguments the same, and you get the same output shown below. First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Call Statement and Input

```
           INIT  H INC1H X INC1X INC2X Y INC1Y INC2Y NH  NX  M IY0 NY  AUX1 NAUX1 AUX2 NAUX2
            |    |   |   |   |     |   |   |     |    |   | |  |   |    |     |    |     |
CALL SCONF(INIT, H , 1 , X , 1  ,  1 , Y,  1  ,  1  , 8, 10, 1, 0, 17, AUX1, 128, AUX2, 23)
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
H        =  (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0)
X        =  (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0, 19.0, 20.0)
```

Output

```
Y        =  (11.0, 34.0, 70.0, 120.0, 185.0, 266.0, 364.0, 480.0,
             516.0, 552.0, 567.0, 560.0, 530.0, 476.0, 397.0, 292.0,
             160.0)
```

Example 2: This example shows how the output from Example 1 differs when the value for NY is 21 rather than 17, and the value for IY0 is −2 rather than 0. This yields two zeros on each end of the convolution.

Output

```
Y        =  (0.0, 0.0, 11.0, 34.0, 70.0, 120.0, 185.0, 266.0, 364.0,
             480.0, 516.0, 552.0, 567.0, 560.0, 530.0, 476.0, 397.0,
             292.0, 160.0, 0.0, 0.0)
```

Example 3: This example shows how to compute the autoconvolution by letting the two input sequences be the same for Example 2. First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Call Statement and Input

```
           INIT  H INC1H X INC1X INC2X Y INC1Y INC2Y NH  NX  M  IY0  NY  AUX1 NAUX1 AUX2 NAUX2
            |    |   |   |   |     |   |   |     |    |   | |   |    |   |     |    |     |
CALL SCONF(INIT, H , 1 , H , 1  ,  1 , Y,  1  ,  1  , 8, 10, 1, -2, 21, AUX1, 128, AUX2, 23)
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
```

Output

```
Y        =  (1.0, 4.0, 10.0, 20.0, 35.0, 56.0, 84.0, 120.0, 147.0,
             164.0, 170.0, 164.0, 145.0, 112.0, 64.0)
```

Example 4: This example shows how to compute all nonzero values of the convolution of the sequence in H with the two sequences in X. First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Call Statement and Input

```
           INIT  H INC1H X INC1X INC2X Y INC1Y INC2Y NH  NX  M IY0 NY  AUX1 NAUX1 AUX2 NAUX2
            |    |   |   |   |     |   |   |     |    |   | |   |   |    |     |    |     |
CALL SCONF(INIT, H , 1 , X,  1  , 10 , Y,  1  , 17  , 8, 10, 2, 0, 17, AUX1, 148, AUX2, 43)
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
H        =  (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0)
```

X contains the following two sequences:

```
11.0  12.0
12.0  13.0
13.0  14.0
14.0  15.0
15.0  16.0
16.0  17.0
17.0  18.0
18.0  19.0
19.0  20.0
20.0  11.0
```

Output: Y contains the following two sequences:

```
 11.0   12.0
 34.0   37.0
 70.0   76.0
120.0  130.0
185.0  200.0
266.0  287.0
364.0  392.0
480.0  516.0
516.0  552.0
552.0  578.0
567.0  582.0
560.0  563.0
530.0  520.0
476.0  452.0
397.0  358.0
292.0  237.0
160.0   88.0
```

Example 5: This example shows how to compute a correlation of a sequence in H, where H and X are ramp functions. It calculates all nonzero values of the correlation of the sequences in H and X. The arrays are declared as follows:

```
REAL*4   H(8), X(10,1)
```

First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Call Statement and Input

```
           INIT  H INC1H X INC1X INC2X Y INC1Y INC2Y NH  NX  M  IY0 NY  AUX1 NAUX1 AUX2 NAUX2
            |    |   |   |   |     |   |   |     |    |   | |   |   |    |     |    |     |
CALL SCORF(INIT, H , 1 , X,  1  ,  1 , Y,  1  ,  1  , 8, 10, 1, -7, 17, AUX1, 128, AUX2, 23)
```

```
INIT      =  1(for initialization)
INIT      =  0(for computation)
H         =(same as input H in Example 1)
X         =(same as input X in Example 1)
```

Output

```
Y         =  (88.0, 173.0, 254.0, 330.0, 400.0, 463.0, 518.0, 564.0,
              600.0, 636.0, 504.0, 385.0, 280.0, 190.0, 116.0,
              59.0, 20.0)
```

Example 6: This example shows how the output from Example 5 differs when the value for NY is 21 rather than 17, and the value for IY0 is –9 rather than 0. This yields two zeros on each end of the correlation.

Output

```
Y       =  (0.0, 0.0, 88.0, 173.0, 254.0, 330.0, 400.0, 463.0, 518.0,
            564.0, 600.0, 636.0, 504.0, 385.0, 280.0, 190.0, 116.0,
            59.0, 20.0, 0.0, 0.0)
```

Example 7: This example shows the effect of interchanging H and X. It uses the same input as Example 5, with H and X switched in the calling sequence, and with IY0 with a value of –9. Unlike convolution, as noted in Example 1, the correlation is not symmetric in H and X. First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Call Statement and Input

```
          INIT  H INC1H X INC1X INC2X Y INC1Y INC2Y NH  NX  M  IY0 NY  AUX1 NAUX1 AUX2 NAUX2
           |    |   |   |   |     |   |   |     |   |   | |   |   |   |     |    |     |
CALL SCONF(INIT, X , 1 , H,  1  ,  1 , Y,  1  ,  1  , 8, 10, 1, -9, 17, AUX1, 128, AUX2, 23)
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
```

Output

```
Y       =  (20.0, 59.0, 116.0, 190.0, 280.0, 385.0, 504.0, 636.0,
            600.0, 564.0, 518.0, 463.0, 400.0, 330.0, 254.0, 173.0,
            88.0)
```

Example 8: This example shows how to compute the autocorrelation by letting the two input sequences be the same. First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation. Because there is only one H input sequence, only one autocorrelation can be computed. Furthermore, this usage does not take advantage of the fact that the output is symmetric. Therefore, you should use SACORF to compute autocorrelations, because it does not have either of these problems.

Call Statement and Input

```
          INIT  H INC1H X INC1X INC2X Y INC1Y INC2Y NH NX M  IY0 NY  AUX1 NAUX1 AUX2 NAUX2
           |    |   |   |   |     |   |   |     |   | | |   |   |   |     |    |     |
CALL SCONF(INIT, H , 1 , H,  1  ,  1 , Y,  1  ,  1 , 8, 8, 1, -7, 15, AUX1, 148, AUX2, 43)
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
```

Output

```
Y       =  (8.0, 23.0, 44.0, 70.0, 100.0, 133.0, 168.0, 204.0, 168.0,
            133.0, 100.0 , 70.0, 44.0, 23.0, 8.0)
```

Example 9: This example shows how to compute all nonzero values of the correlation of the sequence in H with the two sequences in X. First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Call Statement and Input

```
           INIT  H INC1H X INC1X INC2X Y INC1Y INC2Y NH  NX  M  IY0 NY  AUX1 NAUX1 AUX2 NAUX2
            |    |   |   |   |     |   |   |     |    |   | |   |   |   |     |    |     |
CALL SCONF(INIT, H , 1 , X,  1  , 10 , Y,  1  , 17  , 8, 10, 2, -7, 17, AUX1, 148, AUX2, 43)
```

```
INIT      =  1(for initialization)
INIT      =  0(for computation)
H         =(same as input H in Example 4)
X         =(same as input X in Example 4)
```

Output: Y contains the following two sequences:

```
 88.0   96.0
173.0  188.0
254.0  275.0
330.0  356.0
400.0  430.0
463.0  496.0
518.0  553.0
564.0  600.0
600.0  636.0
636.0  592.0
504.0  462.0
385.0  346.0
280.0  245.0
190.0  160.0
116.0   92.0
 59.0   42.0
 20.0   11.0
```

SDCON, DDCON, SDCOR, and DDCOR—Convolution or Correlation with Decimated Output Using a Direct Method

These subroutines compute the convolution and correlation of a sequence with another sequence, with decimated output, using a direct method.

Table 137. Data Types

h, ***x***, ***y***	**Subroutine**
Short-precision real	SDCON
Long-precision real	DDCON
Short-precision real	SDCOR
Long-precision real	DDCOR

Note: These subroutines are the short- and long-precision equivalents of SCOND and SCORD when the decimation interval *id* is equal to 1. Because there is no long-precision version of SCOND and SCORD, you can use DDCON and DDCOR, respectively, with decimation interval *id* = 1 to perform the same function.

Syntax

Fortran	CALL SDCON | DDCON | SDCOR | DDCOR (*h*, *inch*, *x*, *incx*, *y*, *incy*, *nh*, *nx*, *iy0*, *ny*, *id*)
C and C++	sdcon | ddcon | sdcor | ddcor (*h*, *inch*, *x*, *incx*, *y*, *incy*, *nh*, *nx*, *iy0*, *ny*, *id*);
PL/I	CALL SDCON | DDCON | SDCOR | DDCOR (*h*, *inch*, *x*, *incx*, *y*, *incy*, *nh*, *nx*, *iy0*, *ny*, *id*);

On Entry

h
is the array H, consisting of the sequence of length N_h to be convolved or correlated with the sequence in array X. Specified as: an array of (at least) length $1+(N_h-1)|inch|$, containing numbers of the data type indicated in Table 137.

inch
is the stride between the elements within the sequence in array H. Specified as: a fullword integer; $inch > 0$ or $inch < 0$.

x
is the array X, consisting of the input sequence of length N_x, to be convolved or correlated with the sequence in array H. Specified as: an array of (at least) length $1+(N_x-1)|incx|$, containing numbers of the data type indicated in Table 137.

incx
is the stride between the elements within the sequence in array X. Specified as: a fullword integer; $incx > 0$ or $incx < 0$.

y
See On Return.

incy
is the stride between the elements within the sequence in output array Y. Specified as: a fullword integer; $incy > 0$ or $incy < 0$.

nh
is the number of elements, N_h, in the sequence in array H. Specified as: a fullword integer; $N_h > 0$.

nx
is the number of elements, N_x, in the sequence in array X. Specified as: a fullword integer; $N_x > 0$.

iy0
is the convolution or correlation index of the element to be stored in the first position of the sequence in array Y. Specified as: a fullword integer. It can have any value.

ny
is the number of elements, N_y, in the sequence in array Y. Specified as: a fullword integer; $N_y > 0$.

id
is the decimation interval *id* for the output sequence in array Y; that is, every *id*-th value of the convolution or correlation is produced. Specified as: a fullword integer; $id > 0$.

On Return

y
is the array Y of length N_y, consisting of the output sequence that is the result of the convolution or correlation of the sequence in array H with the sequence in array X, given for every *id*-th value in the convolution or correlation.

Returned as: an array of (at least) length $1+(N_y-1)|incy|$, containing numbers of the data type indicated in Table 137 on page 852.

Notes

1. If you specify the same array for X and Y, the following conditions must be true: $incx = incy$, $incx > 0$, $incy > 0$, $id = 1$, and $iy0 \geq N_h-1$ for _DCON and $iy0 \geq 0$ for _DCOR. In this case, output overwrites input. In all other cases, output should not overwrite input; that is, input arrays X and H must have no common elements with output array Y. Otherwise, results are unpredictable. See "Concepts" on page 55.
2. If *iy0* and *ny* are such that output outside the basic range is needed, where the basic range is $0 \leq k \leq (nh+nx-2)$ for SDCON and DDCON and is $(-nh+1) \leq k \leq (nx-1)$ for SDCOR and DDCOR, the subroutine stores zeros using scalar code. It is not efficient to store many zeros in this manner. If you anticipate that this will happen, you may want to adjust *iy0* and *ny*, so the subroutine computes only for *k* in the above range, or use the ESSL subroutine SSCAL or DSCAL to store the zeros, so you achieve better performance.

Function: The convolution and correlation of a sequence in array H with a sequence in array X, with decimated output, are expressed as follows:

Convolution for SDCON and DDCON:

$$y_k = \sum_{j=\max(0,k-N_x+1)}^{\min(N_h-1,k)} h_j x_{k-j}$$

Correlation for SDCOR and DDCOR:

$$y_k = \sum_{j=\max(0,-k)}^{\min(N_h-1,\ N_x-1-k)} h_j x_{k+j}$$

for $k = iy0, iy0+id, iy0+(2)id, ..., iy0+(N_y-1)id$

where:

y_k are elements of the sequence of length N_y in array Y.
x_k are elements of the sequence of length N_x in array X.
h_j are elements of the sequence of length N_h in array H.
iy0 is the convolution or correlation index of the element to be stored in the first position of the sequence in array Y.
min and max select the minimum and maximum values, respectively.

It is assumed that elements outside the range of definition are zero. See reference [4].

Special Usage: SDCON and DDCON can also perform a correlation, autoconvolution, or autocorrelation. To compute a correlation, you must specify a negative stride for H. To compute the autoconvolution, you must specify the two input sequences to be the same. You can also compute the autocorrelation by using both of these techniques together, letting the two input sequences be the same, and specifying a negative stride for the first input sequence. For examples of this, see the examples for SCOND on page 839. Because SCOND and SDCON are functionally the same, their results are the same as long as the decimation interval $id = 1$ for SDCON.

SDCOR and DDCOR can also perform a convolution, autocorrelation, or autoconvolution. To compute a convolution, you must specify a negative stride for H. To compute the autocorrelation, you must specify the two input sequences to be the same. You can also compute the autoconvolution by using both of these techniques together, letting the two input sequences be the same and specifying a negative stride for the first input sequence. For examples of these, see the examples for SCORD on page 840. Because SCORD and SDCOR are functionally the same, their results are the same as long as the decimation interval $id = 1$ for SDCOR.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. nh, nx, or $ny \leq 0$
2. $inch$, $incx$, or $incy = 0$
3. $id \leq 0$

Example 1: This example shows how to compute a convolution of a sequence in H with a sequence in X, where both sequences are ramp functions. It shows how a decimated output can be obtained, using the same input as "Example 1" on page 839 for SCOND and using a decimation interval ID = 2.

Note: For further examples of use, see the examples for SCOND on page 839. Because SCOND and SDCON are functionally the same, their results are the same as long as the decimation interval ID = 1 for SDCON.

Call Statement and Input

```
           H  INCH  X  INCX  Y  INCY  NH  NX IY0  NY  ID
           |   |   |   |   |   |   |   |   |   |   |
CALL SDCON( H , 1  , X , 1  , Y , 1  , 4 , 8 , 0 , 6 , 2 )

H       =  (1.0, 2.0, 3.0, 4.0)
X       =  (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
```

Output

```
Y       =  (11.0, 70.0, 130.0, 150.0, 151.0, 72.0)
```

Example 2: This example shows how to compute a correlation of a sequence in H with a sequence in X, where both sequences are ramp functions. It shows how a decimated output can be obtained, using the same input as "Example 6" on page 840 for SCORD and using a decimation interval ID = 2.

Note: For further examples of use, see the examples for SCORD on page 840. Because SCORD and SDCOR are functionally the same, their results are the same as long as the decimation interval ID = 1 for SDCOR.

Call Statement and Input

```
           H  INCH  X  INCX  Y  INCY  NH  NX  IY0  NY  ID
           |   |   |   |   |   |   |   |   |   |   |
CALL SDCOR( H , 1  , X , 1  , Y , 1  , 4 , 8 , -3 , 6 , 2 )

H       =  (1.0, 2.0, 3.0, 4.0)
X       =  (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
```

Output

```
Y       =  (44.0, 110.0, 140.0, 160.0, 104.0, 18.0)
```

Example 3: This example shows how to compute the same function as computed in "Example 1" on page 839 for SCOND. The input sequences and arguments are the same as that example, except a decimation interval ID = 1 is specified here for SDCON.

Call Statement and Input

```
           H  INCH  X  INCX  Y  INCY  NH  NX IY0  NY   ID
           |   |   |   |   |   |   |   |   |   |    |
CALL SDCON( H , 1  , X , 1  , Y , 1  , 4 , 8 , 0 , 11 , 1 )

H       =  (1.0, 2.0, 3.0, 4.0)
X       =  (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
```

Output

```
Y       =  (11.0, 34.0, 70.0, 120.0, 130.0, 140.0, 150.0, 160.0,
            151.0, 122.0, 72.0)
```

SACOR—Autocorrelation of One or More Sequences

This subroutine computes the autocorrelations of one or more sequences using a direct method. The input and output sequences contain short-precision real numbers.

Note: This subroutine is considered obsolete. It is provided in ESSL only for compatibility with earlier releases. You should use SCORD, SDCOR, SCORF and SACORF instead, because they provide **better performance**. For further details, see reference [4].

Syntax

Fortran	CALL SACOR (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *nx*, *m*, *ny*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	sacor (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *nx*, *m*, *ny*, *aux1*, *naux1*, *aux2*, *naux2*);
PL/I	CALL SACOR (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *nx*, *m*, *ny*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init
is a flag, where:

If *init* ≠ 0, no computation is performed, error checking is performed, and the subroutine exits back to the calling program.

If *init* = 0, the autocorrelations of the sequence in *x* are computed.

Specified as: a fullword integer. It can have any value.

x
is the array X, consisting of *m* input sequences of length N_x, to be autocorrelated. Specified as: an array of (at least) length $1+(N_x-1)inc1x+(m-1)inc2x$, containing short-precision real numbers.

inc1x
is the stride between the elements within each sequence in array X. Specified as: a fullword integer; *inc1x* > 0.

inc2x
is the stride between the first elements of the sequences in array X. Specified as: a fullword integer; *inc2x* > 0.

y
See On Return.

inc1y
is the stride between the elements within each sequence in output array Y. Specified as: a fullword integer; *inc1y* > 0.

inc2y
is the stride between the first elements of each sequence in output array Y. Specified as: a fullword integer; *inc2y* > 0.

nx
is the number of elements, N_x, in each sequence in array X. Specified as: a fullword integer; N_x > 0.

m
is the number of sequences in array X to be correlated. Specified as: a fullword integer; *m* > 0.

ny
is the number of elements, N_y, in each sequence in array Y. Specified as: a fullword integer; N_y > 0.

aux1
is no longer used in the computation, but must still be specified as a dummy argument (for migration purposes from Version 1 of ESSL). It can have any value.

naux1
is no longer used in the computation, but must still be specified as a dummy argument (for migration purposes from Version 1 of ESSL). It can have any value.

aux2
is no longer used in the computation, but must still be specified as a dummy argument (for migration purposes from Version 1 of ESSL). It can have any value.

naux2
is no longer used in the computation, but must still be specified as a dummy argument (for migration purposes from Version 1 of ESSL). It can have any value.

On Return

y
is array Y, consisting of m output sequences of length N_y that are the autocorrelation functions of the sequences in array X. Returned as: an array of (at least) length 1 + (N_y-1)*inc1y* + $(m-1)$*inc2y*, containing short-precision real numbers.

Notes

1. Output should not overwrite input; that is, input arrays X and H must have no common elements with output array Y. Otherwise, results are unpredictable. See "Concepts" on page 55.
2. Auxiliary storage is not needed, but the arguments *aux1*, *naux1*, *aux2*, and *naux2* must still be specified. You can assign any values to these arguments.

Function: The autocorrelations of the sequences in array X are expressed as follows:

$$y_{ki} = \sum_{j=0}^{N_x-1-k} x_{ji}x_{j+k,i}$$

for:

$k = 0, 1, ..., N_y-1$
$i = 1, 2, ..., m$

where:

y_{ki} are elements of the m sequences of length N_y in array Y.
x_{ji} and $x_{j+k,i}$ are elements of the m sequences of length N_x in array X.

See references [17] and [78].

Only one invocation of this subroutine is needed:

1. You do not need to invoke the subroutine with *init* $\neq 0$. If you do, however, the subroutine performs error checking, exits back to the calling program, and no computation is performed.
2. With *init* = 0, the subroutine performs the calculation of the convolutions or correlations.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. *nx*, *ny*, or *m* ≤ 0
2. *inc1x*, *inc2x*, *inc1y*, or *inc2y* ≤ 0 (or incompatible)

Example 1: This example shows how to compute an autocorrelation for three short sequences in array X, where the input sequence length NX is equal to the output sequence length NY. This gives all nonzero autocorrelation values.

The arrays are declared as follows:

```
REAL*4   X(0:49999), Y(0:49999)
REAL*8   AUX1, AUX2
```

Call Statement and Input

```
          INIT  X INC1X INC2X Y INC1Y INC2Y NX  M   NY  AUX1 NAUX1 AUX2   NAUX2
           |    |   |     |   |   |     |   |   |   |    |     |    |      |
CALL SACOR(INIT, X , 1  ,  7 , Y , 1  ,  7 , 7 , 3 , 7 , AUX1 , 0 , AUX2 ,   0)
```

```
INIT     =  0(for computation)
```

X contains the following three sequences:

```
1.0  2.0  3.0
2.0  1.0  2.0
3.0  2.0  1.0
4.0  3.0  2.0
4.0  4.0  3.0
3.0  4.0  4.0
2.0  3.0  4.0
```

Output: Y contains the following three sequences:

```
59.0  59.0  59.0
54.0  50.0  44.0
43.0  39.0  30.0
29.0  27.0  24.0
16.0  18.0  21.0
 7.0  11.0  20.0
 2.0   6.0  12.0
```

Example 2: This example shows how the output from Example 1 differs when the values for NY and INC2Y are 9 rather than 7. This shows that when NY is greater than NX, the output array is longer, and that part is filled with zeros.

Output: Y contains the following three sequences:

```
59.0  59.0  59.0
54.0  50.0  44.0
43.0  39.0  30.0
29.0  27.0  24.0
16.0  18.0  21.0
 7.0  11.0  20.0
 2.0   6.0  12.0
 0.0   0.0   0.0
 0.0   0.0   0.0
```

Example 3: This example shows how the output from Example 1 differs when the value for NY is 5 rather than 7. Also, the values for INC1X and INC1Y are 3, and the values for INC2X and INC2Y are 1 rather than 7. This shows that when NY is less than NX, the output array is shortened.

Output: Y contains the following three sequences:

```
59.0  59.0  59.0
54.0  50.0  44.0
43.0  39.0  30.0
29.0  27.0  24.0
16.0  18.0  21.0
```

SACORF—Autocorrelation of One or More Sequences Using the Mixed-Radix Fourier Method

This subroutine computes the autocorrelations of one or more sequences using the mixed-radix Fourier method. The input and output sequences contain short-precision real numbers.

Note: Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.

Syntax

Fortran	CALL SACORF (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *nx*, *m*, *ny*, *aux1*, *naux1*, *aux2*, *naux2*)
C and C++	sacorf (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *nx*, *m*, *ny*, *aux1*, *naux1*, *aux2*, *naux2*);
PL/I	CALL SACORF (*init*, *x*, *inc1x*, *inc2x*, *y*, *inc1y*, *inc2y*, *nx*, *m*, *ny*, *aux1*, *naux1*, *aux2*, *naux2*);

On Entry

init
is a flag, where:

If *init* ≠ 0, trigonometric functions and other parameters, depending on arguments other than *x*, are computed and saved in *aux1*. The contents of *x* and *y* are not used or changed.

If *init* = 0, the autocorrelations of the sequence in *x* are computed. The only arguments that may change after initialization are *x*, *y*, and *aux2*. All scalar arguments must be the same as when the subroutine was called for initialization with *init* ≠ 0.

Specified as: a fullword integer. It can have any value.

x
is the array X, consisting of *m* input sequences of length N_x, to be autocorrelated. Specified as: an array of (at least) length $1+(N_x-1)inc1x+(m-1)inc2x$, containing short-precision real numbers.

inc1x
is the stride between the elements within each sequence in array X. Specified as: a fullword integer; *inc1x* > 0.

inc2x
is the stride between the first elements of the sequences in array X. Specified as: a fullword integer; *inc2x* > 0.

y
See On Return.

inc1y
is the stride between the elements within each sequence in output array Y. Specified as: a fullword integer; *inc1y* > 0.

inc2y
is the stride between the first elements of each sequence in output array Y. Specified as: a fullword integer; *inc2y* > 0.

nx
is the number of elements, N_x, in each sequence in array X. Specified as: a fullword integer; N_x > 0.

m
is the number of sequences in array X to be correlated. Specified as: a fullword integer; $m > 0$.

ny
is the number of elements, N_y, in each sequence in array Y. Specified as: a fullword integer; $N_y > 0$.

aux1
is the working storage for this subroutine, where:

If *init* ≠ 0, the working storage is computed.

If *init* = 0, the working storage is used in the computation of the autocorrelations.

Specified as: an area of storage, containing *naux1* long-precision real numbers.

naux1
is the number of doublewords in the working storage specified in *aux1*. Specified as: a fullword integer; *naux1* > 21 and *naux1* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For values between 21 and the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 31.

aux2
has the following meaning:

If *naux2* = 0 and error 2015 is unrecoverable, *aux2* is ignored.

Otherwise, it is the working storage used by this subroutine, which is available for use by the calling program between calls to this subroutine.

Specified as: an area of storage, containing *naux2* long-precision real numbers. On output, the contents are overwritten.

naux2
is the number of doublewords in the working storage specified in *aux2*. Specified as: a fullword integer, where:

If *naux2* = 0 and error 2015 is unrecoverable, SACORF dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux2* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For all other values specified less than the minimum value, you have the option of having the minimum value returned in this argument. For details, see "Using Auxiliary Storage in ESSL" on page 31.

On Return

y
has the following meaning, where:

If *init* ≠ 0, this argument is not used, and its contents remain unchanged.

If *init* = 0, this is array Y, consisting of *m* output sequences of length N_y that are the autocorrelation functions of the sequences in array X.

Returned as: an array of (at least) length $1+(N_y-1)inc1y+(m-1)inc2y$, containing short-precision real numbers.

aux1
is the working storage for this subroutine, where:

If *init* ≠ 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If *init* = 0, its contents are unchanged.

Returned as: the contents are not relevant.

Notes

1. *aux1* should **not** be used by the calling program between calls to this subroutine with *init* ≠ 0 and *init* = 0. However, it can be reused after intervening calls to this subroutine with different arguments.
2. If you specify the same array for X and Y, then *inc1x* and *inc1y* must be equal and *inc2x* and *inc2y* must be equal. In this case, output overwrites input.
3. If you specify different arrays for X and Y, they must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.
4. If *ny* is such that output outside the basic range is needed, the subroutine stores zeros. This range is: $0 \le k \le nx-1$.

Formula for Calculating the Length of the Fourier Transform: Before calculating the necessary sizes of *naux1* and *naux2*, you must determine the length *n* of the Fourier transform. To do this, you use the values of the arguments *nx* and *ny*, inserted into the following formula, to get a value for the variable *nf*. After calculating *nf*, reference the formula or table of allowable values of *n* in "Acceptable Lengths for the Transforms" on page 741, selecting the value equal to or greater than *nf*. Following is the formula for determining *nf*:

$nf = \min(ny, nx)+nx+1$

Processor-Independent Formulas for NAUX1 and NAUX2: The required values of *naux1* and *naux2* depend on the value determined for *n* in " Formula for Calculating the Length of the Fourier Transform" and the argument *m*.

NAUX1 Formulas

If $n \le 16384$, use $naux1 = 55000$.
If $n > 16384$, use $naux1 = 40000+1.89n$.

NAUX2 Formulas

If $n \le 16384$, use $naux2 = 50000$.
If $n > 16384$, use $naux2 = 40000+1.64n$.

Function: The autocorrelations of the sequences in array X are expressed as follows:

$$y_{ki} = \sum_{j=0}^{N_x-1-k} x_{ji}x_{j+k,i}$$

for:

$k = 0, 1, ..., N_y-1$

$i = 1, 2, ..., m$

where:

y_{ki} are elements of the m sequences of length N_y in array Y.
x_{ji} and $x_{j+k,i}$ are elements of the m sequences of length N_x in array X.

This subroutine uses a Fourier transform method with a mixed-radix capability. This provides maximum performance for your application. The length of the transform, *n*, that you must calculate to determine the correct sizes for *naux1* and *naux2* is the same length used by the Fourier transform subroutines called by this subroutine. See references [17] and [78].

Two invocations of this subroutine are necessary:

1. With *init* ≠ 0, the subroutine tests and initializes arguments of the program, setting up the *aux1* working storage.
2. With *init* = 0, the subroutine checks that the initialization arguments in the *aux1* working storage correspond to the present arguments, and if so, performs the calculation of the autocorrelations.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux2* = 0, and unable to allocate work area.

Computational Errors: None

Input-Argument Errors

1. *nx*, *ny*, or $m \leq 0$
2. *inc1x*, *inc2x*, *inc1y*, or *inc2y* ≤ 0 (or incompatible)
3. The resulting correlation is too long.
4. The subroutine has not been initialized with the present arguments.
5. *naux1* ≤ 21
6. *naux1* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
7. Error 2015 is recoverable or *naux2*≠0, and *naux2* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example 1: This example shows how to compute an autocorrelation for three short sequences in array X, where the input sequence length NX is equal to the output sequence length NY. This gives all nonzero autocorrelation values. The arrays are declared as follows:

```
REAL*4   X(0:49999), Y(0:49999)
REAL*8   AUX1(30788), AUX2(17105)
```

First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Call Statement and Input

```
           INIT  X INC1X INC2X Y INC1Y INC2Y NX  M   NY  AUX1  NAUX1 AUX2  NAUX2
            |    |   |     |   |   |     |   |   |   |    |      |    |      |
CALL SACORF(INIT, X , 1  ,  7 , Y , 1  ,  7 , 7 , 3 , 7 , AUX1, 2959, AUX2, 4354)
```

```
INIT     =  1(for initialization)
INIT     =  0(for computation)
```

X contains the following three sequences:

```
1.0  2.0  3.0
2.0  1.0  2.0
3.0  2.0  1.0
4.0  3.0  2.0
4.0  4.0  3.0
3.0  4.0  4.0
2.0  3.0  4.0
```

Output: Y contains the following three sequences:

```
59.0  59.0  59.0
54.0  50.0  44.0
43.0  39.0  30.0
29.0  27.0  24.0
16.0  18.0  21.0
 7.0  11.0  20.0
 2.0   6.0  12.0
```

Example 2: This example shows how the output from Example 1 differs when the value for NY and INC2Y are 9 rather than 7. This shows that when NY is greater than NX, the output array is longer and that part is filled with zeros.

Output: Y contains the following three sequences:

```
59.0  59.0  59.0
54.0  50.0  44.0
43.0  39.0  30.0
29.0  27.0  24.0
16.0  18.0  21.0
 7.0  11.0  20.0
 2.0   6.0  12.0
 0.0   0.0   0.0
 0.0   0.0   0.0
```

Example 3: This example shows how the output from Example 1 differs when the value for NY is 5 rather than 7. Also, the values for INC1X and INC1Y are 3 rather than 1, and the values for INC2X and INC2Y are 1 rather than 7. This shows that when NY is less than NX, the output array is shortened.

Output: Y contains the following three sequences:

```
59.0  59.0  59.0
54.0  50.0  44.0
43.0  39.0  30.0
29.0  27.0  24.0
16.0  18.0  21.0
```

Related-Computation Subroutines

This section contains the related-computation subroutine descriptions.

SPOLY and DPOLY—Polynomial Evaluation

These subroutines evaluate a polynomial of degree k, using coefficient vector $\boldsymbol{u}$, input vector $\boldsymbol{x}$, and output vector $\boldsymbol{y}$:

$$y_i = u_0 + u_1 x_i + u_2 x_i^2 + \ldots + u_k x_i^k \qquad \text{for } i = 1,2,\ldots,n$$

where u_k, x_i, and y_i are elements of $\boldsymbol{u}$, $\boldsymbol{x}$, and $\boldsymbol{y}$, respectively.

Table 138. Data Types

u*, *x*, *y	**Subroutine**
Short-precision real	SPOLY
Long-precision real	DPOLY

Syntax

Fortran	CALL SPOLY | DPOLY (*u*, *incu*, *k*, *x*, *incx*, *y*, *incy*, *n*)
C and C++	spoly | dpoly (*u*, *incu*, *k*, *x*, *incx*, *y*, *incy*, *n*);
PL/I	CALL SPOLY | DPOLY (*u*, *incu*, *k*, *x*, *incx*, *y*, *incy*, *n*);

On Entry

u
is the coefficient vector $\boldsymbol{u}$ of length k+1. It contains elements u_0, u_1, u_0, u_1, u_2, ..., u_k, which are stored in this order. Specified as: a one-dimensional array of (at least) length $1+k|incu|$, containing numbers of the data type indicated in Table 138.

incu
is the stride for vector $\boldsymbol{u}$. Specified as: a fullword integer. It can have any value.

k
is the degree k of the polynomial. Specified as: a fullword integer; $k \geq 0$.

x
is the input vector $\boldsymbol{x}$ of length n. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 138.

incx
is the stride for vector $\boldsymbol{x}$. Specified as: a fullword integer. It can have any value.

y
See On Return.

incy
is the stride for the output vector $\boldsymbol{y}$. Specified as: a fullword integer. It can have any value.

n
is the number of elements in input vector $\boldsymbol{x}$ and the number of resulting elements in output vector $\boldsymbol{y}$. Specified as: a fullword integer; $n \geq 0$.

On Return

y
is the output vector ***y*** of length *n*, containing the results of the polynomial evaluation. Returned as: a one-dimensional array of (at least) length 1+(*n*−1)|*incy*|, containing numbers of the data type indicated in Table 138 on page 867.

Note: Vectors ***u***, ***x***, and ***y*** must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.

Function: The evaluation of the polynomial:

$$y_i = u_0 + u_1 x_i + u_2 x_i^2 + \ldots + u_k x_i^k \qquad \text{for } i = 1,2,\ldots,n$$

is expressed as follows:

$$y_i = u_0 + x_i\,(u_1 + x_i\,(u_2 + \ldots + x_i\,(u_{k-1} + x_i u_k)\ldots)) \qquad \text{for } i = 1, 2, \ldots, n$$

See reference [75] for Horner's Rule. If *n* is 0, no computation is performed. For SPOLY, intermediate results are accumulated in long precision.

SPOLY provides the same function as the IBM 3838 function POLY, with restrictions removed. DPOLY provides a long-precision computation that is not included in the IBM 3838 functions. See the *IBM 3838 Array Processor Functional Characteristics* manual.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. $k < 0$
2. $n < 0$

Example 1: This example shows a polynomial evaluation with the degree, `K`, equal to 0.

Call Statement and Input

```
            U   INCU   K   X   INCX   Y  INCY  N
            |    |     |   |    |     |   |    |
CALL SPOLY( U , INCU , 0 , X , INCX , Y , 1  , 3 )

U        =  (4.0)
INCU     =(not relevant)
X        =(not relevant)
INCX     =(not relevant)
```

Output

```
Y        =  (4.0, 4.0, 4.0)
```

Example 2: This example shows a polynomial evaluation, using a negative stride `INCU` for vector ***u***. For ***u***, processing begins at element `U(4)` which is 1.0.

Call Statement and Input

```
            U  INCU  K   X  INCX  Y  INCY  N
            |    |   |   |   |    |   |    |
CALL SPOLY( U , -1 , 3 , X , 1  , Y , 1  , 3 )

U        =  (4.0, 3.0, 2.0, 1.0)
X        =  (2.0, 1.0, -3.0)
```

Output

```
Y        =  (49.0, 10.0, -86.0)
```

Example 3: This example shows a polynomial evaluation, using a stride INCX of 0 for input vector ***x***.

Call Statement and Input

```
            U  INCU  K   X  INCX  Y  INCY  N
            |   |    |   |   |    |   |    |
CALL SPOLY( U , 1  , 3 , X , 0  , Y , 1  , 3 )

U        =  (4.0, 3.0, 2.0, 1.0)
X        =  (2.0, . , . )
```

Output

```
Y        =  (26.0, 26.0, 26.0)
```

Example 4: This example shows a polynomial evaluation, using a stride INCX greater than 1 for input vector ***x***, and a negative stride INCY for output vector ***y***. For ***y***, results are stored beginning at element Y(5).

Call Statement and Input

```
            U  INCU  K   X  INCX  Y  INCY  N
            |   |    |   |   |    |    |   |
CALL SPOLY( U , 1  , 3 , X , 2  , Y , -2 , 3 )

U        =  (4.0, 3.0, 2.0, 1.0)
X        =  (2.0, . , -3.0, . , 1.0)
```

Output

```
Y        =  (10.0, . , -14.0, . , 26.0)
```

SIZC and DIZC—I-th Zero Crossing

These subroutines find the position of the *i*-th zero crossing in vector ***x***. This is the *i*-th transition between positive and negative or negative and positive, where 0 is considered a positive value. It returns the position of the element in vector ***x*** where the *i*-th zero crossing is detected. The direction of the scan is either from the first element to the last or from the last element to the first, depending on the value you specify for the scan direction argument.

Table 139. Data Types

x	**Subroutine**
Short-precision real	SIZC
Long-precision real	DIZC

Syntax

Fortran	CALL SIZC | DIZC (*x*, *idrx*, *n*, *i*, *ky*)
C and C++	sizc | dizc (*x*, *idrx*, *n*, *i*, *ky*);
PL/I	CALL SIZC | DIZC (*x*, *idrx*, *n*, *i*, *ky*);

On Entry

x
is the target vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 139.

idrx
indicates the scan direction. If it is positive or 0, ***x*** is scanned from the first element to the last (1, *n*). If it is negative, ***x*** is scanned from the last element to the first (*n*, 1). Specified as: a fullword integer. It can have any value.

n
is the number of elements in vector ***x***. Specified as: a fullword integer; $n > 1$.

i
is the number of the zero crossing to be identified. Specified as: a fullword integer; $i > 0$.

ky
See On Return.

On Return

ky
is the integer vector ***ky*** of length 2, containing elements ky_1 and ky_2, where:

If the *i*-th zero crossing is found:

- $ky_1 = j$, where *j* is the position of the element x_j at the point that the *i*-th zero crossing is found. The position is always relative to the beginning of the vector regardless of the scan direction.
- $ky_2 = i$

If the *i*-th zero crossing is not found:

- $ky_1 = 0$

- ky_2 = the total number of zero crossings encountered in the scan.

Returned as: an array of (at least) length 2, containing fullword integers.

Note: The *aux* and *naux* arguments, required in some earlier releases of ESSL, are no longer required by these subroutines. If your program still includes them, you do not have to change your program; it continues to run normally. It ignores these arguments. However, if you did any program checking for error code 2015, you may want to remove it, because this error no longer occurs. (You must not code these arguments in your C program.)

Function: The *i*-th zero crossing in vector ***x*** is found by scanning vector ***x*** for *i* occurrences of TRUE for the following logical expressions. A zero crossing is defined here as a crossing either from a positive value to a negative value or from a negative value to a positive value, where 0 is considered a positive value. If the *i*-th zero crossing is found, the value of *j* at that point is returned in ky_1 as the position of the *i*-th zero crossing, and *i* is returned in ky_2.

If $idrx \geq 0$:

$$\text{TRUE} = (x_{j-1} < 0 \text{ and } x_j \geq 0) \text{ or } (x_{j-1} \geq 0 \text{ and } x_j < 0) \text{ for } j = 2, n$$

If $idrx < 0$:

$$\text{TRUE} = (x_{j+1} < 0 \text{ and } x_j \geq 0) \text{ or } (x_{j+1} \geq 0 \text{ and } x_j < 0) \text{ for } j = n-1, 1$$

If the position of the *i*-th zero crossing is not found, 0 is returned in y_1 and the number of zero crossings encountered in the scan is returned in y_2.

SIZC provides the same functions as the IBM 3838 functions NZCP and NZCN, with restrictions removed. It combines these functions into one ESSL subroutine. DIZC provides a long-precision computation that is not included in the IBM 3838 functions. See the *IBM 3838 Array Processor Functional Characteristics* manual.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. $n \leq 1$
2. $i \leq 0$

Example 1: This example shows a scan of a vector ***x*** from the first element to the last. It is looking for the fifth zero crossing, which is encountered at position 9.

Call Statement and Input

```
           X  IDRX  N    I   KY
           |   |    |    |   |
CALL SIZC( X , 1  , 12 , 5 , KY )

X       =  (2.0, -1.0, -3.0, 3.0, 0.0, 8.0, -2.0, 0.0, -5.0, -3.0,
            2.0, -9.0)
```

Output

```
KY      =  (9, 5)
```

Example 2: This example shows a scan of a vector ***x*** from the last element to the first. It is looking for the seventh zero crossing, which is encountered at position 3. Because `IDRX` is negative, `X` is scanned from the last element, `X(12)`, to the first element, `X(1)`.

Call Statement and Input

```
           X  IDRX  N    I   KY
           |    |   |    |   |
CALL SIZC( X , -1 , 12 , 7 , KY )

X       =  (2.0, -1.0, 3.0, -3.0, 0.0, -8.0, -2.0, 0.0, -5.0, -3.0,
            2.0, -9.0)
```

Output

```
KY      =  (3, 7)
```

Example 3: This example shows a scan of a vector ***x*** when the *i*-th zero crossing is not found. It encounters seven zero crossings and returns this value in `KY(2)`.

Call Statement and Input

```
           X  IDRX  N    I    KY
           |   |    |    |    |
CALL SIZC( X , 1  , 12 , 10 , KY )

X       =  (2.0, -1.0, -3.0, 3.0, 0.0, 8.0, -2.0, 0.0, -5.0, -3.0,
            2.0, -9.0)
```

Output

```
KY      =  (0, 7)
```

STREC and DTREC—Time-Varying Recursive Filter

These subroutines implement the first-order time-varying recursive equation, using initial value *s*, target vectors ***u*** and ***x***, and output vector ***y***.

Table 140. Data Types

s*, *u*, *x*, *y	**Subroutine**
Short-precision real	STREC
Long-precision real	DTREC

Syntax

Fortran	CALL STREC | DTREC (*s*, *u*, *incu*, *x*, *incx*, *y*, *incy*, *n* , *iopt*)
C and C++	strec | dtrec (*s*, *u*, *incu*, *x*, *incx*, *y*, *incy*, *n*, *iopt*);
PL/I	CALL STREC | DTREC (*s*, *u*, *incu*, *x*, *incx*, *y*, *incy*, *n* , *iopt*);

On Entry

s
is the scalar *s* used in the initial computation for y_1. Specified as: a number of the data type indicated in Table 140.

u
is the target vector ***u*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incu|$, containing numbers of the data type indicated in Table 140.

incu
is the stride for target vector ***u***. Specified as: a fullword integer. It can have any value.

x
is the target vector ***x*** of length *n*. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 140.

incx
is the stride for target vector ***x.*** Specified as: a fullword integer. It can have any value.

y
See On Return.

incy
is the stride for output vector ***y.*** Specified as: a fullword integer; $incy > 0$ or $incy < 0$.

n
is the number of elements in vectors ***u*** and ***x*** and the number of resulting elements in output vector ***y***. Specified as: a fullword integer; $n \geq 0$.

iopt
this argument has no effect on the performance of the computation, but still must be specified as: a fullword integer; $iopt = 0$ or 1.

On Return

y
is the vector ***y*** of length *n*, containing the results of the implementation of the first-order time-varying recursive equation. Returned as: a one-dimensional array of

(at least) length $1+(n-1)|incy|$, containing numbers of the data type indicated in Table 140.

Note: Vectors ***u***, ***x***, and ***y*** must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 55.

Function: The first-order time-varying recursive equation is expressed as follows:

$y_1 = s+u_1x_1$
$y_2 = u_2y_1+u_1x_2$
.
.
.
$y_i = u_iy_{i-1}+u_1x_i$ for $i = 3, 4, ..., n$

STREC provides the same function as the IBM 3838 function REC, with restrictions removed. DTREC provides a long-precision computation that is not included in the IBM 3838 functions. See the *IBM 3838 Array Processor Functional Characteristics* manual.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. $incy = 0$
2. $n < 0$
3. $iopt \neq 0$ or 1

Example 1: This example shows all strides `INCU`, `INCX`, and `INCY` equal to 1 for vectors ***u***, ***x***, and ***y***, respectively.

Call Statement and Input

```
             S    U  INCU  X  INCX  Y  INCY  N  IOPT
             |    |   |    |   |    |   |    |   |
CALL STREC( 1.0 , U , 1  , X , 1  , Y , 1  , 8 , 0 )

U        =  (1.0, 2.0, 3.0, 3.0, 2.0, 1.0, 1.0, 2.0)
X        =  (3.0, 2.0, 1.0, 1.0, 2.0, 3.0, 3.0, 2.0)
```

Output

```
Y        =  (4.0, 10.0, 31.0, 94.0, 190.0, 193.0, 196.0, 394.0)
```

Example 2: This example shows a stride, INCU, that is greater than 1 for vector ***u***. The strides INCX and INCY for vectors ***x*** and ***y***, respectively, are 1.

Call Statement and Input

```
             S    U  INCU  X  INCX  Y  INCY  N  IOPT
             |    |   |    |   |    |   |    |   |
CALL STREC( 1.0 , U , 2  , X , 1  , Y , 1  , 4 , 0 )

U        =  (1.0, . , 3.0, . , 2.0, . , 1.0, . )
X        =  (3.0, 2.0, 1.0, 1.0, 2.0, 3.0, 3.0, 2.0)
```

Output

```
Y        =  (4.0, 14.0, 29.0, 30.0)
```

Example 3: This example shows a stride, INCU, of 1 for vector ***u***, a stride, INCX, that is greater than 1 for vector ***x***, and a negative stride, INCY, for vector ***y***. For ***y***, results are stored beginning at element Y(4).

Call Statement and Input

```
             S    U  INCU  X  INCX  Y  INCY  N  IOPT
             |    |   |    |   |    |   |    |   |
CALL STREC( 1.0 , U , 1  , X , 2  , Y , -1 , 4 , 1 )

U        =  (1.0, 2.0, 3.0, 3.0, 2.0, 1.0, 1.0, 2.0)
X        =  (3.0, . , 1.0, . , 2.0, . , 3.0)
```

Output

```
Y        =  (90.0, 29.0, 9.0, 4.0)
```

SQINT and DQINT—Quadratic Interpolation

These subroutines perform a quadratic interpolation at specified points in the vector ***x***, using initial linear displacement in the samples *s*, sample interval *g*, output scaling parameter Ω, and sample reflection times in vector ***t***. The result is returned in vector ***y***.

Table 141. Data Types

x, *s*, *g*, Ω, ***t***, ***y***	**Subroutine**
Short-precision real	SQINT
Long-precision real	DQINT

Syntax

Fortran	CALL SQINT | DQINT (*s*, *g*, *omega*, *x*, *incx*, *n*, *t*, *inct*, *y*, *incy*, *m*)
C and C++	sqint | dqint (*s*, *g*, *omega*, *x*, *incx*, *n*, *t*, *inct*, *y*, *incy*, *m*);
PL/I	CALL SQINT | DQINT (*s*, *g*, *omega*, *x*, *incx*, *n*, *t*, *inct*, *y*, *incy*, *m*);

On Entry

s
is the scalar *s*, containing the initial linear displacement in samples. Specified as: a number of the data type indicated in Table 141.

g
is the scalar *g*, containing the sample interval. Specified as: a number of the data type indicated in Table 141; $g > 0.0$.

omega
is the output scaling parameter Ω. Specified as: a number of the data type indicated in Table 141.

x
is the vector ***x*** of length *n*, containing the trace data. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 141.

incx
is the stride for vector ***x***. Specified as: a fullword integer; $incx > 0$ or $incx < 0$.

n
is the number of elements in vector ***x***. Specified as: a fullword integer; $n \geq 3$.

t
is the vector ***t*** of length *m*, containing the sample reflection times to be processed. Specified as: a one-dimensional array of (at least) length $1+(m-1)|inct|$, containing numbers of the data type indicated in Table 141.

inct
is the stride for vector ***t***. Specified as: a fullword integer; $inct > 0$ or $inct < 0$.

y
See On Return.

incy
is the stride for output vector ***y***. Specified as: a fullword integer; $incy > 0$ or $incy < 0$.

m
is the number of elements in vector ***t*** and the number of elements in output vector ***y***. Specified as: a fullword integer; $m \geq 0$.

On Return

y
is the vector ***y*** of length *m*, containing the results of the quadratic interpolation. Returned as: a one-dimensional array of (at least) length 1+(*m*−1)|*incy*|, containing numbers of the data type indicated in Table 141 on page 876.

Function: The quadratic interpolation, which is expressed as follows:

$$y_i = \Omega \left(trace_{k_i} \left(f_i^2 - f_i \right) + 2\ trace_{k_i+1} \left(1 - f_i^2 \right) + trace_{k_i+2} \left(f_i^2 + f_i \right) \right)$$

for $i = 1, 2, ..., m$

uses the following values:

x is the vector containing the specified points.
s is the initial linear displacement in the samples.
g is a sample interval.
Ω is the output scaling parameter.
t is the vector containing the sample reflection times.

and where ***trace***, ***k***, ***f***, and ***w*** are four working vectors, and *so* is a working scalar defined as:

$trace_1 = 3x_1 - 3x_2 + x_3$
$trace_{i+1} = x_i$ for $i = 1, 2, ..., n$
$so = s + 2.0$
$w_i = so + t_i / g$ for $i = 1, 2, ..., m$
f_i = fraction part of w_i
$k_i + 1$ = integer part of w_i

Note: Allowing k_i+1 to have a value of 2 results in performance degradation. If possible, avoid specifying a point at which this occurs.

If *n* or *m* is 0, no computation is performed.

SQINT provides the same function as the IBM 3838 function INT, with restrictions removed. DQINT provides a long-precision computation that is not included in the IBM 3838 functions. See the *IBM 3838 Array Processor Functional Characteristics* manual.

Error Conditions

Computational Errors: The condition ($k_i+1 > n$) or ($k_i+1 \leq 2$) has occurred, where *n* is the number of elements in vector ***x***. See " Function" for how to calculate k_i.

- The lower range *l* and the upper range *j* of the vector are identified in the computational error message.
- The return code is set to 1.
- The ranges *l* and *j* of the vector can be determined at run time by using the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2100 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see "What Can You Do about ESSL Computational Errors?" on page 48.

Input-Argument Errors

1. $n < 3$
2. $m < 0$
3. $g \leq 0$
4. $incx = 0$
5. $inct = 0$
6. $incy = 0$

Example 1: This example shows a quadratic interpolation, using vectors with strides of 1.

Call Statement and Input

```
             S     G   OMEGA  X  INCX  N   T  INCT  Y  INCY  M
             |     |     |    |   |    |   |   |    |   |    |
CALL SQINT( 2.0 , 1.0 , 1.0 , X , 1  , 8 , T , 1  , Y , 1  , 4 )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0)
T        =  (1.5, 2.5, 3.5, 4.5)
```

Output

```
Y        =  (9.0, 11.0, 13.0, 15.0)
```

Example 2: This example shows a quadratic interpolation, using vectors with a positive stride of 1 and negative strides of −1.

Call Statement and Input

```
             S     G   OMEGA  X  INCX  N   T  INCT  Y  INCY  M
             |     |     |    |   |    |   |   |    |   |    |
CALL SQINT( 2.0 , 1.0 , 1.0 , X , -1 , 8 , T , -1 , Y , 1  , 4 )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0)
T        =  (1.5, 2.5, 3.5, 4.5)
```

Output

```
Y        =  (3.0, 5.0, 7.0, 9.0)
```

Example 3: This example shows a quadratic interpolation, using vectors with a positive stride greater than 1 and negative strides less than −1.

Call Statement and Input

```
             S     G   OMEGA  X  INCX  N   T  INCT  Y  INCY  M
             |     |     |    |   |    |   |   |    |   |    |
CALL SQINT( 2.0 , 1.0 , 1.0 , X , -2 , 8 , T , -1 , Y , 2  , 4 )

X        =  (1.0, . , 3.0, . , 5.0, . , 7.0, . , 9.0, . , 11.0, . ,
             13.0, . , 15.0)
T        =  (1.36, 2.36, 3.36, 4.36)
```

Output

```
Y        =  (4.56, . , 8.56, . , 12.56, . , 16.56)
```

Example 4: This example shows a quadratic interpolation, using vectors with positive strides and larger values for S and G than shown in the previous examples.

Call Statement and Input

```
             S      G   OMEGA  X  INCX  N   T  INCT  Y  INCY  M
             |      |     |    |   |    |   |   |    |   |    |
CALL SQINT( 3.0 , 10.0 , 1.0 , X , 1  , 8 , T , 2  , Y , 3  , 4 )

X        =  (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0)
T        =  (1.5, . , 2.5, . , 3.5, . , 4.5)
```

Output

```
Y        =  (8.3, . , . , 8.5, . , . , 8.7, . , . , 8.9)
```

SWLEV, DWLEV, CWLEV, and ZWLEV—Wiener-Levinson Filter Coefficients

These subroutines compute the coefficients of an n-point Wiener-Levinson filter, using vector ***x***, the trace for which the filter is to be designed, and vector ***u***, the right-hand side of the system, chosen to remove reverberations or sharpen the wavelet. The result is returned in vector ***y***.

Table 142. Data Types

x, ***u***, ***y***	*aux*	**Subroutine**
Short-precision real	Long-precision real	SWLEV
Long-precision real	Long-precision real	DWLEV
Short-precision complex	Long-precision complex	CWLEV
Long-precision complex	Long-precision complex	ZWLEV

Syntax

Fortran	CALL SWLEV | DWLEV | CWLEV | ZWLEV | (*x*, *incx*, *u*, *incu*, *y*, *incy*, *n*, *aux*, *naux*)
C and C++	swlev | dwlev | cwlev | zwlev (*x*, *incx*, *u*, *incu*, *y*, *incy*, *n*, *aux*, *naux*);
PL/I	CALL SWLEV | DWLEV | CWLEV | ZWLEV (*x*, *incx*, *u*, *incu*, *y*, *incy*, *n*, *aux*, *naux*);

On Entry

x
is the vector ***x*** of length *n*, containing the trace data for which the filter is to be designed.

For SWLEV and DWLEV, *x* represents the first row (or the first column) of a positive definite or negative definite symmetric Toeplitz matrix, which is the autocorrelation matrix for which the filter is designed.

For CWLEV and ZWLEV, *x* represents the first row of a positive definite or negative definite complex Hermitian Toeplitz matrix, which is the autocorrelation matrix for which the filter is designed.

Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 142.

incx
is the stride for vector ***x***. Specified as: a fullword integer; $incx > 0$.

u
is the vector ***u*** of length *n*, containing the right-hand side of the system to be solved. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incu|$, containing numbers of the data type indicated in Table 142.

incu
is the stride for vector ***u***. Specified as: a fullword integer. It can have any value.

y
See On Return.

incy
is the stride for vector ***y***. Specified as: a fullword integer; $incy > 0$ or $incy < 0$.

n
is the number of elements in vectors ***x***, ***u***, and ***y***. Specified as: a fullword integer; $n \geq 0$.

aux
has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is the storage work area used by these subroutines.

Specified as: an area of storage of length *naux*, containing numbers of the data type indicated in Table 142 on page 880.

naux
is the size of the work area specified by *aux*—that is, the number of elements in *aux*. Specified as: a fullword integer, where:

If *naux* = 0 and error 2015 is unrecoverable, SWLEV, DWLEV, CWLEV, and ZWLEV dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, $naux \geq 3n$.

You cannot use dynamic allocation if you need the information returned in AUX(1).

On Return

y
is the vector ***y*** of length *n*, containing the solution vector—that is, the coefficients of the n-point Wiener-Levinson filter. Returned as: a one-dimensional array of (at least) length $1+(n-1)|incy|$, containing numbers of the data type indicated in Table 142 on page 880.

aux
is the storage work area used by these subroutines, where if $naux \neq 0$:

If AUX(1) = 0.0, the input Toeplitz matrix is positive definite or negative definite.

If AUX(1) > 0.0, the input Toeplitz matrix is indefinite (that is, it is not positive definite and it is not negative definite). The value returned in AUX(1) is the order of the first submatrix of ***A*** that is indefinite. The subroutine continues processing. See reference [59] for information about under what circumstances your solution vector *y* would be valid.

All other values in *aux* are overwritten and are not significant.

Returned as: an area of storage of length *naux*, containing numbers of the data type indicated in Table 142 on page 880, where AUX(1)≥0.0.

Notes

1. For a description of a positive definite or negative definite symmetric Toeplitz matrix, see "Positive Definite or Negative Definite Symmetric Toeplitz Matrix" on page 71.
2. For a description of a positive definite or negative definite complex Hermitian Toeplitz matrix, see "Positive Definite or Negative Definite Complex Hermitian Toeplitz Matrix" on page 72.
3. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: The computation of the coefficients of an n-point Wiener-Levinson filter in vector ***y*** is expressed as solving the following system:

$$\boldsymbol{Ay} = \boldsymbol{u}$$

where:

- For SWLEV and DWLEV, matrix ***A*** is a real symmetric Toeplitz matrix whose first row (or first column) is represented by vector ***x***.

  For CWLEV and ZWLEV, matrix ***A*** is a complex Hermitian Toeplitz matrix whose first row is represented by vector ***x***.
- ***u*** is the vector specifying the right side of the system, chosen to remove reverberations or to sharpen the wavelet.
- ***y*** is the solution vector.

See reference [59], [27], and the *IBM 3838 Array Processor Functional Characteristics*.

If *n* is 0, no computation is performed. For SWLEV and CWLEV, intermediate results are accumulated in long precision.

SWLEV provides the same function as the IBM 3838 function WLEV, with restrictions removed. See the *IBM 3838 Array Processor Functional Characteristics* manual.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors: None

Input-Argument Errors

1. $n < 0$
2. $incx \leq 0$
3. $incy = 0$
4. Error 2015 is recoverable or $naux \neq 0$, and *naux* is too small—that is, less than the minimum required value specified in the syntax for this argument. Return code 1 is returned if error 2015 is recoverable.

Example 1: This example shows how to compute filter coefficients in vector ***y*** by solving the system $\boldsymbol{Ay} = \boldsymbol{u}$. Matrix ***A*** is:

```
┌                          ┐
│ 50.0  -8.0   7.0  -5.0 │
│ -8.0  50.0  -8.0   7.0 │
│  7.0  -8.0  50.0  -8.0 │
│ -5.0   7.0  -8.0  50.0 │
└                          ┘
```

This input Toeplitz matrix is positive definite, as indicated by the zero value in `AUX(1)` on output.

Call Statement and Input

```
            X  INCX  U  INCU  Y  INCY  N   AUX  NAUX
            |   |    |   |    |   |    |    |    |
CALL SWLEV( X , 1  , U , 1  , Y , 1  , 4 , AUX , 12 )

X        =  (50.0, -8.0, 7.0, -5.0)
U        =  (40.0, -10.0, 30.0, 20.0)
AUX      =(not relevant)
```

Output

```
Y        =  (0.7667, -0.0663, 0.5745, 0.5778)
AUX      =  (0.0, . , . , . , . , . , . , . , . , . , . , . , . , . , . )
```

Example 2: This example shows how to compute filter coefficients in vector ***y*** by solving the system ***Ay*** = ***u***. Matrix ***A*** is:

$$\begin{bmatrix} 10.0 & -8.0 & 7.0 & -5.0 \\ -8.0 & 10.0 & -8.0 & 7.0 \\ 7.0 & -8.0 & 10.0 & -8.0 \\ -5.0 & 7.0 & -8.0 & 10.0 \end{bmatrix}$$

This input Toeplitz matrix is not positive definite, as indicated by the zero value in `AUX(1)` on output.

Call Statement and Input

```
            X  INCX  U  INCU  Y  INCY  N   AUX  NAUX
            |   |    |   |    |   |    |    |    |
CALL SWLEV( X , 1  , U , 1  , Y , 1  , 4 , AUX , 12 )

X        =  (10.0, -8.0, 7.0, -5.0)
U        =  (40.0, -10.0, 30.0, 20.0)
AUX      =(not relevant)
```

Output

```
Y        =  (5.1111, 5.5555, 12.2222, 10.4444)
AUX      =  (0.0, . , . , . , . , . , . , . , . , . , . , . , . , . , . )
```

Example 3: This example shows a vector ***x*** with a stride greater than 1, a vector ***u*** with a negative stride, and a vector ***y*** with a stride of 1. It uses the same input Toeplitz matrix as in Example 2, which is not positive definite.

Call Statement and Input

```
            X  INCX  U  INCU  Y  INCY  N   AUX  NAUX
            |   |    |   |    |   |    |    |    |
CALL SWLEV( X , 2  , U , -2 , Y , 1  , 4 , AUX , 12 )

X        =  (10.0, . , -8.0, . , 7.0, . , -5.0)
U        =  (20.0, . , 30.0, . , -10.0, . , 40.0)
AUX      =(not relevant)
```

Output

```
Y       =  (5.1111, 5.5555, 12.2222, 10.4444)
AUX     =  (0.0, . , . , . , . , . , . , . , . , . , . , . , . , . , . , . , . )
```

Example 4: This example shows how to compute filter coefficients in vector ***y*** by solving the system ***Ay*** = ***u***. Matrix ***A*** is:

$$\begin{bmatrix} (10.0, 0.0) & (2.0, -3.0) & (-3.0, 1.0) & (1.0, 1.0) \\ (2.0, 3.0) & (10.0, 0.0) & (2.0, -3.0) & (-3.0, 1.0) \\ (-3.0, -1.0) & (2.0, 3.0) & (10.0, 0.0) & (2.0, -3.0) \\ (1.0, -1.0) & (-3.0, -1.0) & (2.0, 3.0) & (10.0, 0.0) \end{bmatrix}$$

This input complex Hermitian Toeplitz matrix is positive definite, as indicated by the zero value in `AUX(1)` on output.

Call Statement and Input

```
            X  INCX  U  INCU  Y  INCY  N   AUX  NAUX
            |   |    |   |    |   |    |    |    |
CALL ZWLEV( X , 1  , U , 1  , Y , 1  , 4 , AUX , 12 )

X       =  ((10.0, 0.0), (2.0, -3.0), (-3.0, 1.0), (1.0, 1.0))
U       =  ((8.0, 3.0), (21.0, -5.0), (67.0, -13.0), (72.0, 11.0))
AUX     =(not relevant)
```

Output

```
Y       =  ((1.0, 0.0), (3.0, 0.0), (5.0, 0.0), (7.0, 0.0))
AUX     =  ((0.0, 0.0), . , . , . , . , . , . , . , . , . , . , . , . , . , . , . )
```

Example 5: This example shows a vector ***x*** with a stride greater than 1, a vector ***u*** with a negative stride, and a vector ***y*** with a stride of 1. It uses the same input complex Hermitian Toeplitz matrix as in Example 4.

This input complex Hermitian Toeplitz matrix is positive definite, as indicated by the zero value in `AUX(1)` on output.

Call Statement and Input

```
            X  INCX  U  INCU   Y  INCY  N   AUX  NAUX
            |   |    |   |    |   |    |    |    |
CALL ZWLEV( X , 2  , U , -2  , Y , 1  , 4 , AUX , 12 )

X       =  ((10.0, 0.0), . , (2.0, -3.0), . , (-3.0, 1.0), . ,
            (1.0, 1.0))
U       =  ((72.0, 11.0), . , (67.0, -13.0), . , (21.0, -5.0), . ,
            (8.0, 3.0), . )
AUX     =(not relevant)
```

Output

```
Y       =  ((1.0, 0.0), (3.0, 0.0), (5.0, 0.0), (7.0, 0.0))
AUX     =  ((0.0, 0.0), . , . , . , . , . , . , . , . , . , . , . , . , . , . , . )
```

Chapter 13. Sorting and Searching

The sorting and searching subroutines are described in this chapter.

Overview of the Sorting and Searching Subroutines

The sorting and searching subroutines operate on three types of data: integer, short-precision real, and long-precision-real (Table 143). The sorting subroutines perform sorts with or without index designations. The searching subroutines perform either a binary or sequential search.

Table 143. List of Sorting and Searching Subroutines

Descriptive Name	Integer Subroutine	Short-Precision Subroutine	Long-Precision Subroutine	Page
Sort the Elements of a Sequence	ISORT	SSORT	DSORT	888
Sort the Elements of a Sequence and Note the Original Element Positions	ISORTX	SSORTX	DSORTX	890
Sort the Elements of a Sequence Using a Stable Sort and Note the Original Element Positions	ISORTS	SSORTS	DSORTS	893
Binary Search for Elements of a Sequence X in a Sorted Sequence Y	IBSRCH	SBSRCH	DBSRCH	896
Sequential Search for Elements of a Sequence X in the Sequence Y	ISSRCH	SSSRCH	DSSRCH	900

Use Considerations

It is important to understand the concept of stride for sequences when using these subroutines. For example, in the sort subroutines, a negative stride causes a sequence to be sorted into descending order in an array. In the search subroutines, a negative stride reverses the direction of the search. See “How Stride Is Used for Vectors” on page 58.

Performance and Accuracy Considerations

1. The binary search subroutines provide better performance than the sequential search subroutines because of the nature of the searching algorithms. However, the binary search subroutines require that, before the subroutine is called, the sequence to be searched is sorted into ascending order. Therefore, if your data is already sorted, a binary search subroutine is faster. On the other hand, if your data is in random order and the number of elements being searched for is small, a sequential search subroutine is faster than doing a sort and binary search.
2. When doing multiple invocations of the binary search subroutines, you get better overall performance from the searching algorithms by doing fewer invocations and specifying larger search element arrays for argument *x*.

© Copyright IBM Corp. 1997

3. If you do not need the results provided in array RC by these subroutine, you get better performance if you do not request it. That is, specify 0 for the *iopt* argument.

Sorting and Searching Subroutines

This section contains the sorting and searching subroutine descriptions.

ISORT, SSORT, and DSORT—Sort the Elements of a Sequence

These subroutines sort the elements of sequence ***x***.

Table 144. Data Types

x	**Subroutine**
Integer	ISORT
Short-precision real	SSORT
Long-precision real	DSORT

Syntax

Fortran	CALL ISORT | SSORT | DSORT (*x*, *incx*, *n*)
C and C++	isort | ssort | dsort (*x*, *incx*, *n*);
PL/I	CALL ISORT | SSORT | DSORT (*x*, *incx*, *n*);

On Entry

x
is the sequence ***x*** of length *n*, to be sorted. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 144.

incx
is the stride for both the input sequence ***x*** and the output sequence ***x***. If it is positive, elements are sorted into ascending order in the array, and if it is negative, elements are sorted into descending order in the array.

Specified as: a fullword integer. It can have any value.

n
is the number of elements in sequence ***x***. Specified as: a fullword integer; $n \geq 0$.

On Return

x
is the sequence ***x*** of length *n*, with its elements sorted into designated order in the array. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 144.

Function: The elements of input sequence ***x*** are sorted into ascending order, in place and using a partition sort. The elements of output sequence ***x*** can be expressed as follows:

$$x_1 \leq x_2 \leq x_3 \leq \ldots \leq x_n$$

By specifying a negative stride for sequence ***x***, the elements of sequence ***x*** are assumed to be reversed in the array, $(x_n, x_{n-1}, \ldots, x_1)$, thus producing a sort into descending order within the array. If *n* is 0 or 1 or if *incx* is 0, no sort is performed. See reference [69].

Error Conditions

Resource Errors: Unable to allocate internal work area.

Computational Errors: None

Input-Argument Errors: $n < 0$

Example 1: This example shows a sequence ***x*** with a positive stride.

Call Statement and Input

```
            X  INCX  N
            |   |    |
CALL ISORT( X , 2  , 5 )

X       =  (2, . , -1, . , 5, . , 4, . , -2)
```

Output

```
X       =  (-2, . , -1, . , 2, . , 4, . , 5)
```

Example 2: This example shows a sequence ***x*** with a negative stride.

Call Statement and Input

```
            X   INCX  N
            |    |    |
CALL ISORT( X , -1  , 5 )

X       =  (2, -1, 5, 4, -2)
```

Output

```
X       =  (5, 4, 2, -1, -2)
```

ISORTX, SSORTX, and DSORTX—Sort the Elements of a Sequence and Note the Original Element Positions

These subroutines sort the elements of sequence ***x***. The original positions of the elements in sequence ***x*** are returned in the indices array, INDX. Where equal elements occur in the input sequence, they do not necessarily remain in the same relative order in the output sequence.

Note: If you need a stable sort, you should use ISORTS, SSORTS, or DSORTS rather than these subroutines.

Table 145. Data Types

x	**Subroutine**
Integer	ISORTX
Short-precision real	SSORTX
Long-precision real	DSORTX

Syntax

Fortran	CALL ISORTX | SSORTX | DSORTX (*x*, *incx*, *n*, *indx*)
C and C++	isortx | ssortx | dsortx (*x*, *incx*, *n*, *indx*);
PL/I	CALL ISORTX | SSORTX | DSORTX (*x*, *incx*, *n*, *indx*);

On Entry

x
is the sequence ***x*** of length *n*, to be sorted. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$ elements, containing numbers of the data type indicated in Table 145.

incx
is the stride for both the input sequence ***x*** and the output sequence ***x***. If it is positive, elements are sorted into ascending order in the array, and if it is negative, elements are sorted into descending order in the array.

Specified as: a fullword integer. It can have any value.

n
is the number of elements in sequence ***x***. Specified as: a fullword integer; $n \geq 0$.

indx
See On Return.

On Return

x
is the sequence ***x*** of length *n*, with its elements sorted into designated order in the array. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 145.

indx
is the array, referred to as INDX, containing the *n* indices that indicate, for the elements in the sorted output sequence, the original positions of those elements in input sequence ***x***.

Note: It is important to remember that when you specify a negative stride, ESSL assumes that the order of the input and output sequence elements in the X array is reversed; however, the elements in INDX are not reversed. See "Function" on page 891.

Returned as: a one-dimensional array of length n, containing fullword integers; $1 \le$ (INDX elements) $\le n$.

Function: The elements of input sequence $\boldsymbol{x}$ are sorted into ascending order, in place and using a partition sort. The elements of output sequence $\boldsymbol{x}$ can be expressed as follows:

$$x_1 \le x_2 \le x_3 \le \ldots \le x_n$$

Where equal elements occur in the input sequence, they do not necessarily remain in the same relative order in the output sequence.

By specifying a negative stride for $\boldsymbol{x}$, the elements of input sequence $\boldsymbol{x}$ are assumed to be reversed in the array, $(x_n, x_{n-1}, \ldots, x_1)$, thus producing a sort into descending order within the array.

In addition, the INDX array contains the n indices that indicate, for the elements in the sorted output sequence, the original positions of those elements in input sequence $\boldsymbol{x}$. (These are not the positions in the array, but rather the positions in the sequence.) For each element x_j in the input sequence, becoming element xx_k in the output sequence, the elements in INDX are defined as follows:

INDX(k) $= j$ for $j = 1, n$ and $k = 1, n$
where $xx_k = x_j$

To understand INDX when you specify a negative stride, you should remember that both the input and output sequences, $\boldsymbol{x}$, are assumed to be in reverse order in array X, but INDX is not affected by stride. The sequence elements of $\boldsymbol{x}$ are assumed to be stored in your input array as follows:

X $= (x_n, x_{n-1}, \ldots, x_1)$

The sequence elements of $\boldsymbol{x}$ are stored in your output array by ESSL as follows:

X $= (xx_n, xx_{n-1}, \ldots, xx_1)$

where the elements xx_k are the elements x_j, sorted into descending order in X. As an example of how INDX is calculated, if $xx_1 = x_{n-1}$, then INDX(1) $= n-1$.

If n is 0, no computation is performed. See reference [69].

Error Conditions

Resource Errors: Unable to allocate internal work area.

Computational Errors: None

Input-Argument Errors: $n < 0$

Example 1: This example shows how to sort a sequence ***x*** into ascending order by specifying a positive stride.

Call Statement and Input

```
             X  INCX  N   INDX
             |   |    |    |
CALL ISORTX( X , 2  , 5 , INDX )

X        =  (2, . , -1, . , 5, . , 1, . , -2)
```

Output

```
X        =  (-2, . , -1, . , 1, . , 2, . , 5)
INDX     =  (5, 2, 4, 1, 3)
```

Example 2: This example shows how to sort a sequence ***x*** into descending order by specifying a negative stride. Therefore, both the input and output sequences are assumed to be reversed in the array X. The input sequence is assumed to be stored as follows:

X = (x_5, x_4, x_3, x_2, x_1) = (2, -1, 5, 1, -2)

The output sequence is stored by ESSL as follows:

X = (xx_5, xx_4, xx_3, xx_2, xx_1) = (5, 2, 1, -1, -2)

As a result, INDX is defined as follows:

INDX = ($indx_1$, $indx_2$, $indx_3$, $indx_4$, $indx_5$) = (1, 4, 2, 5, 3)

For example, because output sequence element $xx_4 = 2$ is input sequence element x_5, then INDX(4) = 5.

Call Statement and Input

```
             X   INCX  N   INDX
             |    |    |    |
CALL ISORTX( X , -1  , 5 , INDX )

X        =  (2, -1, 5, 1, -2)
```

Output

```
X        =  (5, 2, 1, -1, -2)
INDX     =  (1, 4, 2, 5, 3)
```

ISORTS, SSORTS, and DSORTS—Sort the Elements of a Sequence Using a Stable Sort and Note the Original Element Positions

These subroutines sort the elements of sequence ***x*** using a stable sort; that is, where equal elements occur in the input sequence, they remain in the same relative order in the output sequence. The original positions of the elements in sequence ***x*** are returned in the indices array `INDX`.

Note: If you need a stable sort, then you should use these subroutines rather than ISORTX, SSORTX, or DSORTX.

Table 146. Data Types

x, *work*	**Subroutine**
Integer	ISORTS
Short-precision real	SSORTS
Long-precision real	DSORTS

Syntax

Fortran	CALL ISORTS | SSORTS | DSORTS (*x*, *incx*, *n*, *indx*, *work*, *lwork*)
C and C++	isorts | ssorts | dsorts (*x*, *incx*, *n*, *indx*, *work*, *lwork*);
PL/I	CALL ISORTS | SSORTS | DSORTS (*x*, *incx*, *n*, *indx*, *work*, *lwork*);

On Entry

x
is the sequence ***x*** of length *n*, to be sorted. Specified as: a one-dimensional array of (at least) length 1+(*n*−1)|*incx*| elements, containing numbers of the data type indicated in Table 146.

incx
is the stride for both the input sequence ***x*** and the output sequence ***x***. If it is positive, elements are sorted into ascending order in the array, and if it is negative, elements are sorted into descending order in the array.

Specified as: a fullword integer. It can have any value.

n
is the number of elements in sequence ***x***. Specified as: a fullword integer; $n \geq 0$.

indx
See On Return.

work
is the storage work area used by this subroutine. Its size is specified by *lwork*. Specified as: an area of storage, containing numbers of the data type indicated in Table 146.

lwork
is the size of the work area specified by *work*— that is, the number of elements in *work*. Specified as: a fullword integer; $lwork \geq n/2$.

Note: This is the value to achieve optimal performance. The sort is performed regardless of the value you specify for *lwork*, but you may receive an attention message.

On Return

x
is the sequence ***x*** of length *n*, with its elements sorted into designated order in the array. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 146 on page 893.

indx
is the array, referred to as INDX, containing the *n* indices that indicate, for the elements in the sorted output sequence, the original positions of those elements in input sequence ***x***.

Note: It is important to remember that when you specify a negative stride, ESSL assumes that the order of the input and output sequence elements in the X array is reversed; however, the elements in INDX are not reversed. See "Function."

Returned as: a one-dimensional array of length *n*, containing fullword integers; $1 \le$ (INDX elements) $\le n$.

Function: The elements of input sequence ***x*** are sorted into ascending order using a partition sort. The sorting is stable; that is, where equal elements occur in the input sequence, they remain in the same relative order in the output sequence. The elements of output sequence ***x*** can be expressed as follows:

$$x_1 \le x_2 \le x_3 \le \ldots \le x_n$$

By specifying a negative stride for ***x***, the elements of input sequence ***x*** are assumed to be reversed in the array, $(x_n, x_{n-1}, \ldots, x_1)$, thus producing a sort into descending order within the array.

In addition, the INDX array contains the *n* indices that indicate, for the elements in the sorted output sequence, the original positions of those elements in input sequence ***x***. (These are not the positions in the array, but rather the positions in the sequence.) For each element x_j in the input sequence, becoming element xx_k in the output sequence, the elements in INDX are defined as follows:

$$\text{INDX}(k) = j \quad \text{for } j = 1, n \text{ and } k = 1, n$$
where $xx_k = x_j$

To understand INDX when you specify a negative stride, you should remember that both the input and output sequences, ***x***, are assumed to be in reverse order in array X, but INDX is not affected by stride. The sequence elements of ***x*** are assumed to be stored in your input array as follows:

$$\text{X} = (x_n, x_{n-1}, \ldots, x_1)$$

The sequence elements of ***x*** are stored in your output array by ESSL as follows:

$$\text{X} = (xx_n, xx_{n-1}, \ldots, xx_1)$$

where the elements xx_k are the elements x_j, sorted into descending order in X. As an example of how INDX is calculated, if $xx_1 = x_{n-1}$, then INDX(1) = $n-1$.

If *n* is 0, no computation is performed. See references [28] and [69].

Error Conditions

Resource Errors: Unable to allocate internal work area.

Computational Errors: None

Input-Argument Errors: $n < 0$

Example 1: This example shows how to sort a sequence ***x*** into ascending order by specifying a positive stride. Because this is a stable sort, the −1 elements remain in the same relative order in the output sequence, indicated by `INDX(2) = 2` and `INDX(3) = 4`.

Call Statement and Input

```
             X  INCX  N   INDX   WORK  LWORK
             |   |    |    |      |      |
CALL ISORTS( X , 2  , 5 , INDX , WORK ,  5  )

X        =  (2, . , -1, . , 5, . , -1, . , -2)
```

Output

```
X        =  (-2, . , -1, . , -1, . , 2, . , 5)
INDX     =  (5, 2, 4, 1, 3)
```

Example 2: This example shows how to sort a sequence ***x*** into descending order by specifying a negative stride. Therefore, both the input and output sequences are assumed to be reversed in the array `X`. The input sequence is assumed to be stored as follows:

`X` = $(x_5, x_4, x_3, x_2, x_1)$ = `(2, -1, 5, -1, -2)`

The output sequence is stored by ESSL as follows:

`X` = $(xx_5, xx_4, xx_3, xx_2, xx_1)$ = `(5, 2, -1, -1, -2)`

As a result, `INDX` is defined as follows:

`INDX` = $(indx_1, indx_2, indx_3, indx_4, indx_5)$ = `(1, 2, 4, 5, 3)`

For example, because output sequence element $xx_4 = 2$ is input sequence element x_5, then `INDX(4) = 5`. Also, because this is a stable sort, the −1 elements remain in the same relative order in the output sequence, indicated by `INDX(2) = 2` and `INDX(3) = 4`.

Call Statement and Input

```
             X   INCX  N   INDX   WORK  LWORK
             |    |    |    |      |      |
CALL ISORTS( X , -1  , 5 , INDX , WORK ,  5  )

X        =  (2, -1, 5, -1, -2)
```

Output

```
X        =  (5, 2, -1, -1, -2)
INDX     =  (1, 2, 4, 5, 3)
```

IBSRCH, SBSRCH, and DBSRCH—Binary Search for Elements of a Sequence X in a Sorted Sequence Y

These subroutines perform a binary search for the locations of the elements of sequence ***x*** in another sequence ***y***, where ***y*** has been sorted into ascending order. The first occurrence of each element is found. When an exact match is not found, the position of the next larger element in ***y*** is indicated. The locations are returned in the indices array INDX, and, optionally, return codes indicating whether the exact elements were found are returned in array RC.

Table 147. Data Types

x, ***y***	**Subroutine**
Integer	IBSRCH
Short-precision real	SBSRCH
Long-precision real	DBSRCH

Syntax

Fortran	CALL IBSRCH | SBSRCH | DBSRCH (*x*, *incx*, *n*, *y*, *incy*, *m*, *indx*, *rc*, *iopt*)
C and C++	ibsrch | sbsrch | dbsrch (*x*, *incx*, *n*, *y*, *incy*, *m*, *indx*, *rc*, *iopt*);
PL/I	CALL IBSRCH | SBSRCH | DBSRCH (*x*, *incx*, *n*, *y*, *incy*, *m*, *indx*, *rc*, *iopt*);

On Entry

x
is the sequence ***x*** of length *n*, containing the elements for which sequence ***y*** is searched. Specified as: a one-dimensional array, containing numbers of the data type indicated in Table 147. It must have at least $1+(n-1)|incx|$ elements.

incx
is the stride for sequence ***x***. Specified as: a fullword integer. It can have any value.

n
is the number of elements in sequence ***x*** and arrays INDX and RC. Specified as: a fullword integer; $n \geq 0$.

y
is the sequence ***y*** of length *m*, to be searched, where ***y*** must be sorted into ascending order.

Note: Be careful in specifying the stride for sequence ***y***. A negative stride reverses the direction of the search, because the order of the sequence elements is reversed in the array.

Specified as: a one-dimensional array of (at least) length $1+(m-1)|incy|$, containing numbers of the data type indicated in Table 147.

incy
is the stride for sequence ***y***. Specified as: a fullword integer. It can have any value.

m
is the number of elements in sequence ***y***. Specified as: a fullword integer; $m \geq 0$.

indx
See On Return.

rc
See On Return.

iopt
has the following meaning, where:

If *iopt* = 0, the *rc* argument is not used in the computation.

If *iopt* = 1, the *rc* argument is used in the computation.

Specified as: a fullword integer; *iopt* = 0 or 1.

On Return

indx
is the array, referred to as INDX, containing the *n* indices that indicate the positions of the elements of sequence ***x*** in sequence ***y***. The first occurrence of the element found in sequence ***y*** is indicated in array INDX. When an exact match between an element of sequence ***x*** and an element of sequence ***y*** is not found, the position of the next larger element in sequence ***y*** is indicated. When the element in sequence ***x*** is larger than all the elements in sequence ***y***, then *m*+1 is indicated in array INDX.

Returned as: a one-dimensional array of length *n*, containing fullword integers; 1 ≤ (INDX elements) ≤ *m*+1.

rc
has the following meaning, where:

If *iopt* = 0, then *rc* is not used, and its contents remain unchanged.

If *iopt* = 1, it is the array, referred to as RC, containing the *n* return codes that indicate whether the elements in sequence ***x*** were found in sequence ***y***. For $i = 1, n$, elements RC(i) = 0 if x_i matches an element in sequence ***y***, and RC(i) = 1 if an exact match is not found in sequence ***y***.

Returned as: a one-dimensional array of length *n*, containing fullword integers; RC(i) = 0 or 1.

Notes

1. The elements of ***y*** must be sorted into ascending order; otherwise, results are unpredictable. For details on how to do this, see "ISORT, SSORT, and DSORT—Sort the Elements of a Sequence" on page 888.
2. If you do not need the results provided in array RC by these subroutines, you get better performance if you do not request it. That is, specify 0 for the *iopt* argument.

Function: These subroutines perform a binary search for the first occurrence (or last occurrence, using negative stride) of the locations of the elements of sequence ***x*** in another sequence ***y***, where ***y*** must be sorted into ascending order before calling this subroutine. The first occurrence of each element is found. Two arrays are returned, containing the results of the binary searches:

- INDX, the indices array, contains the positions of the elements of sequence ***x*** in sequence ***y***. When an exact match between values of elements in sequences ***x*** and ***y*** is not found, the location of the next larger element in sequence ***y*** is indicated in array INDX.
- RC, the return codes array, indicates for each element in sequence ***x*** whether the exact element was found in sequence ***y***. If you do not need these results, you get better performance if you set *iopt* = 0.

The results returned for the INDX and RC arrays are expressed as follows:

For $i = 1, n$
  for all $y_j \geq x_i, j = 1, m$, INDX(i) = min(j)
  if all $y_j < x_i, j = 1, m$, INDX(i) = m+1
And for $i = 1, n$
  if $x_i = y_{\text{INDX}(i)}$, RC(i) = 0
  if $x_i \neq y_{\text{INDX}(i)}$, RC($i$) = 1

where:

x is a sequence of length n, containing the search elements.
y is a sequence of length m to be searched. It must be sorted into ascending order.
INDX is the array of length n of indices.
RC is the array of length n of return codes.

See reference [69]. If n is 0, no search is performed. If m is 0, then:

INDX(i) = 1 and RC(i) = 1 for $i = 1, n$

It is important to note that a negative stride for sequence ***y*** reverses the direction of the search, because the order of the sequence elements is reversed in the array. For more details on sorting sequences, see “Function” on page 888.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. $n < 0$
2. $m < 0$
3. *iopt* ≠ 0 or 1

Example 1: This example shows a search where sequences ***x*** and ***y*** have positive strides, and where the optional return codes are returned as part of the output.

Call Statement and Input

```
             X  INCX  N   Y  INCY  M    INDX   RC  IOPT
             |   |    |   |   |    |     |     |    |
CALL IBSRCH( X , 2  , 5 , Y , 1  , 10 , INDX , RC , 1 )

X        =  (-3, . , 125, . , 30, . , 20, . , 70)
Y        =  (10, 20, 30, 30, 40, 50, 60, 80, 90, 100)
```

Output

```
INDX     =  (1, 11, 3, 2, 8)
RC       =  (1, 1, 0, 0, 1)
```

Example 2: This example shows the same calling sequence as in Example 1, except that it includes the IOPT argument, specified as 1. This is equivalent to using the calling sequence in Example 1 and gives the same results.

Call Statement and Input

```
            X  INCX  N   Y  INCY  M    INDX   RC  IOPT
            |   |    |   |   |    |     |     |    |
CALL IBSRCH( X , 2  , 5 , Y , 1  , 10 , INDX , RC , 1  )
```

Example 3: This example shows a search where sequence ***x*** has a negative stride, and sequence ***y*** has a positive stride. The optional return codes are not requested, because IOPT is specified as 0.

Call Statement and Input

```
            X   INCX  N   Y  INCY  M    INDX   RC  IOPT
            |    |    |   |   |    |     |     |    |
CALL IBSRCH( X , -2  , 5 , Y , 1  , 10 , INDX , RC , 0 )

X        =  (-3, . , 125, . , 30, . , 20, . , 70)
Y        =  (10, 20, 30, 30, 40, 50, 60, 80, 90, 100)
```

Output

```
INDX     =  (8, 2, 3, 11, 1)
RC       =  (not relevant)
```

Example 4: This example shows a search where sequence ***x*** has a positive stride, and sequence ***y*** has a negative stride. As shown below, elements of ***y*** are in descending order in array Y. The optional return codes are not requested, because IOPT is specified as 0.

Call Statement and Input

```
            X  INCX  N   Y   INCY  M    INDX   RC  IOPT
            |   |    |   |    |    |     |     |    |
CALL IBSRCH( X , 2  , 5 , Y , -1  , 10 , INDX , RC , 0  )

X        =  (-3, . , 125, . , 30, . , 20, . , 70)
Y        =  (100, 90, 80, 60, 50, 40, 30, 30, 20, 10)
RC       =(not relevant)
```

Output

```
INDX     =  (1, 11, 3, 2, 8)
```

ISSRCH, SSSRCH, and DSSRCH—Sequential Search for Elements of a Sequence X in the Sequence Y

These subroutines perform a sequential search for the locations of the elements of sequence ***x*** in another sequence ***y***. Depending on the sign of the *idir* argument, the search direction indicator, the location of either the first or last occurrence of each element is indicated in the resulting indices array INDX. When an exact match between elements is not found, the position is indicated as 0.

Table 148. Data Types

x, ***y***	**Subroutine**
Integer	ISSRCH
Short-precision real	SSSRCH
Long-precision real	DSSRCH

Syntax

Fortran	CALL ISSRCH | SSSRCH | DSSRCH (*x*, *incx*, *n*, *y*, *incy*, *m*, *idir*, *indx*)
C and C++	issrch | sssrch | dssrch (*x*, *incx*, *n*, *y*, *incy*, *m*, *idir*, *indx*);
PL/I	CALL ISSRCH | SSSRCH | DSSRCH (*x*, *incx*, *n*, *y*, *incy*, *m*, *idir*, *indx*);

On Entry

x
is the sequence ***x*** of length *n*, containing the elements for which sequence ***y*** is searched. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 148.

incx
is the stride for sequence ***x***. Specified as: a fullword integer. It can have any value.

n
is the number of elements in sequence ***x*** and array INDX. Specified as: a fullword integer; $n \geq 0$.

y
is the sequence ***y*** of length *m* to be searched.

Note: Be careful in specifying the stride for sequence ***y***. A negative stride reverses the direction of the search, because the order of the sequence elements is reversed in the array.

Specified as: a one-dimensional array of (at least) length $1+(m-1)|incy|$, containing numbers of the data type indicated in Table 148.

incy
is the stride for sequence ***y***. Specified as: a fullword integer. It can have any value.

m
is the number of elements in sequence ***y***. Specified as: a fullword integer; $m \geq 0$.

idir
indicates the search direction, where:

If $idir \geq 0$, sequence ***y*** is searched from the first element to the last (1, *n*), thus finding the first occurrence of the element in the sequence.

If *idir* < 0, sequence ***y*** is searched from the last element to the first (*n*, 1), thus finding the last occurrence of the element in the sequence.

Specified as: a fullword integer. It can have any value.

indx

See On Return.

On Return

indx

is the array, referred to as INDX, containing the *n* indices that indicate the positions of the elements of sequence ***x*** in sequence ***y***, where:

If *idir* ≥ 0, the first occurrence of the element found in sequence ***y*** is indicated in array INDX.

If *idir* < 0, the last occurrence of the element found in sequence ***y*** is indicated in array INDX.

In all cases, if no match is found, 0 is indicated in array INDX.

Returned as: a one-dimensional array of length *n*, containing fullword integers; 0 ≤ (INDX elements) ≤ *m*.

Function: These subroutines perform a sequential search for the first occurrence (or last occurrence, using a negative *idir*) of the locations of the elements of sequence ***x*** in another sequence ***y***. The results of the sequential searches are returned in the indices array INDX, indicating the positions of the elements of sequence ***x*** in sequence ***y***. The positions indicated in array INDX are calculated relative to the first sequence element position—that is, the position of y_1. When an exact match between values of elements in sequences ***x*** and ***y*** is not found, 0 is indicated in array INDX for that position.

The results returned in array INDX are expressed as follows:

For $i = 1, n$
  for all $y_j = x_i, j = 1, m$
    INDX(i) = min(j), if *idir* ≥ 0
    INDX(i) = max(j), if *idir* < 0
  if all $y_j \neq x_i, j = 1, m$
    INDX(i) = 0

where:

x is a sequence of length *n*, containing the search elements.
y is a sequence of length *m* to be searched.
INDX is the array of length *n* of indices.

See reference [69]. If *n* is 0, no search is performed.

It is important to note that a negative stride for sequence ***y*** reverses the direction of the search, because the order of the sequence elements is reversed in the array.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. $n < 0$
2. $m < 0$

Example 1: This example shows a search where sequences ***x*** and ***y*** have positive strides, and the search direction indicator, *idir*, is positive.

Call Statement and Input

```
             X  INCX  N   Y  INCY  M  IDIR  INDX
             |   |    |   |   |    |   |     |
CALL ISSRCH( X , 1  , 3 , Y , 2  , 8 , 1  , INDX )

X        =  (0, 12, 3)
Y        =  (0, . , 8, . , 12, . , 0, . , 1, . , 4, . , 0, . , 2)
```

Output

```
INDX     =  (1, 3, 0)
```

Example 2: This example shows a search where sequences ***x*** and ***y*** have positive strides, and the search direction indicator, *idir*, is negative.

Call Statement and Input

```
             X  INCX  N   Y  INCY  M   IDIR  INDX
             |   |    |   |   |    |    |     |
CALL ISSRCH( X , 2  , 3 , Y , 2  , 8 , -1  , INDX )

X        =  (0, . , 12, . , 3)
Y        =  (0, . , 8, . , 12, . , 0, . , 1, . , 4, . , 0, . , 2)
```

Output

```
INDX     =  (7, 3, 0)
```

Example 3: This example shows a search where sequences ***x*** and ***y*** have negative strides, and the search direction indicator, *idir*, is positive.

Call Statement and Input

```
             X   INCX  N   Y   INCY  M  IDIR  INDX
             |    |    |   |    |    |  |     |
CALL ISSRCH( X , -1  , 3 , Y , -2  , 8 , 1  , INDX )

X        =  (0, 12, 3)
Y        =  (0, . , 8, . , 12, . , 0, . , 1, . , 4, . , 0, . , 2)
```

Output

```
INDX     =  (0, 6, 2)
```

Example 4: This example shows a search where sequences ***x*** and ***y*** have negative strides, and the search direction indicator, *idir*, is negative.

Call Statement and Input

```
             X   INCX  N   Y   INCY  M   IDIR  INDX
             |    |    |   |   |    |    |     |
CALL ISSRCH( X , -2  , 3 , Y , -1  , 8 , -1  , INDX )

X        =  (0, . , 12, . , 3)
Y        =  (0, 8, 12, 0, 1, 4, 0, 2)
```

Output

```
INDX     =  (0, 6, 8)
```

Chapter 14. Interpolation

The interpolation subroutines are described in this chapter.

Overview of the Interpolation Subroutines

The interpolation subroutines provide the capabilities of doing polynomial interpolation, local polynomial interpolation, and one- and two-dimensional cubic spline interpolation (Table 149).

Table 149. List of Interpolation Subroutines

Descriptive Name	Short-Precision Subroutine	Long-Precision Subroutine	Page
Polynomial Interpolation	SPINT	DPINT	907
Local Polynomial Interpolation	STPINT	DTPINT	912
Cubic Spline Interpolation	SCSINT	DCSINT	915
Two-Dimensional Cubic Spline Interpolation	SCSIN2	DCSIN2	921

Use Considerations

Polynomial interpolation (SPINT and DPINT) is a global scheme. As the number of data points increases, the degree of the interpolating polynomial is raised; therefore, the graph of the interpolating polynomial tends to be oscillatory.

Local polynomial interpolation (STPINT and DTPINT) is a local scheme. The data generated is affected only by locally grouped data points. The degree of the local interpolating polynomial is usually lower than a global interpolating polynomial.

Performance and Accuracy Considerations

1. Doing extrapolation with SPINT and DPINT is not encouraged unless you know the consequences of doing polynomial extrapolation.
2. If performance is the overriding consideration, you should investigate using the general signal processing subroutines, DQINT and SQINT.
3. There are some ESSL-specific rules that apply to the results of computations on the workstation processors using the ANSI/IEEE standards. For details, see “What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?” on page 44.

© Copyright IBM Corp. 1997

Interpolation Subroutines

This section contains the interpolation subroutine descriptions.

SPINT and DPINT—Polynomial Interpolation

These subroutines compute the Newton divided difference coefficients and perform a polynomial interpolation through a set of data points at specified abscissas.

Table 150. Data Types

x, ***y***, ***c***, ***t***, ***s***	**Subroutine**
Short-precision real	SPINT
Long-precision real	DPINT

Syntax

Fortran	CALL SPINT | DPINT (*x*, *y*, *n*, *c*, *ninit*, *t*, *s*, *m*)
C and C++	spint | dpint (*x*, *y*, *n*, *c*, *ninit*, *t*, *s*, *m*);
PL/I	CALL SPINT | DPINT (*x*, *y*, *n*, *c*, *ninit*, *t*, *s*, *m*);

On Entry

x
is the vector ***x*** of length *n*, containing the abscissas of the data points used in the interpolations. The elements of ***x*** must be distinct. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 150.

y
is the vector ***y*** of length *n*, containing the ordinates of the data points used in the interpolations. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 150.

n
is the number of elements in vectors ***x, y,*** and ***c***—that is, the number of data points. Specified as: a fullword integer; $n \geq 0$.

c
is the vector ***c*** of length *n*, where:

If $ninit \leq 0$, all elements of ***c*** are undefined on entry.

If $ninit > 0$, ***c*** contains the Newton divided difference coefficients, c_j for $j = 1$, *ninit*, for the interpolating polynomial through the data points (x_j,y_j) for $j = 1$, *ninit*. If $ninit < n$, the values of c_j for $j = ninit+1, n$ are undefined.

Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 150.

ninit
indicates the following:

If $ninit \leq 0$, this is the first call to this subroutine with the data in ***x*** and ***y***; therefore, none of the Newton divided difference coefficients in ***c*** have been initialized.

If $ninit > 0$, a previous call to this subroutine was made with the data points (x_j, y_j) for $j = 1$, *ninit*, where:

- If $ninit = n$, all the Newton divided difference coefficients in ***c*** were computed for the data points. No additional coefficients are computed on this entry.

- If *ninit* < *n*, the first *ninit* Newton divided difference coefficients in ***c*** were computed for the data points (x_j, y_j) for $j = 1$, *ninit*. The coefficients are updated for the additional data points (x_j, y_j) for j = *ninit*+1, *n* on this entry.

Specified as: a fullword integer; *ninit* ≤ *n*.

t
is the vector ***t*** of length *m*, containing the abscissas at which interpolation is to be done. Specified as: a one-dimensional array of (at least) length *m*, containing numbers of the data type indicated in Table 150 on page 907.

s
See On Return.

m
is the number of elements in vectors ***t*** and ***s***—that is, the number of interpolations to be performed. Specified as: a fullword integer; *m* ≥ 0.

On Return

c
is the vector ***c*** of length *n*, containing the coefficients of the Newton divided difference form of the interpolating polynomial through the data points (x_j,y_j) for $j = 1$, *n*. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 150 on page 907.

ninit
is the number of coefficients, *n*, in output vector ***c***. (If you call this subroutine again with the same data, this value should be specified for *ninit*.) Returned as: a fullword integer; *ninit* = *n*.

s
is the vector ***s*** of length *m*, containing the resulting interpolated values; that is, each s_i is the value of the interpolating polynomial evaluated at t_i. Returned as: a one-dimensional array of (at least) length *m*, containing numbers of the data type indicated in Table 150 on page 907.

Notes

1. In your C program, argument *ninit* must be passed by reference.
2. Vectors ***x, y,*** and ***t*** must have no common elements with vectors ***c*** and ***s***, and vector ***c*** must have no common element with vector ***s***; otherwise, results are unpredictable.
3. The elements of vector ***x*** must be distinct; that is, $x_i \neq x_j$ if $i \neq j$ for $i, j = 1, n$.

Function: Polynomial interpolation is performed at specified abscissas, t_i for $i = 1, m$, in vector ***t***, using the method of Newton divided differences through the data points:

(x_j, y_j) for $j = 1, n$

where:

x_j are elements of vector ***x***.
y_j are elements of vector ***y***.

The interpolated value at each t_i is returned in s_i for $i = 1, m$. See references [15] and [51]. The interpolating values returned in ***s*** are computed using the Newton divided difference coefficients, as defined in the following section.

The divided difference coefficients, c_j for $j = 1, n$, are returned in vector ***c***. These coefficients can then be reused on subsequent calls to this subroutine, using the same data

points (x_j, y_j), but with new values of t_i. If the number of data points is increased from one call this subroutine to the next, the new coefficients are computed, and the existing coefficients are updated (not recomputed). This feature can be used to test for the convergence of the interpolations through a sequence of an increasingly larger set of points.

The values specified for *ninit* and *m* indicate which combination of functions are performed by this subroutine: computing the coefficients, performing the interpolation, or both. If $m = 0$, only the divided difference coefficients are computed. No interpolation is performed. If $n = 0$, no computation or interpolation is performed.

Newton Divided Differences and Interpolating Values: The Newton divided differences of the following data points:

(x_j, y_j) for $j = 1, n$
where $x_j \neq x_l$ if $j \neq l$ for $j, l = 1, n$

are denoted by $\delta_k y_j$ for $k = 0, 1, 2, ..., n-1$ and $j = 1, 2, ..., n-k$, and are defined as follows:

For $k = 0$ and 1:
$\delta_0 y_j = y_j$ for $j = 1, 2, ..., n$
$\delta_1 y_j = (y_{j+1} - y_j) / (x_{j+1} - x_j)$ for $j = 1, 2, ..., n-1$
For $k = 2, 3, ..., n-1$:
$\delta_k y_j = (\delta_{k-1} y_{j+1} - \delta_{k-1} y_j) / (x_{j+k} - x_j)$ for $j = 1, 2, ..., n-k$

The value s of the Newton divided difference form of the interpolating polynomial evaluated at an abscissa t is given by:

$$s = y_n + (t-x_n)\,\delta_1 y_{n-1} + (t-x_{n-1})\,(t-x_n)\,\delta_2 y_{n-2} + ...+(t-x_2)\,(t-x_3)\,...\,(t-x_n)\,\delta_{n-1} y_1$$

Therefore, on output, the coefficients in vector ***c*** are as follows:

$$c_n = y_n$$
$$c_{n-1} = \delta_1 y_{n-1}$$
$$c_{n-2} = \delta_2 y_{n-2}$$
.
.
.
$$c_1 = \delta_{n-1} y_1$$

Also, the interpolating values in ***s***, in terms of ***c***, are as follows for $i = 1, m$:

$$s_i = c_n + (t_i-x_n)\,c_{n-1} + (t_i-x_{n-1})\,(t_i-x_n)\,c_{n-2} + ... + (t_i-x_2)\,(t_i-x_3)\,...\,(t_i-x_n)\,c_1$$

Error Conditions

Computational Errors: None

Input-Argument Errors

1. $n < 0$
2. $ninit > n$
3. $m < 0$

Example 1: This example shows a quadratic polynomial interpolation on the initial call with the specified data points; that is, NINIT = 0, and C contains all undefined values. On output, NINIT and C are updated with new values.

Call Statement and Input

```
           X   Y   N   C  NINIT  T   S   M
           |   |   |   |    |    |   |   |
CALL SPINT( X , Y , 3 , C ,  0  , T , S , 2 )
```

```
X        =  (-0.50, 0.00, 1.00)
Y        =  (0.25, 0.00, 1.00)
C        =  ( . , . , . )
T        =  (-0.2, 0.2)
```

Output

```
C        =  (1.00, 1.00, 1.00)
NINIT    =  3
S        =  (0.04, 0.04)
```

Example 2: This example shows a quadratic polynomial interpolation on a subsequent call with the same data points specified in Example 1, but using a different set of abscissas in T. In this case, NINIT = N = 3, and C contains the values defined on output in Example 1. On output here, the values in NINIT and C are unchanged.

Call Statement and Input

```
           X   Y   N   C  NINIT  T   S   M
           |   |   |   |    |    |   |   |
CALL SPINT( X , Y , 3 , C ,  3  , T , S , 2 )
```

```
X        =  (-0.50, 0.00, 1.00)
Y        =  (0.25, 0.00, 1.00)
C        =  (1.00, 1.00, 1.00)
T        =  (-0.10, 0.10)
```

Output

```
C        =  (1.00, 1.00, 1.00)
NINIT    =  3
S        =  (0.01, 0.01)
```

Example 3: This example is the same as Example 2 except that it specifies additional data points on the subsequent call to the subroutine. In this case, 0 < NINIT < N. On output here, the values in NINIT and C are updated. The interpolating polynomial is a degree of 4.

Call Statement and Input

```
           X   Y   N   C  NINIT  T   S   M
           |   |   |   |    |    |   |   |
CALL SPINT( X , Y , 5 , C ,  3  , T , S , 2 )

X        =  (-0.50, 0.00, 1.00, -1.00, 0.50)
Y        =  (0.25, 0.00, 1.00, 1.10, 0.26)
C        =  (1.00, 1.00, 1.00, . , . )
T        =  (-0.10, 0.10)
```

Output

```
C        =  (0.04, -0.06, 1.02, -0.56, 0.26)
NINIT    =  5
S        =  (0.0072, 0.0130)
```

STPINT and DTPINT—Local Polynomial Interpolation

These subroutines perform a polynomial interpolation at specified abscissas, using data points selected from a table of data.

Table 151. Data Types

x, ***y***, ***t***, ***s***, *aux*	**Subroutine**
Short-precision real	STPINT
Long-precision real	DTPINT

Syntax

Fortran	CALL STPINT | DTPINT (*x*, *y*, *n*, *nint*, *t*, *s*, *m*, *aux*, *naux*)
C and C++	stpint | dtpint (*x*, *y*, *n*, *nint*, *t*, *s*, *m*, *aux*, *naux*);
PL/I	CALL STPINT | DTPINT (*x*, *y*, *n*, *nint*, *t*, *s*, *m*, *aux*, *naux*);

On Entry

x
is the vector ***x*** of length *n*, containing the abscissas of the data points used in the interpolations. The elements of ***x*** must be distinct and sorted into ascending order. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 151.

y
is the vector ***y*** of length *n*, containing the ordinates of the data points used in the interpolations. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 151.

n
is the number of elements in vectors ***x*** and ***y***—that is, the number of data points. Specified as: a fullword integer; $n \geq 0$.

nint
is the number of data points to be used in the interpolation at any given point. Specified as: a fullword integer; $0 \leq nint \leq n$.

t
is the vector ***t*** of length *m*, containing the abscissas at which interpolation is to be done. For optimal performance, ***t*** should be sorted into ascending order. Specified as: a one-dimensional array of (at least) length *m*, containing numbers of the data type indicated in Table 151.

s
See On Return.

m
is the number of elements in vectors ***t*** and ***s***—that is, the number of interpolations to be performed. Specified as: a fullword integer; $m \geq 0$.

aux
has the following meaning:

If $naux = 0$ and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is the storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing numbers of the data type indicated in Table 151. On output, the contents are overwritten.

naux
is the size of the work area specified by *aux*—that is, the number of elements in *aux*. Specified as: a fullword integer, where:

If *naux* = 0 and error 2015 is unrecoverable, STPINT and DTPINT dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux* ≥ *nint*+*m*.

On Return

s
is the vector ***s*** of length *m*, containing the resulting interpolated values; that is, each s_i is the value of the interpolating polynomial evaluated at t_i. Returned as: a one-dimensional array of (at least) length *m*, containing numbers of the data type indicated in Table 151 on page 912.

Notes

1. Vectors ***x, y,*** and ***t*** must have no common elements with vector ***s*** or work area *aux*; otherwise, results are unpredictable. See "Concepts" on page 55.
2. The elements of vector ***x*** must be distinct and must be sorted into ascending order; that is, $x_1 < x_2 < ... < x_n$. Otherwise, results are unpredictable. For details on how to do this, see "ISORT, SSORT, and DSORT—Sort the Elements of a Sequence" on page 888.
3. The elements of vector ***t*** should be sorted into ascending order; that is, $t_1 \le t_2 \le t_3 \le ... \le t_m$. Otherwise, performance is affected.
4. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: Polynomial interpolation is performed at specified abscissas, t_i for $i = 1, m$, in vector ***t***, using *nint* points selected from the following data:

(x_j, y_j) for $j = 1, n$

where:

$x_1 < x_2 < x_3 < ... < x_n$
x_j are elements of vector ***x***.
y_j are elements of vector ***y***.

The points (x_j, y_j), used in the interpolation at a given abscissa t_i, are chosen as follows, where $k = nint/2$:

For $t_i \le x_{k+1}$, the first *nint* points are used.
For $t_i > x_{n-nint+k}$, the last *nint* points are used.
Otherwise, points *h* through *h*+*nint*−1 are used, where:
$x_{h+k-1} < t_i \le x_{h+k}$

The interpolated value at each t_i is returned in s_i for $i = 1, m$. See references [15] and [51]. If *n, nint*, or *m* is 0, no computation is performed. For a definition of the

polynomial interpolation function performed through a set of data points, see "Function" on page 908.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors: None

Input-Argument Errors

1. $n < 0$
2. $nint < 0$ or $nint > n$
3. $m < 0$
4. Error 2015 is recoverable or $naux \neq 0$, and *naux* is too small—that is, less than the minimum required value specified in the syntax for this argument. Return code 1 is returned if error 2015 is recoverable.

Example 1: This example shows interpolation using two data points—that is, linear interpolation—at each t_i value.

Call Statement and Input

```
             X   Y   N   NINT   T   S   M   AUX  NAUX
             |   |   |     |    |   |   |    |    |
CALL STPINT( X , Y , 10 ,  2  , T , S , 5 , AUX , 7  )

X        =  (0.0, 0.4, 1.0, 1.5, 2.1, 2.6, 3.0, 3.4, 3.9, 4.3)
Y        =  (1.0, 2.0, 3.0, 4.0, 5.0, 5.0, 4.0, 3.0, 2.0, 1.0)
T        =  (-1.0, 0.1, 1.1, 1.2, 3.9)
```

Output

```
S        =  (-1.5000, 1.2500, 3.2000, 3.4000, 2.0000)
```

Example 2: This example shows interpolation using three data points—that is, quadratic interpolation—at each t_i value.

Call Statement and Input

```
             X   Y   N   NINT   T   S   M   AUX  NAUX
             |   |   |     |    |   |   |    |    |
CALL STPINT( X , Y , 10 ,  3  , T , S , 5 , AUX , 8  )

X        =  (0.0, 0.4, 1.0, 1.5, 2.1, 2.6, 3.0, 3.4, 3.9, 4.3)
Y        =  (1.0, 2.0, 3.0, 4.0, 5.0, 5.0, 4.0, 3.0, 2.0, 1.0)
T        =  (-1.0, 0.1, 1.1, 1.2, 3.9)
```

Output

```
S        =  (-2.6667, 1.2750, 3.2121, 3.4182, 2.0000)
```

SCSINT and DCSINT—Cubic Spline Interpolation

These subroutines compute the coefficients of the cubic spline through a set of data points and evaluate the spline at specified abscissas.

Table 152. Data Types

x, ***y***, ***C***, ***t***, ***s***	**Subroutine**
Short-precision real	SCSINT
Long-precision real	DCSINT

Syntax

Fortran	CALL SCSINT | DCSINT (*x*, *y*, *c*, *n*, *init*, *t*, *s*, *m*)
C and C++	scsint | dcsint (*x*, *y*, *c*, *n*, *init*, *t*, *s*, *m*);
PL/I	CALL SCSINT | DCSINT (*x*, *y*, *c*, *n*, *init*, *t*, *s*, *m*);

On Entry

x

is the vector ***x*** of length *n*, containing the abscissas of the data points that define the spline. The elements of ***x*** must be distinct and sorted into ascending order. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 152.

y

is the vector ***y*** of length *n*, containing the ordinates of the data points that define the spline. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 152.

c

is the matrix ***C*** with elements c_{jk} for $j = 1, n$ and $k = 1, 4$ that contain the following:

If $init \leq 0$, all elements of ***c*** are undefined on entry.

If $init = 1$, c_{11} contains the spline derivative at x_1.

If $init = 2$, c_{21} contains the spline derivative at x_n.

If $init = 3$, c_{11} contains the spline derivative at x_1, and c_{21} contains the spline derivative at x_n.

If $init > 3$, ***c*** contains the coefficients of the spline computed for the data points (x_j,y_j) for $j = 1, n$ on a previous call to this subroutine.

Specified as: an *n* by (at least) 4 array, containing numbers of the data type indicated in Table 152.

n

is the number of elements in vectors ***x*** and ***y*** and the number of rows in matrix ***C***—that is, the number of data points. Specified as: a fullword integer; $n \geq 0$.

init

indicates the following, where in those cases for uninitialized coefficients, this is the first call to this subroutine with the data in ***x*** and ***y***:

If *init* ≤ 0, the coefficients are uninitialized. The second derivatives of the spline at x_1 and x_n are set to zero. (These are free end conditions, also called natural boundary conditions.)

If *init* = 1, the coefficients are uninitialized. The value in c_{11} is used as the spline derivative at x_1.

If *init* = 2, the coefficients are uninitialized. The value in c_{21} is used as the spline derivative at x_n.

If *init* = 3, the coefficients are uninitialized. The value in c_{11} is used as the spline derivative at x_1 and the value in c_{21} is used as the spline derivative at x_n.

If *init* > 3, the coefficients in ***c*** were computed for data points (x_j, y_j) for $j = 1, n$ on a previous call to this subroutine.

Specified as: a fullword integer. It can have any value.

t

is the vector ***t*** of length *m*, containing the abscissas at which the spline is evaluated. Specified as: a one-dimensional array of (at least) length *m*, containing numbers of the data type indicated in Table 152 on page 915.

s

See On Return.

m

is the number of elements in vectors ***t*** and ***s***—that is, the number of points at which the spline interpolation is evaluated. Specified as: a fullword integer; $m \geq 0$.

On Return

c

is the matrix ***C***, containing the coefficients of the spline through the data points (x_j,y_j) for $j = 1, n$. Returned as: an *n* by (at least) 4 array, containing numbers of the data type indicated in Table 152 on page 915.

init

is an indicator that is set to indicate that the coefficients have been initialized. (If you call this subroutine again with the same data, this value should be specified for *init*.) Returned as: a fullword integer; *init* = 4.

s

is the vector ***s*** of length *m*, containing the resulting values of the spline; that is, each s_i is the value of the spline evaluated at t_i. Returned as: a one-dimensional array of (at least) length *m*, containing numbers of the data type indicated in Table 152 on page 915.

Notes

1. In your C program, argument *init* must be passed by reference.
2. Vectors ***x, y,*** and ***t*** must have no common elements with matrix ***C*** and vector ***s***, and matrix ***C*** must have no common elements with vector ***s***; otherwise, results are unpredictable.
3. The elements of vector ***x*** must be distinct and must be sorted into ascending order; that is, $x_1 < x_2 < ... < x_n$. Otherwise, results are unpredictable. For details on how to do this, see "ISORT, SSORT, and DSORT—Sort the Elements of a Sequence" on page 888.

Function: Interpolation is performed at specified abscissas, t_i for $i = 1, m$, in vector ***t***, using the cubic spline passing through the data points:

(x_j, y_j) for $j = 1, n$

where:

$x_1 < x_2 < x_3 < ... < x_n$
x_j are elements of vector ***x***.
y_j are elements of vector ***y***.

The value of the cubic spline at each t_i is returned in s_i for $i = 1, m$. See references [15] and [51]. The coefficients of the spline, c_{jk} for $j = 1, n$ and $k = 1, 4$, are returned in matrix ***C***. These coefficients can then be reused on subsequent calls to this subroutine, using the same data points (x_j, y_j), but with new values of t_i. The cubic spline values returned in ***s*** are computed using the coefficients as follows:

$$s_i = c_{j1} + c_{j2}\,(x_j - t_i) + c_{j3}\,(x_j - t_i)^2 + c_{j4}\,(x_j - t_i)^3 \qquad \text{for } i = 1, m$$

where:

$j = 1$ for $t_i \leq x_1$
$j = k$ for $x_1 < t_i \leq x_n$
$j = n$ for $x_n < t_i$, such that $x_{k-1} < t_i \leq x_k$

The values specified for *m* and *init* indicate which combination of functions are performed by this subroutine:

- If $m = 0$ and $init > 3$, no computation is performed.
- If $m = 0$ and $init \leq 3$, only the coefficients are computed, and no interpolation is performed.
- If $m \neq 0$ and $init > 3$, the coefficients are not computed, and the interpolation is performed.
- If $m \neq 0$ and $init \leq 3$, the coefficients are computed, and the interpolation is performed.

In addition, if $n = 0$, no computation is performed.

The values specified for *n* and *init* determine the type of spline function:

- If $n = 1$, the constructed spline is a constant function.
- If $n = 2$ and $init = 0$, the constructed spline is a line through the points.
- If $n = 2$ and $init = 1$, the constructed spline is a quadratic function through the points whose derivative at x_1 is c_{11}.
- If $n = 2$ and $init = 2$, the constructed spline is a quadratic function through the points whose derivative at x_n is c_{21}.
- If $n = 2$ and $init = 3$, the constructed spline is a cubic function through the points whose derivative at x_1 is c_{11} and at x_n is c_{21}.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. $n < 0$
2. $m < 0$

Example 1: This example computes the spline coefficients through a set of data points with no derivative value specified. It also evaluates the spline at the abscissas specified in T. On output, INIT and C are updated with new values.

Call Statement and Input

```
             X   Y   C   N  INIT  T   S   M
             |   |   |   |   |    |   |   |
CALL SCSINT( X , Y , C , 6 , 0  , T , S , 4 )

X        =  (1.000, 2.000, 3.000, 4.000, 5.000, 6.000)
Y        =  (0.000, 1.000, 2.000, 1.100, 0.000, -1.000)
C        =(not relevant)
T        =  (-1.000, 2.500, 4.000, 7.000)
```

Output

```
         ⎡                              ⎤
         ⎢  0.000  -0.868   0.000  -0.132 ⎥
         ⎢  1.000  -1.264   0.396  -0.132 ⎥
C    =   ⎢  2.000  -0.076  -1.585   0.660 ⎥
         ⎢  1.100   1.267   0.243  -0.609 ⎥
         ⎢  0.000   1.010   0.014   0.076 ⎥
         ⎢ -1.000   0.995   0.000   0.005 ⎥
         ⎣                              ⎦

INIT     =  4
S        =  (-2.792, 1.649, 1.100, -2.000)
```

Example 2: This example computes the spline coefficients through a set of data points with a derivative value specified at the right endpoint. It also evaluates the spline at the abscissas specified in T. On output, INIT and C are updated with new values.

Call Statement and Input

```
             X   Y   C   N  INIT  T   S   M
             |   |   |   |   |    |   |   |
CALL SCSINT( X , Y , C , 6 , 2  , T , S , 4 )

X        =  (1.000, 2.000, 3.000, 4.000, 5.000, 6.000)
Y        =  (0.000, 1.000, 2.000, 1.100, 0.000, -1.000)

         ⎡             ⎤
         ⎢  .   .  .  . ⎥
         ⎢ 0.1  .  .  . ⎥
C    =   ⎢  .   .  .  . ⎥
         ⎢  .   .  .  . ⎥
         ⎢  .   .  .  . ⎥
         ⎢  .   .  .  . ⎥
         ⎣             ⎦

T        =  (-1.000, 2.500, 4.000, 7.000)
```

Output

```
        ⎡                                  ⎤
        ⎢  0.000  -0.865   0.000  -0.135 ⎥
        ⎢  1.000  -1.270   0.405  -0.135 ⎥
C    =  ⎢  2.000  -0.054  -1.621   0.675 ⎥
        ⎢  1.100   1.188   0.379  -0.667 ⎥
        ⎢  0.000   1.303  -0.494   0.291 ⎥
        ⎢ -1.000   0.100   1.897  -0.797 ⎥
        ⎣                                  ⎦

INIT     =  4
S        =  (-2.810, 1.652, 1.100, 1.794)
```

Example 3: This example computes the spline coefficients through a set of data points with a derivative value specified at both endpoints. It does not evaluate the spline at any points. On output, `INIT` and `C` are updated with new values. Because arrays are not needed for arguments *t* and *s*, the value 0 is specified in their place.

Call Statement and Input

```
             X   Y   C   N  INIT  T   S   M
             |   |   |   |   |    |   |   |
CALL SCSINT( X , Y , C , 6 , 3  , 0 , 0 , 0 )

X        =  (1.000, 2.000, 3.000, 4.000, 5.000, 6.000)
Y        =  (0.000, 1.000, 2.000, 1.100, 0.000, -1.000)

        ⎡                 ⎤
        ⎢ -1.0  .  .  . ⎥
        ⎢  0.1  .  .  . ⎥
C    =  ⎢   .   .  .  . ⎥
        ⎢   .   .  .  . ⎥
        ⎢   .   .  .  . ⎥
        ⎢   .   .  .  . ⎥
        ⎣                 ⎦
```

Output

```
        ⎡                                  ⎤
        ⎢  0.000   1.000   3.230   1.230 ⎥
        ⎢  1.000  -1.770  -0.460   1.230 ⎥
C    =  ⎢  2.000   0.079  -1.389   0.310 ⎥
        ⎢  1.100   1.152   0.316  -0.568 ⎥
        ⎢  0.000   1.312  -0.476   0.264 ⎥
        ⎢ -1.000  -0.100   1.888  -0.788 ⎥
        ⎣                                  ⎦

INIT     =  4
```

Example 4: This example evaluates the spline at a set of points, using the coefficients obtained in Example 3.

Call Statement and Input

```
             X   Y   C   N  INIT  T   S   M
             |   |   |   |   |    |   |   |
CALL SCSINT( X , Y , C , 6 , 4  , T , S , 4 )

X        =  (1.000, 2.000, 3.000, 4.000, 5.000, 6.000)
Y        =  (0.000, 1.000, 2.000, 1.100, 0.000, -1.000)
C        =(same as output C in Example 3 )
T        =  (-1.000, 2.500, 4.000, 7.000)
```

Output

```
C        =(same as output C in Example 3 )
S        =  (24.762, 1.731, 1.100, 1.776)
INIT     =  4
```

SCSIN2 and DCSIN2—Two-Dimensional Cubic Spline Interpolation

These subroutines compute the interpolation values at a specified set of points, using data defined on a rectangular mesh in the x-y plane.

Table 153. Data Types

x, ***y***, ***Z***, ***t***, ***u***, *aux*, ***S***	**Subroutine**
Short-precision real	SCSIN2
Long-precision real	DCSIN2

Syntax

Fortran	CALL SCSIN2 | DCSIN2 (*x*, *y*, *z*, *n1*, *n2*, *ldz*, *t*, *u*, *m1*, *m2*, *s*, *lds*, *aux*, *naux*)
C and C++	scsin2 | dcsin2 (*x*, *y*, *z*, *n1*, *n2*, *ldz*, *t*, *u*, *m1*, *m2*, *s*, *lds*, *aux*, *naux*);
PL/I	CALL SCSIN2 | DCSIN2 (*x*, *y*, *z*, *n1*, *n2*, *ldz*, *t*, *u*, *m1*, *m2*, *s*, *lds*, *aux*, *naux*);

On Entry

x
is the vector ***x*** of length *n1*, containing the x-coordinates of the data points that define the spline. The elements of ***x*** must be distinct and sorted into ascending order. Specified as: a one-dimensional array of (at least) length *n1*, containing numbers of the data type indicated in Table 153.

y
is the vector ***y*** of length *n2*, containing the y-coordinates of the data points that define the spline. The elements of ***y*** must be distinct and sorted into ascending order. Specified as: a one-dimensional array of (at least) length *n2*, containing numbers of the data type indicated in Table 153.

z
is the matrix **Z**, containing the data at (x_i, y_j) for $i = 1, n1$ and $j = 1, n2$ that defines the spline. Specified as: an *ldz* by (at least) *n2* array, containing numbers of the data type indicated in Table 153.

n1
is the number of elements in vector ***x*** and the number of rows in matrix **Z**—that is, the number of x-coordinates at which the spline is defined. Specified as: a fullword integer; $n1 \geq 0$.

n2
is the number of elements in vector ***y*** and the number of columns in matrix **Z**—that is, the number of y-coordinates at which the spline is defined. Specified as: a fullword integer; $n2 \geq 0$.

ldz
is the leading dimension of the array specified for *z*. Specified as: a fullword integer; $ldz > 0$ and $ldz \geq n1$.

t
is the vector ***t*** of length *m1*, containing the x-coordinates at which the spline is evaluated. Specified as: a one-dimensional array of (at least) length *m1*, containing numbers of the data type indicated in Table 153.

u
is the vector ***u*** of length *m2*, containing the y-coordinates at which the spline is evaluated. Specified as: a one-dimensional array of (at least) length *m2*, containing numbers of the data type indicated in Table 153.

m1
is the number of elements in vector ***t***—that is, the number of x-coordinates at which the spline interpolation is evaluated. Specified as: a fullword integer; $m1 \geq 0$.

m2
is the number of elements in vector ***u***—that is, the number of y-coordinates at which the spline interpolation is evaluated. Specified as: a fullword integer; $m2 \geq 0$.

s
See On Return.

lds
is the leading dimension of the array specified for *s*. Specified as: a fullword integer; $lds > 0$ and $lds \geq m1$.

aux
has the following meaning:

If $naux = 0$ and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is the storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing numbers of the data type indicated in Table 151 on page 912. On output, the contents are overwritten.

naux
is the size of the work area specified by *aux*—that is, the number of elements in *aux*. Specified as: a fullword integer, where:

If $naux = 0$ and error 2015 is unrecoverable, SCSIN2 and DCSIN2 dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, $naux \geq (10)(\max(n1, n2))+(n2+1)(m1)+2(m2)$.

On Return

s
is the matrix ***S*** with elements s_{kh} that contain the interpolation values at (t_k, u_h) for $k = 1, m1$ and $h = 1, m2$. Returned as: an *lds* by (at least) *m2* array, containing numbers of the data type indicated in Table 153 on page 921.

Notes

1. The cyclic reduction method used to solve the equations in this subroutine can generate underflows on well-scaled problems. This does not affect accuracy, but it may decrease performance. For this reason, you may want to disable underflow before calling this subroutine.
2. Vectors ***x***, ***y***, ***t***, and ***u***, matrix ***Z***, and the *aux* work area must have no common elements with matrix ***S***; otherwise, results are unpredictable.
3. The elements within vectors ***x*** and ***y*** must be distinct. In addition, the elements in the vectors must be sorted into ascending order; that is, $x_1 < x_2 < \ldots < x_{n1}$ and $y_1 < y_2 < \ldots < y_{n2}$. Otherwise, results are unpredictable. For details on how to do this, see "ISORT, SSORT, and DSORT—Sort the Elements of a Sequence" on page 888.

4. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: Interpolation is performed at a specified set of points:

(t_k, u_h) for $k = 1, m1$ and $h = 1, m2$

by fitting bicubic spline functions with natural boundary conditions, using the following set of data, defined on a rectangular grid, (x_i, y_j) for $i = 1, n1$ and $j = 1, n2$:

z_{ij} for $i = 1, n1$ and $j = 1, n2$

where t_k, u_h, x_i, y_j, and z_{ij} are elements of vectors ***t***, ***u***, ***x***, and ***y*** and matrix ***Z***, respectively. In vectors ***x*** and ***y***, elements are assumed to be sorted into ascending order.

The interpolation involves two steps:

1. For each *j* from 1 to *n2*, the single variable cubic spline:

   $$s_{y_j}(x)$$

   with natural boundary conditions, is constructed using the data points:

   (x_i, z_{ij}) for $i = 1, n1$

   The following interpolation values are then computed:

   $$s_{y_j}(t_k) \qquad \text{for } k = 1, m1$$

2. For each *k* from 1 to *m1*, the single variable cubic spline:

   $$s_{t_k}(y),$$

   with natural boundary conditions, is constructed using the data points:

   $$\left(y_j, s_{y_j}(t_k)\right) \qquad \text{for } j = 1, n2$$

   The following interpolation values are then computed:

   $$s_{kh} = s_{t_k}(u_h) \qquad \text{for } l = 1, m2$$

See references [51] and [57]. Because natural boundary conditions (zero second derivatives at the end of the ranges) are used for the splines, unless the underlying function has these properties, interpolated values near the boundaries may be less satisfactory than elsewhere. If *n1*, *n2*, *m1*, or *m2* is 0, no computation is performed.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors: None

Input-Argument Errors

1. *n1* < 0 or *n1* > *ldz*
2. *n2* < 0
3. *m1* < 0 or *m1* > *lds*
4. *m2* < 0
5. *ldz* < 0
6. *lds* < 0
7. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value specified in the syntax for this argument. Return code 1 is returned if error 2015 is recoverable.

Example: This example computes the interpolated values at a specified set of points, given by `T` and `U`, from a set of data points defined on a rectangular mesh in the x-y plane, using `X`, `Y`, and `Z`.

Call Statement and Input

```
            X   Y   Z   N1  N2 LDZ  T   U   M1  M2  S  LDS  AUX  NAUX
            |   |   |   |   |   |   |   |   |   |   |   |    |    |
CALL SCSIN2( X , Y , Z , 6 , 5 , 6 , T , U , 4 , 3 , S , 4 , AUX , 90 )

X        =  (0.0, 0.2, 0.3, 0.4, 0.5, 0.7)
Y        =  (0.0, 0.2, 0.3, 0.4, 0.6)

        ┌                                  ┐
        | 0.000  0.008  0.027  0.064  0.216 |
        | 0.008  0.016  0.035  0.072  0.224 |
Z    =  | 0.027  0.035  0.054  0.091  0.243 |
        | 0.064  0.072  0.091  0.128  0.280 |
        | 0.125  0.133  0.152  0.189  0.341 |
        | 0.343  0.351  0.370  0.407  0.559 |
        └                                  ┘

T        =  (0.10, 0.15, 0.25, 0.35)
U        =  (0.05, 0.25, 0.45)
```

Output

```
        ┌                    ┐
        | 0.001  0.017  0.095 |
S    =  | 0.003  0.019  0.097 |
        | 0.016  0.031  0.110 |
        | 0.043  0.059  0.137 |
        └                    ┘
```

Chapter 15. Numerical Quadrature

The numerical quadrature subroutines are described in this chapter.

Overview of the Numerical Quadrature Subroutines

The numerical quadrature subroutines provide Gaussian quadrature methods for integrating a tabulated function and a user-supplied function over a finite, semi-infinite, or infinite region of integration (Table 154).

Table 154. List of Numerical Quadrature Subroutines

Descriptive Name	Short-Precision Subroutine	Long-Precision Subroutine	Page
Numerical Quadrature Performed on a Set of Points	SPTNQ	DPTNQ	929
Numerical Quadrature Performed on a Function Using Gauss-Legendre Quadrature	SGLNQ†	DGLNQ†	932
Numerical Quadrature Performed on a Function Over a Rectangle Using Two-Dimensional Gauss-Legendre Quadrature	SGLNQ2†	DGLNQ2†	935
Numerical Quadrature Performed on a Function Using Gauss-Laguerre Quadrature	SGLGQ†	DGLGQ†	941
Numerical Quadrature Performed on a Function Using Gauss-Rational Quadrature	SGRAQ†	DGRAQ†	945
Numerical Quadrature Performed on a Function Using Gauss-Hermite Quadrature	SGHMQ†	DGHMQ†	949

† This subprogram is invoked as a function in a Fortran program.

Use Considerations

This section contains some key points about using the numerical quadrature subroutines.

Choosing the Method

The theoretical aspects of choosing the method to use for integration can be found in the references [26], [58], and [86].

Performance and Accuracy Considerations

1. There are *n* function evaluations for a method of order *n*. Because function evaluations are expensive in terms of computing time, you should weigh the considerations for computing time and accuracy in choosing a value for *n*.
2. To achieve optimal performance in the _GLNQ2 subroutines, specify the first variable integrated to be the variable having more points. This allows both the

© Copyright IBM Corp. 1997

subroutine and the function evaluation to achieve optimal performance. Details on how to do this are given in "Notes" on page 936.

3. There are some ESSL-specific rules that apply to the results of computations on the workstation processors using the ANSI/IEEE standards. For details, see "What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?" on page 44.

Programming Considerations for the SUBF Subroutine

This section describes how to design and code the *subf* subroutine for use by the numerical quadrature subrutines.

Designing SUBF

For the Gaussian quadrature subroutines, you must supply a separate subroutine that is callable by ESSL. You specify the name of the subroutine in the *subf* argument. This subroutine name is selected by you. You should design the *subf* subroutine so it receives, as input, a tabulated set of points at which the integrand is evaluated, and it returns, as output, the values of the integrand evaluated at these points.

Depending on the numerical quadrature subroutine that you use, the *subf* subroutine is defined in one of the two following ways:

- For _GLNQ, _GLGQ, _GRAQ, and _GHMQ, you define the *subf* subroutine with three arguments: *t*, *y*, and *n*, where:

  t is an input array, referred to as T, of tabulated Gaussian quadrature abscissas, containing *n* real numbers, t_i, where t_i is automatically provided by the ESSL subroutine and is determined by *n* and the Gaussian quadrature method chosen.

  y is an output array, referred to as Y, containing *n* real numbers, where for the integrand, the following is true: $y_i = f(t_i)$ for $i = 1, n$.

  n is a positive integer indicating the number of elements in T and Y.

- For _GLNQ2, you define the *subf* subroutine with six arguments: *s*, *n1*, *t*, *n2*, *z*, and *ldz*, where:

  s is an input array, referred to as S, of tabulated Gaussian quadrature abscissas, containing *n1* real numbers, s_i, where s_i is automatically provided by the ESSL subroutine and is determined by *n1* and the Gaussian quadrature method.

  n1 is a positive integer indicating the number of elements in S and the number of rows to be used in array **Z**.

  t is an input array, referred to as T, of tabulated Gaussian quadrature abscissas, containing *n2* real numbers, t_i, where t_i is automatically provided by the ESSL subroutine and is determined by *n2* and the Gaussian quadrature method.

  n2 is a positive integer indicating the number of elements in T and the number of columns to be used in array **Z**.

  z is an *ldz* by (at least) *n2* output array, referred to as Z, of real numbers, where for the integrand, the following is true: $z_{ij} = f(s_i, t_j)$ for $i = 1, n1$ and $j = 1, n2$.

ldz is a positive integer indicating the size of the leading dimension of the array **Z**.

Coding and Setting Up SUBF in Your Program

Examples of coding a *subf* subroutine in Fortran are provided for each subroutine in this chapter. Examples of coding a *subf* subroutine in C, C++, and PL/I are provided in "Example 1" on page 937.

Depending on the programming language you use for your program that calls the numerical quadrature subroutines, you have a choice of one or more languages that you can use for writing *subf*. These rules and other langauge-related coding rules for setting up *subf* in your program are described in the following sections:

- "Setting Up a User-Supplied Subroutine for ESSL in Fortran" on page 111
- "Setting Up a User-Supplied Subroutine for ESSL in C" on page 132
- "Setting Up a User-Supplied Subroutine for ESSL in C++" on page 147

Numerical Quadrature Subroutines

This section contains the numerical quadrature subroutine descriptions.

SPTNQ and DPTNQ—Numerical Quadrature Performed on a Set of Points

These subroutines approximate the integral of a real valued function specified in tabular form, (x_i, y_i) for $i = 1, n$. For more than four points, an error estimate is returned along with the resulting value.

Table 155. Data Types

x, ***y***, *xyint*, *eest*	**Subroutine**
Short-precision real	SPTNQ
Long-precision real	DPTNQ

Syntax

Fortran	CALL SPTNQ | DPTNQ (*x*, *y*, *n*, *xyint*, *eest*)
C and C++	sptnq | dptnq (*x*, *y*, *n*, *xyint*, *eest*);
PL/I	CALL SPTNQ | DPTNQ (*x*, *y*, *n*, *xyint*, *eest*);

On Entry

x
is the vector ***x*** of length *n*, containing the abscissas of the data points to be integrated. The elements of ***x*** must be distinct and sorted into ascending or descending order. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 155.

y
is the vector ***y*** of length *n*, containing the ordinates of the data points to be integrated. Specified as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 155.

n
is the number of elements in vectors ***x*** and ***y***—that is, the number of data points. The value of *n* determines the algorithm used by this subroutine. For details, see "Function" on page 930. Specified as: a fullword integer; $n \geq 2$.

xyint
See On Return.

eest
See On Return.

On Return

xyint
is the approximation *xyint* of the integral. Returned as: a number of the data type indicated in Table 155.

eest
has the following meaning, where:

If $n < 5$, it is undefined and is set to 0.

If $n \geq 5$, it is an estimate, *eest*, of the error in the integral, where *xyint*+*eest* tends to give a better approximation to the integral than *xyint*. For details, see references [26] and [58].

Returned as: a number of the data type indicated in Table 155.

Notes

1. In your C program, arguments *xyint* and *eest* must be passed by reference.
2. The elements of vector ***x*** must be distinct—that is, $x_i \neq x_j$ for $i \neq j$,—and they must be sorted into ascending or descending order; otherwise, results are unpredictable. For how to do this, see "ISORT, SSORT, and DSORT—Sort the Elements of a Sequence" on page 888.

Function: The integral is approximated for a real valued function specified in tabular form, (x_i, y_i) for $i = 1, n$, where x_i are distinct and sorted into ascending or descending order, and $n \geq 2$. If $y_i = f(x_i)$ for $i = 1, n$, then on output, *xyint* is an approximation to the integral of the following form:

$$\int_{x_1}^{x_n} f(x)dx$$

The algorithm used by this subroutine is based on the number of data points used in the computation, where:

- If $n = 2$, the trapezoid rule is used to do the integration.
- If $n = 3$, the parabola through the three points is integrated.
- If $n \geq 4$, the method of Gill and Miller is used to do the integration.

For $n \geq 5$, an estimate of the error *eest* is returned. For the method of Gill and Miller, it is shown that adding the estimate of the error *eest* to the result *xyint* often gives a better approximation to the integral than the result *xyint* by itself. For $n < 5$, an estimate of the error is not returned. In this case, a value of 0 is returned for *eest*. See references [58] and [26].

Error Conditions

Computational Errors: None

Input-Argument Errors: $n < 2$

Example 1: This example shows the result of an integration, where the abscissas in `X` are sorted into ascending order.

Call Statement and Input

```
            X   Y   N    XYINT   EEST
            |   |   |      |       |
CALL SPTNQ( X , Y , 10 , XYINT , EEST )

X       =  (0.0, 0.4, 1.0, 1.5, 2.1, 2.6, 3.0, 3.4, 3.9, 4.3)
Y       =  (1.0, 2.0, 3.0, 4.0, 5.0, 4.5, 4.0, 3.0, 3.5, 3.3)
```

Output

```
XYINT    =  15.137
EEST     =  -0.003
```

Example 2: This example shows the result of an integration, where the abscissas in X are sorted into descending order.

Call Statement and Input

```
            X   Y   N    XYINT   EEST
            |   |   |      |      |
CALL SPTNQ( X , Y , 10 , XYINT , EEST )

X        =  (4.3, 3.9, 3.4, 3.0, 2.6, 2.1, 1.5, 1.0, 0.4, 0.0)
Y        =  (3.3, 3.5, 3.0, 4.0, 4.5, 5.0, 4.0, 3.0, 2.0, 1.0)
```

Output

```
XYINT    =  -15.137
EEST     =  0.003
```

SGLNQ and DGLNQ—Numerical Quadrature Performed on a Function Using Gauss-Legendre Quadrature

These functions approximate the integral of a real valued function over a finite interval, using the Gauss-Legendre Quadrature method of specified order.

Table 156. Data Types

a, *b*, **Result**	**Subroutine**
Short-precision real	SGLNQ
Long-precision real	DGLNQ

Syntax

Fortran	SGLNQ | DGLNQ (*subf*, *a*, *b*, *n*)
C and C++	sglnq | dglnq (*subf*, *a*, *b*, *n*);
PL/I	SGLNQ | DGLNQ (*subf*, *a*, *b*, *n*);

On Entry

subf
is the user-supplied subroutine that evaluates the integrand function. The subroutine should be defined with three arguments: *t*, *y*, and *n*. For details, see " Programming Considerations for the SUBF Subroutine" on page 926.

Specified as: *subf* must be declared as an external subroutine in you application program. It can be whatever name you choose.

a
is the lower limit of integration, *a*. Specified as: a number of the data type indicated in Table 156.

b
is the upper limit of integration, *b*. Specified as: a number of the data type indicated in Table 156.

n
is the order of the quadrature method to be used. Specified as: a fullword integer; *n* = 1, 2, 3, 4, 5, 6, 8, 10, 12, 14, 16, 20, 24, 32, 40, 48, 64, 96, 128, or 256.

On Return

Function value

is the approximation of the integral. Returned as: a number of the data type indicated in Table 156.

Notes

1. Declare the DGLNQ function in your program as returning a long-precision real number. Declare the SGLNQ, if necessary, as returning a short-precision real number.
2. The subroutine specified for *subf* must be declared as external in your program. Also, data types used by *subf* must agree with the data types specified by this ESSL subroutine. The variable *x*, described under "Function" on page 933, and the argument *n* correspond to the *subf* arguments *t* and *n*, respectively. For details on

how to set up the subroutine, see " Programming Considerations for the SUBF Subroutine" on page 926.

Function: The integral is approximated for a real valued function over a finite interval, using the Gauss-Legendre Quadrature method of specified order. The region of integration is from *a* to *b*. The method of order *n* is theoretically exact for integrals of the following form, where *f* is a polynomial of degree less than 2*n*:

$$\int_a^b f(x)dx$$

The method of order *n* is a good approximation when your integrand is closely approximated by a function of the form $f(x)$, where *f* is a polynomial of degree less than 2*n*. See references [26] and [86]. The result is returned as the function value.

Error Conditions

Computational Errors: None

Input-Argument Errors: *n* is not an allowable value, as listed in the syntax for this argument.

Example: This example shows how to compute the integral of the function *f* given by:

$f(x) = x^2+e^x$

over the interval (0.0, 2.0), using the Gauss-Legendre method with 10 points:

$$\int_{0.0}^{2.0}\left(x^2+e^x\right)dx$$

The user-supplied subroutine FUN1, which evaluates the integrand function, is coded in Fortran as follows:

```
   SUBROUTINE FUN1 (T,Y,N)
   INTEGER*4 N
   REAL*4 T(*),Y(*)
   DO 1 I=1,N
1     Y(I)=T(I)**2+EXP(T(I))
   RETURN
   END
```

Program Statements and Input

```
EXTERNAL FUN1
        .
        .
        .
              SUBF    A     B    N
               |      |     |    |
XINT = SGLNQ( FUN1 , 0.0 , 2.0 , 10 )
        .
        .
        .

FUN1     =(see above)
```

Output

```
XINT     =  9.056
```

SGLNQ2 and DGLNQ2—Numerical Quadrature Performed on a Function Over a Rectangle Using Two-Dimensional Gauss-Legendre Quadrature

These functions approximate the integral of a real valued function of two variables over a rectangular region, using the Gauss-Legendre Quadrature method of specified order in each variable.

Table 157. Data Types

a, *b*, *c*, *d*, **Z**, **Result**	**Subroutine**
Short-precision real	SGLNQ2
Long-precision real	DGLNQ2

Syntax

Fortran	SGLNQ2 | DGLNQ2 (*subf*, *a*, *b*, *n1*, *c*, *d*, *n2*, *z*, *ldz*)
C and C++	sglnq2 | dglnq2 (*subf*, *a*, *b*, *n1*, *c*, *d*, *n2*, *z*, *ldz*);
PL/I	SGLNQ2 | DGLNQ2 (*subf*, *a*, *b*, *n1*, *c*, *d*, *n2*, *z*, *ldz*);

On Entry

subf
is the user-supplied subroutine that evaluates the integrand function. The subroutine should be defined with six arguments: *s*, *n1*, *t*, *n2*, *z*, and *ldz*. For details, see " Programming Considerations for the SUBF Subroutine" on page 926.

Specified as: *subf* must be declared as an external subroutine in your application program. It can be whatever name you choose.

a
is the lower limit of integration, *a*, for the first variable integrated. Specified as: a number of the data type indicated in Table 157.

b
is the upper limit of integration, *b*, for the first variable integrated. Specified as: a number of the data type indicated in Table 157.

n1
is the order of the quadrature method to be used for the first variable integrated. Specified as: a fullword integer; *n1* = 1, 2, 3, 4, 5, 6, 8, 10, 12, 14, 16, 20, 24, 32, 40, 48, 64, 96, 128, or 256.

c
is the lower limit of integration, *c*, for the second variable integrated. Specified as: a number of the data type indicated in Table 157.

d
is the upper limit of integration, *d*, for the second variable integrated. Specified as: a number of the data type indicated in Table 157.

n2
is the order of the quadrature method to be used for the second variable integrated. Specified as: a fullword integer; *n2* = 1, 2, 3, 4, 5, 6, 8, 10, 12, 14, 16, 20, 24, 32, 40, 48, 64, 96, 128, or 256.

z
is the matrix **Z**, containing the *n1* rows and *n2* columns of data used to evaluate the integrand function. (The output values from the *subf* subroutine are placed in **Z**.) Specified as: an *ldz* by (at least) *n2* array, containing numbers of the data type indicated in Table 157.

ldz
is the size of the leading dimension of the array specified for *z*. Specified as: a fullword integer; $ldz > 0$ and $ldz \geq n1$.

On Return

Function value
is the approximation of the integral. Returned as: a number of the data type indicated in Table 157 on page 935.

Notes

1. Declare the DGLNQ2 function in your program as returning a long-precision real number. Declare the SGLNQ2 function, if necessary, as returning a short-precision real number.
2. The subroutine specified for *subf* must be declared as external in your program. Also, data types used by *subf* must agree with the data types specified by this ESSL subroutine. For details on how to set up the subroutine, see " Programming Considerations for the SUBF Subroutine" on page 926.

Function: The integral:

$$\int_c^d \int_a^b f(s,t)\, ds\, dt$$

is approximated for a real valued function of two variables *s* and *t*, over a rectangular region, using the Gauss-Legendre Quadrature method of specified order in each variable. The region of integration is:

(*a*, *b*) for *s*
(*c*, *d*) for *t*

The method gives a good approximation when your integrand is closely approximated by a function of the form $f(s, t)$, where *f* is a polynomial of degree less than $2(n1)$ for *s* and $2(n2)$ for *t*. See the function description for "SGLNQ and DGLNQ—Numerical Quadrature Performed on a Function Using Gauss-Legendre Quadrature" on page 932 and references [26] and [86]. The result is returned as the function value.

Special Usage: To achieve optimal performance in this subroutine and in the functional evaluation, specify the first variable integrated in this subroutine as the variable having more points. The first variable integrated is the variable in the inner integral. For example, in the following integration, *x* is the first variable integrated:

$$\int_{u1}^{u2} \int_{r1}^{r2} f(x,y)\, dx\, dy$$

This is the suggested order of integration if the *x* variable has more points than the *y* variable. On the other hand, if the *y* variable has more points, you make *y* the first variable integrated.

Because the order of integration does not matter to the resulting approximation, you may be able to reverse the order that *x* and *y* are integrated and get better performance. This can be expressed as:

$$\int_{u1}^{u2}\int_{r1}^{r2} f(x,y)\,dx\,dy \quad = \quad \int_{r1}^{r2}\int_{u1}^{u2} f(x,y)\,dy\,dx$$

Results are mathematically equivalent. However, because the algorithm is computed in a different way, results may not be bitwise identical.

Table 158 shows how to assign your variables to the _GLNQ2 and *subf* arguments for the *x*-*y* integration shown on the left and for the *y*-*x* integration shown on the right. For examples of how to do each of these, see "Example 1" and "Example 2" on page 940.

Table 158. How to Assign Your Variables for x-y Integration Versus y-x Integration

_GLNQ2 and SUBF Arguments	**Variables for *x*-*y* Integration**	**Variables for *y*-*x* Integration**
For _GLNQ2:		
a	*r1*	*u1*
b	*r2*	*u2*
n1	(order for *x*)	(order for *y*)
c	*u1*	*r1*
d	*u2*	*r2*
n2	(order for *y*)	(order for *x*)
For *subf*:		
s	*x*	*y*
t	*y*	*x*
n1	(order for *x*)	(order for *y*)
n2	(order for *y*)	(order for *x*)

Error Conditions

Computational Errors: None

Input-Argument Errors

1. $ldz \le 0$
2. $n1 > ldz$
3. *n1* or *n2* is not an allowable value, as listed in the syntax for this argument.

Example 1: This example shows how to compute the integral of the function *f* given by:

$f(x, y) = e^x \sin y$

over the intervals (0.0, 2.0) for the first variable *x* and (−2.0, −1.0) for the second variable *y*, using the Gauss-Legendre method with 10 points in the *x* variable and 5 points in the *y* variable:

$$\int_{-2.0}^{-1.0}\int_{0.0}^{2.0}\left(e^{x}\ \sin\ y\right)dx\ dy$$

Because the variable *x* has more points, it is the first variable integrated. This allows the SGLNQ2 subroutine and the FUN1 evaluation to achieve optimal performance. Therefore, the *x* and *y* variables correspond to S and T in the FUN1 subroutine. Also, the *x* and *y* variables correspond to the A, B, N1 and C, D, N2 sets of arguments, respectively, for SGLNQ2.

Using Fortran for SUBF: The user-supplied subroutine FUN1, which evaluates the integrand function, is coded in Fortran as follows:

```
      SUBROUTINE FUN1 (S,N1,T,N2,Z,LDZ)
      INTEGER*4 N1,N2,LDZ
      REAL*4 S(*),T(*),Z(LDZ,*)
      DO 1 J=1,N2
      DO 2 I=1,N1
2        Z(I,J)=EXP(S(I))*SIN(T(J))
1        CONTINUE
      RETURN
      END
```

Note: The computation for this user-supplied subroutine FUN1 can also be performed by using the following statements in place of the above DO loops, using T1 and T2 as temporary storage areas:

```
         .
         .
         .
      DO 1 I=1,N1
1        T1(I)=EXP(S(I))
      DO 2 J=1,N2
2        T2(J)=SIN(T(J))
      DO 3 J=1,N2
      DO 4 I=1,N1
4        Z(I,J)=T1(I)*T2(J)
3        CONTINUE
         .
         .
         .
```

When coding your application, this is the preferred technique. It reduces the number of evaluations performed and, therefore, provides better performance.

Using C for SUBF: The user-supplied subroutine FUN1, which evaluates the integrand function, is coded in C as follows:

```
void fun1(s, n1, t, n2, z, ldz)
float *s, *t, *z;
int *n1, *n2, *ldz;
{
     int i, j;
     for(j = 0; j < *n2; ++j, z += *ldz)
         {
          for(i = 0; i < *n1; ++i)
          z[i] = exp(s[i]) * sin(t[j]);
         }
}
```

Using C++ for SUBF: The user-supplied subroutine FUN1, which evaluates the integrand function, is coded in C++ as follows:

```
void fun1(float *s, int *n1, float *t, int *n2, float *z, int *ldz)
{
     int i, j;
     for(j = 0; j < *n2; ++j, z += *ldz)
         {
         for(i = 0; i < *n1; ++i)
          z[i] = exp(s[i]) * sin(t[j]);
         }
}
```

Using PL/I for SUBF: The user-supplied subroutine FUN1, which evaluates the integrand function, is coded in PL/I as follows:

```
FUN1: PROCEDURE(S,N1,T,N2,Z,LDZ) OPTIONS(FORTRAN,NOMAP);
DCL (N1,N2,LDZ,I,J) REAL FIXED BINARY(31,0);
DCL (S(10),T(10),Z(5,10)) REAL FLOAT DEC(16) ALIGNED CONNECTED;
DO J=1 TO N1;
   DO I=1 TO N2;
      Z(I,J)=EXP(S(J))*SIN(T(I));
   END;
END;
RETURN;
END FUN1;
```

Program Statements and Input

```
EXTERNAL FUN1
        .
        .
        .
                SUBF    A     B    N1     C      D    N2  Z   LDZ
                 |      |     |    |      |      |    |   |    |
XYINT = SGLNQ2( FUN1 , 0.0 , 2.0 , 10 , -2.0 , -1.0 , 5 , Z , 10 )
        .
        .
        .
```

```
FUN1      =(see sections above)
Z         =(not relevant)
```

Output

```
XYINT    =  -6.1108
```

Example 2: This example shows how to reverse the order of integration of the variables *x* and *y*. It computes the integral of the function *f* given by:

$$f(x, y) = \cos x \sin y$$

over the intervals (0.0, 1.0) for the variable *x* and (0.0, 20.0) for the variable *y*, using the Gauss-Legendre method with 5 points in the *x* variable and 48 points in the *y* variable. Because the order of integration does not matter to the approximation:

$$\int_{0.0}^{20.0}\int_{0.0}^{1.0}(\cos\ x\ \sin\ y)\,dx\,dy \quad = \quad \int_{0.0}^{1.0}\int_{0.0}^{20.0}(\cos\ x\ \sin\ y)\,dy\,dx$$

the variable *y*, having more points, is the first variable integrated (performing the integration shown on the right.) This allows the SGLNQ2 subroutine and the FUN1 evaluation to achieve optimal performance. Therefore, the *x* and *y* variables correspond to T and S in the FUN2 subroutine. Also, the *x* and *y* variables correspond to the C, D, N2 and A, B, N1 sets of arguments, respectively, for SGLNQ2.

The user-supplied subroutine FUN2, which evaluates the integrand function, is coded in Fortran as follows:

```
   SUBROUTINE FUN2 (S,N1,T,N2,Z,LDZ)
   INTEGER*4 N1,N2,LDZ
   REAL*4 S(*),T(*),Z(LDZ,*)
   DO 1 J=1,N2
   DO 2 I=1,N1
2     Z(I,J)=COS(T(J))*SIN(S(I))
1     CONTINUE
   RETURN
   END
```

Note: The same coding principles for achieving good performance that are noted in "Example 1" on page 937 also apply to this user-supplied subroutine FUN2.

Program Statements and Input

```
EXTERNAL FUN2.
        .
        .
        .
                SUBF    A      B    N1    C     D    N2   Z  LDZ
                 |      |      |    |     |     |    |    |   |
YXINT = SGLNQ2( FUN2 , 0.0 , 20.0 , 48 , 0.0 , 1.0 , 5  , Z , 48 )
        .
        .
        .

FUN2      =(see above)
Z         =(not relevant)
```

Output

```
YXINT    =  0.4981
```

SGLGQ and DGLGQ—Numerical Quadrature Performed on a Function Using Gauss-Laguerre Quadrature

These functions approximate the integral of a real valued function over a semi-infinite interval, using the Gauss-Laguerre Quadrature method of specified order.

Table 159. Data Types

a, *b*, **Result**	**Subroutine**
Short-precision real	SGLGQ
Long-precision real	DGLGQ

Syntax

Fortran	SGLGQ | DGLGQ (*subf*, *a*, *b*, *n*)
C and C++	sglgq | dglgq (*subf*, *a*, *b*, *n*);
PL/I	SGLGQ | DGLGQ (*subf*, *a*, *b*, *n*);

On Entry

subf
is the user-supplied subroutine that evaluates the integrand function. The subroutine should be defined with three arguments: *t*, *y*, and *n*. For details, see " Programming Considerations for the SUBF Subroutine" on page 926.

Specified as: *subf* must be declared as an external subroutine in your application program. It can be whatever name you choose.

a
has the following meaning, where:

If $b > 0$, it is the lower limit of integration.

If $b < 0$, it is the upper limit of integration.

Specified as: a number of the data type indicated in Table 159.

b
is the scaling constant *b* for the exponential. Specified as: a number of the data type indicated in Table 159; $b > 0$ or $b < 0$.

n
is the order of the quadrature method to be used. Specified as: a fullword integer; $n = 1, 2, 3, 4, 5, 6, 8, 10, 12, 14, 16, 20, 24, 32, 40, 48$, or 64.

On Return

Function value
is the approximation of the integral. Returned as: a number of the data type indicated in Table 159.

Notes

1. Declare the DGLGQ function in your program as returning a long-precision real number. Declare the SGLGQ function, if necessary, as returning a short-precision real number.

2. The subroutine specified for *subf* must be declared as external in your program. Also, data types used by *subf* must agree with the data types specified by this ESSL subroutine. The variable *x*, described under "Function" on page 942, and the argument *n* correspond to the *subf* arguments *t* and *n*, respectively. For details on how to set up the subroutine, see " Programming Considerations for the SUBF Subroutine" on page 926.

Function: The integral is approximated for a real valued function over a semi-infinite interval, using the Gauss-Laguerre Quadrature method of specified order. The region of integration is:

(a, ∞) if $b > 0$
$(-\infty, a)$ if $b < 0$

The method of order *n* is theoretically exact for integrals of the following form, where *f* is a polynomial of degree less than $2n$:

$$\int_a^{\infty} f(x)e^{-bx}dx \qquad \text{if } b > 0$$

$$\int_{-\infty}^{a} f(x)e^{-bx}dx \qquad \text{if } b < 0$$

The method of order *n* is a good approximation when your integrand is closely approximated by a function of the form $f(x)e^{-bx}$, where *f* is a polynomial of degree less than $2n$. See references [26] and [86]. The result is returned as the function value.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. $b = 0$
2. *n* is not an allowable value, as listed in the syntax for this argument.

Example 1: This example shows how to compute the integral of the function *f* given by:

$f(x) = \sin(3.0x)e^{-1.5x}$

over the interval $(-2.0, \infty)$, using the Gauss-Laguerre method with 20 points:

$$\int_{-2.0}^{\infty} \left(\sin(3.0x)e^{-1.5x}\right)dx$$

The user-supplied subroutine `FUN1`, which evaluates the integrand function, is coded in Fortran as follows:

```
   SUBROUTINE FUN1 (T,Y,N)
   INTEGER*4 N
   REAL*4 T(*),Y(*)
   DO 1 I=1,N
1     Y(I)=SIN(3.0*T(I))*EXP(-1.5*T(I))
   RETURN
   END
```

Program Statements and Input

```
EXTERNAL FUN1
        .
        .
        .
              SUBF     A     B    N
               |       |     |    |
XINT = SGLGQ( FUN1 , -2.0 , 1.5 , 20 )
        .
        .
        .
```

FUN1 =(see above)

Output

```
XINT     =  5.891
```

Example 2: This example shows how to compute the integral of the function f given by:

$$f(x) = \sin(3.0x)e^{1.5x}$$

over the interval $(-\infty, -2.0)$, using the Gauss-Laguerre method with 20 points:

$$\int_{-\infty}^{-2.0} \sin(3.0x)e^{1.5x}dx$$

The user-supplied subroutine FUN2, which evaluates the integrand function, is coded in Fortran as follows:

```
   SUBROUTINE FUN2 (T,Y,N)
   INTEGER*4 N
   REAL*4 T(*),Y(*),TEMP
   DO 1 I=1,N
1     Y(I)=SIN(3.0*T(I))*EXP(1.5*T(I))
   RETURN
   END
```

Program Statements and Input

```
EXTERNAL FUN2
         .
         .
         .
               SUBF     A      B    N
                |       |      |    |
XINT = SGLGQ( FUN2 , -2.0 , -1.5 , 20 )
         .
         .
         .
FUN2     =  (see above)
```

Output

```
XINT     =  -0.011
```

SGRAQ and DGRAQ—Numerical Quadrature Performed on a Function Using Gauss-Rational Quadrature

These functions approximate the integral of a real valued function over a semi-infinite interval, using the Gaussian-Rational quadrature method of specified order.

Table 160. Data Types

a, *b*, **Result**	**Subroutine**
Short-precision real	SGRAQ
Long-precision real	DGRAQ

Syntax

Fortran	SGRAQ | DGRAQ (*subf*, *a*, *b*, *n*)
C and C++	sgraq | dgraq (*subf*, *a*, *b*, *n*);
PL/I	SGRAQ | DGRAQ (*subf*, *a*, *b*, *n*);

On Entry

subf
is the user-supplied subroutine that evaluates the integrand function. The subroutine should be defined with three arguments: *t*, *y*, and *n*. For details, see " Programming Considerations for the SUBF Subroutine" on page 926.

Specified as: *subf* must be declared as an external subroutine in your application program. It can be whatever name you choose.

a
has the following meaning, where:

If $a+b > 0$, it is the lower limit of integration.

If $a+b < 0$, it is the upper limit of integration.

Specified as: a number of the data type indicated in Table 160.

b
is the centering constant *b* for the integrand. Specified as: a number of the data type indicated in Table 160.

n
is the order of the quadrature method to be used. Specified as: a fullword integer; $n = 1, 2, 3, 4, 5, 6, 8, 10, 12, 14, 16, 20, 24, 32, 40, 48, 64, 96, 128$, or 256.

On Return

Function value

is the approximation of the integral. Returned as: a number of the data type indicated in Table 160.

Notes

1. Declare the DGRAQ function in your program as returning a long-precision real number. Declare the SGRAQ function, if necessary, as returning a short-precision real number.

2. The subroutine specified for *subf* must be declared as external in your program. Also, data types used by *subf* must agree with the data types specified by this ESSL subroutine. The variable *x*, described under "Function" on page 946, and the argument *n* correspond to the *subf* arguments *t* and *n*, respectively. For details on how to set up the subroutine, see " Programming Considerations for the SUBF Subroutine" on page 926.

Function: The integral is approximated for a real valued function over a semi-infinite interval, using the Gauss-Rational quadrature method of specified order. The region of integration is:

(a, ∞) if $a+b > 0$
$(-\infty, a)$ if $a+b < 0$

The method of order *n* is theoretically exact for integrals of the following form, where *f* is a polynomial of degree less than 2*n*:

$$\int_a^{\infty} f\left(\frac{1}{x+b}\right)\frac{1}{(x+b)^2}\,dx \qquad \text{if } a+b>0$$

$$\int_{-\infty}^{a} f\left(\frac{1}{x+b}\right)\frac{1}{(x+b)^2}\,dx \qquad \text{if } a+b<0$$

The method of order *n* is a good approximation when your integrand is closely approximated by a function of the following form, where *f* is a polynomial of degree less than 2*n*:

$$f\left(\frac{1}{x+b}\right)\frac{1}{(x+b)^2}$$

See references [26] and [86]. The result is returned as the function value to a Fortran, C, C++, or PL/I program.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. $a+b = 0$
2. *n* is not an allowable value, as listed in the syntax for this argument.

Example 1: This example shows how to compute the integral of the function *f* given by:

$f(x) = (e^{1.0}/x) / x^2$

over the interval $(-\infty, -2.0)$, using the Gauss-Rational method with 10 points:

$$\int_{-\infty}^{-2.0} \left(\frac{e^{1.0/x}}{x^2} \right) dx$$

The user-supplied subroutine FUN1, which evaluates the integrand function, is coded in Fortran as follows:

```
   SUBROUTINE FUN1 (T,Y,N)
   INTEGER*4 N
   REAL*4 T(*),Y(*),TEMP
   DO 1 I=1,N
        TEMP=1.0/T(I)
1       Y(I)=EXP(TEMP)*TEMP**2
   RETURN
   END
```

Program Statements and Input

```
EXTERNAL FUN1
        .
        .
        .
              SUBF     A     B    N
               |       |     |    |
XINT = SGRAQ( FUN1 , -2.0 , 0.0 , 10 )
        .
        .
        .
```

FUN1 =(see above)

Output

```
XINT     =  0.393
```

Example 2: This example shows how to compute the integral of the function f given by:

$f(x) = (x-3.0)^{-2} + 10(x-3.0)^{-11}$

over the interval (4.0, ∞), using the Gauss-Rational method with 6 points:

$$\int_{4.0}^{\infty} \left((x-3.0)^{-2} + 10(x-3.0)^{-11} \right) dx$$

The user-supplied subroutine FUN2, which evaluates the integrand function, is coded in Fortran as follows:

```
   SUBROUTINE FUN2 (T,Y,N)
   INTEGER*4 N
   REAL*4 T(*),Y(*),TEMP
   DO 1 I=1,N
        TEMP=1.0/(T(I)-3.0)
1       Y(I)=TEMP**2+10.0*TEMP**11
   RETURN
   END
```

Program Statements and Input

```
EXTERNAL FUN2
        .
        .
        .
              SUBF    A      B    N
               |      |      |    |
XINT = SGRAQ( FUN2 , 4.0 , -3.0 , 6 )
        .
        .
        .
FUN2     =  (see above)
```

Output

```
XINT     =  2.00
```

SGHMQ and DGHMQ—Numerical Quadrature Performed on a Function Using Gauss-Hermite Quadrature

These functions approximate the integral of a real valued function over the entire real line, using the Gauss-Hermite Quadrature method of specified order.

Table 161. Data Types

a, *b*, **Result**	**Subroutine**
Short-precision real	SGHMQ
Long-precision real	DGHMQ

Syntax

Fortran	SGHMQ | DGHMQ (*subf*, *a*, *b*, *n*)
C and C++	sghmq | dghmq (*subf*, *a*, *b*, *n*);
PL/I	SGHMQ | DGHMQ (*subf*, *a*, *b*, *n*);

On Entry

subf
is the user-supplied subroutine that evaluates the integrand function. The subroutine should be defined with three arguments: *t*, *y*, and *n*. For details, see " Programming Considerations for the SUBF Subroutine" on page 926.

Specified as: *subf* must be declared as an external subroutine in your application program. It can be whatever name you choose.

a
is the centering constant *a* for the exponential. Specified as: a number of the data type indicated in Table 161.

b
is the scaling constant *b* for the exponential. Specified as: a number of the data type indicated in Table 161; $b > 0$.

n
is the order of the quadrature method to be used. Specified as: a fullword integer; $n = 1, 2, 3, 4, 5, 6, 8, 10, 12, 14, 16, 20, 24, 32, 40, 48, 64$, or 96.

On Return

Function value

is the approximation of the integral. Returned as: a number of the data type indicated in Table 161.

Notes

1. Declare the DGHMQ function in your program as returning a long-precision real number. Declare the SGHMQ function, if necessary, as returning a short-precision real number.
2. The subroutine specified for *subf* must be declared as external in your program. Also, data types used by *subf* must agree with the data types specified by this ESSL subroutine. The variable *x*, described under "Function" on page 950, and the

argument *n* correspond to the *subf* arguments *t* and *n*, respectively. For details on how to set up the subroutine, see " Programming Considerations for the SUBF Subroutine" on page 926.

Function: The integral is approximated for a real valued function over the entire real line, using the Gauss-Hermite Quadrature method of specified order. The region of integration is from $-\infty$ to ∞. The method of order *n* is theoretically exact for integrals of the following form, where *f* is a polynomial of degree less than 2*n*:

$$\int_{-\infty}^{\infty} f(x)e^{-b(x-a)^2}dx$$

The method of order *n* is a good approximation when your integrand is closely approximated by a function of the following form, where *f* is a polynomial of degree less than 2*n*:

$$f(x)e^{-b(x-a)^2}$$

See references [26] and [86]. The result is returned as the function value to a Fortran, C, C++, or PL/I program.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. $b \leq 0$
2. *n* is not an allowable value, as listed in the syntax for this argument.

Example: This example shows how to compute the integral of the function *f* given by:

$$f(x) = x^2e^{-2(x+5.0)^2}$$

over the interval $(-\infty, \infty)$, using the Gauss-Hermite method with 4 points:

$$\int_{-\infty}^{\infty}\left(x^2e^{-2(x+5.0)^2}\right)dx$$

The user-supplied subroutine FUN1, which evaluates the integrand function, is coded in Fortran as follows:

```
   SUBROUTINE FUN1 (T,Y,N)
   INTEGER*4 N
   REAL*4 T(*),Y(*)
   DO 1 I=1,N
1     Y(I)=T(I)**2*EXP(-2.0*(T(I)+5.0)**2)
   RETURN
   END
```

Program Statements and Input

```
EXTERNAL FUN1
        .
        .
        .
              SUBF     A     B    N
               |       |     |    |
XINT = SGHMQ( FUN1 , -5.0 , 2.0 , 4 )
        .
        .
        .
```

FUN1 =(see above)

Output

```
XINT     =  31.646
```

Chapter 16. Random Number Generation

The random number generation subroutines are described in this chapter.

Overview of the Random Number Generation Subroutines

Random number generation subroutines generate uniformly distributed random numbers or normally distributed random numbers (Table 162).

Table 162. List of Random Number Generation Subroutines

Descriptive Name	Short-Precision Subroutine	Long-Precision Subroutine	Page
Generate a Vector of Uniformly Distributed Random Numbers	SURAND	DURAND	954
Generate a Vector of Normally Distributed Random Numbers	SNRAND	DNRAND	957
Generate a Vector of Long Period Uniformly Distributed Random Numbers	SURXOR	DURXOR	961

Use Considerations

If you need a very long period random number generator, you should use SURXOR and DURXOR, rather than SURAND and DURAND, respectively. The very long period of the generator used by SURXOR and DURXOR, $2^{1279}-1$, makes it useful in modern statistical simulations in which the shorter period of other generators can be exhausted during a single run. As a result, if you need a large number of random numbers, you can use these subroutines, because with this generator, you are not be requesting more than a small percentage of the entire period of the generator.

Random Number Generation Subroutines

This section contains the random number generation subroutine descriptions.

© Copyright IBM Corp. 1997

SURAND and DURAND—Generate a Vector of Uniformly Distributed Random Numbers

These subroutines generate vector ***x*** of uniform (0,1) pseudo-random numbers, using the multiplicative congruential method with a user-specified seed.

Table 163. Data Types

x	*seed*	**Subroutine**
Short-precision real	Long-precision real	SURAND
Long-precision real	Long-precision real	DURAND

Note: If you need a very long period random number generator, use SURXOR and DURXOR instead of these subroutines.

Syntax

Fortran	CALL SURAND | DURAND (*seed*, *n*, *x*)
C and C++	surand | durand (*seed*, *n*, *x*);
PL/I	CALL SURAND | DURAND (*seed*, *n*, *x*);

On Entry

seed
is the initial value used to generate the random numbers. Specified as: a number of the data type indicated in Table 163. It should be a whole number; that is, the fraction part should be 0. (If you specify a mixed number, it is truncated.) Its value must be $1.0 \le seed < (2147483647.0 = 2^{31}-1)$.

Note: *seed* is always a long-precision real number, even in SURAND.

n
is the number of random numbers to be generated. Specified as: a fullword integer; $n \ge 0$.

x
See On Return.

On Return

seed
is the new seed that is to be used to generate additional random numbers in subsequent invocations of SURAND or DURAND. Returned as: a number of the data type indicated in Table 163. It is a whole number whose value is $1.0 \le seed < (2147483647.0 = 2^{31}-1)$.

x
is a vector of length *n*, containing the uniform pseudo-random numbers with values between 0 and 1. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 163.

Note: In your C program, argument *seed* must be passed by reference.

Function: The uniform (0,1) pseudo-random numbers are generated as follows, using the multiplicative congruential method:

$s_i = (a(s_{i-1}))\ \mathrm{mod}(m) = (a^i s_0)\ \mathrm{mod}(m)$
$x_i = s_i/m \qquad \text{for } i = 1, 2, ..., n$

where:

s_i is a random sequence.
x_i is a random number.
s_0 is the initial seed provided by the caller.
$a = 7^5 = 16807.0$
$m = 2^{31}-1 = 2147483647.0$
n is the number of random numbers to be generated.

See references [70] and [74]. If n is 0, no computation is performed, and the initial seed is unchanged.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. $n < 0$
2. *seed* < 1.0 or *seed* ≥ 2147483647.0

Example 1: This example shows a call to SURAND to generate 10 random numbers.

Call Statement and Input

```
             SEED    N   X
               |     |   |
CALL SURAND( SEED , 10 , X )

SEED     =  80629.0
```

Note: It is important to note that SEED is a long-precision number, even though X contains short-precision numbers.

Output

```
SEED     =  759150100.0

X        =  (0.6310323,
             0.7603202,
             0.7015232,
             0.5014868,
             0.4895853,
             0.4602344,
             0.1603608,
             0.1832564,
             0.9899062,
             0.3535068)
```

Example 2: This example shows a call to DURAND to generate 10 random numbers.

Call Statement and Input

```
             SEED    N   X
               |     |   |
CALL DURAND( SEED , 10 , X )

SEED     =  80629.0
```

Output

```
SEED     =  759150100.0

X        =  (0.6310323270182275,
             0.7603201953509451,
             0.7015232633340746,
             0.5014868557925740,
             0.4895853057920864,
             0.4602344475967038,
             0.1603607578018497,
             0.1832563756887132,
             0.9899062002030695,
             0.3535068129904134)
```

SNRAND and DNRAND—Generate a Vector of Normally Distributed Random Numbers

These subroutines generate vector ***x*** of normally distributed pseudo-random numbers, with a mean of 0 and a standard deviation of 1, using Polar methods with a user-specified seed.

Table 164. Data Types

x, *aux*	*seed*	**Subroutine**
Short-precision real	Long-precision real	SNRAND
Long-precision real	Long-precision real	DNRAND

Syntax

Fortran	CALL SNRAND | DNRAND (*seed*, *n*, *x*, *aux*, *naux*)
C and C++	snrand | dnrand (*seed*, *n*, *x*, *aux*, *naux*);
PL/I	CALL SNRAND | DNRAND (*seed*, *n*, *x*, *aux*, *naux*);

On Entry

seed
is the initial value used to generate the random numbers. Specified as: a number of the data type indicated in Table 164. It must be a whole number; that is, the fraction part must be 0. Its value must be $1.0 \le seed < (2147483647.0 = 2^{31}-1)$.

Note: *seed* is always a long-precision real number, even in SNRAND.

n
is the number of random numbers to be generated. Specified as: a fullword integer; *n* must be an even number and $n \ge 0$.

x
See On Return.

aux
has the following meaning:

If $naux = 0$ and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is the storage work area used by this subroutine. Its size must be greater than or equal to *n*/2.

Specified as: an area of storage, containing numbers of the data type indicated in Table 164. They can have any value.

naux
is the size of the work area specified by *aux*. Specified as: a fullword integer, where:

If $naux = 0$ and error 2015 is unrecoverable, SNRAND and DNRAND dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, $naux \ge n/2$.

On Return

seed
is the new seed that is to be used to generate additional random numbers in subsequent invocations of SNRAND or DNRAND. Returned as: a number of the data type indicated in Table 164 on page 957. It is a whole number whose value is $1.0 \le seed < (2147483647.0 = 2^{31}-1)$.

x
is a vector of length *n*, containing the normally distributed pseudo-random numbers. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 164 on page 957.

Notes

1. In your C program, argument *seed* must be passed by reference.
2. Vector ***x*** must have no common elements with the storage area specified for *aux*; otherwise, results are unpredictable.
3. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: The normally distributed pseudo-random numbers, with a mean of 0 and a standard deviation of 1, are generated as follows, using Polar methods with a user-specified seed. The Polar method, which this technique is based on, was developed by G. E. P. Box, M. E. Muller, and G. Marsaglia and is described in reference [70].

1. Using *seed*, a vector of uniform (0,1) pseudo-random numbers, u_i for $i = 1, n$, is generated by calling SURAND or DURAND, respectively. These u_i values are then used in the subsequent steps.
2. All (y_j, z_j) for $j = 1, n/2$ are set as follows, where each (y, z) is a point in the square -1 to 1:

   $$y_j = 2u_{2j-1}-1$$
   $$z_j = 2u_{2j}-1$$

3. All p_j for $j = 1, n/2$ are set as follows, where each p measures the square of the radius of (y, z):

   $$p_j = y_j^2 + z_j^2$$

   If $p_j \ge 1$, then p_j is discarded, and steps 1 through 3 are repeated until $p_j < 1$.
4. All x_i for $i = 1, n$ are set as follows to produce the normally distributed random numbers:

   $$x_{2j-1} = y_j \left((-2 \ln p_j) / p_j\right)^{0.5}$$
   $$x_{2j} = z_j \left((-2 \ln p_j) / p_j\right)^{0.5}$$
   $$\text{for } j = 1, n/2$$

If n is 0, no computation is performed, and the initial seed is unchanged.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors: None

Input-Argument Errors

1. $n < 0$ or n is an odd number
2. *seed* < 1.0 or *seed* ≥ 2147483647.0
3. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example 1: This example shows a call to SNRAND to generate 10 random numbers.

Call Statement and Input

```
             SEED   N    X   AUX  NAUX
               |    |    |    |    |
CALL SNRAND( SEED , 10 , X , AUX , 5  )

SEED     =  80629.0
```

Note: It is important to note that SEED is a long-precision number, even though X contains short-precision numbers.

Output

```
SEED     =  48669425.0

X        =  (0.660649538,
             1.312503695,
             1.906438112,
             0.014065863,
            -0.800935328,
            -3.058144093,
            -0.397426069,
            -0.370634943,
            -0.064151444,
            -0.275887042)
```

Example 2: This example shows a call to DNRAND to generate 10 random numbers.

Call Statement and Input

```
             SEED   N    X   AUX  NAUX
               |    |    |    |    |
CALL DNRAND( SEED , 10 , X , AUX , 5  )

SEED     =  80629.0
```

Output

```
SEED     =  48669425.0

X        =  (0.6606495655963802,
             1.3125037758861060,
             1.9064381379483730,
             0.0140658628770495,
            -0.8009353314494653,
            -3.0581441239248530,
            -0.3974260845722100,
            -0.3706349643478605,
            -0.0641514443372939,
            -0.2758870630332470)
```

SURXOR and DURXOR—Generate a Vector of Long Period Uniformly Distributed Random Numbers

These subroutines generate a vector ***x*** of uniform [0,1) pseudo-random numbers, using the Tausworthe exclusive-or algorithm.

Table 165. Data Types

x, ***vseed***	***iseed***	**Subroutine**
Short-precision real	Integer	SURXOR
Long-precision real	Integer	DURXOR

Syntax

Fortran	CALL SURXOR | DURXOR (*iseed*, *n*, *x*, *vseed*)
C and C++	surxor | durxor (*iseed*, *n*, *x*, *vseed*);
PL/I	CALL SURXOR | DURXOR (*iseed*, *n*, *x*, *vseed*);

On Entry

iseed
has the following meaning, where:

If *iseed* ≠ 0, *iseed* is the initial value used to generate the random numbers. You specify *iseed* ≠ 0 when you call this subroutine for the first time or when you changed *vseed* between calls to this subroutine.

If *iseed* = 0, *vseed* is used to generate the random numbers, where *vseed* was initialized by an earlier call to this subroutine. ESSL assumes you have not changed *vseed* between calls to this subroutine, when you specify *iseed* = 0.

Specified as: a fullword integer, as indicated in Table 165.

n
is the number of random numbers to be generated. Specified as: a fullword integer; *n* ≥ 0.

x
See On Return.

vseed
is the work area used by this subroutine and has the following meaning, where:

If *iseed* ≠ 0, *vseed* is not used for input. The work area can contain anything.

If *iseed* = 0, *vseed* contains the seed vector generated by a preceding call to this subroutine. *vseed* is used in this computation to generate the new random numbers. It should not be changed between calls to this subroutine.

Specified as: a one-dimensional array of (at least) length 10000, containing numbers of the data type indicated in Table 165.

On Return

iseed
is set to 0 for subsequent calls to SURXOR or DURXOR. Returned as: a fullword integer, as indicated in Table 165.

x
is a vector of length *n*, containing the uniform pseudo-random numbers with the following values: $0 \leq x < 1$. Returned as: a one-dimensional array of (at least) length *n*, containing numbers of the data type indicated in Table 165.

vseed
is the work area used by these subroutines, containing the new seed that is to be used in subsequent calls to this subroutine. Returned as: a one-dimensional array of (at least) length 10000, containing numbers of the data type indicated in Table 165 on page 961.

Notes

1. You can generate the same vector *x* of random numbers by starting over and specifying your original nonzero *iseed* value.
2. Multiple calls to these subroutines with mixed sizes generate the same sequence of numbers as a single call the total length, assuming you specify the same initial *iseed* in both cases. For example, you can generate the same vector *x* of random numbers by calling this subroutine twice and specifying $n = 10$ or by calling this subroutine once and specifying $n = 20$. You need to specify the same *iseed* in the initial call in both cases, and $iseed = 0$ in the second call with $n = 10$.
3. Vector ***x*** must have no common elements with the storage area specified for *vseed*; otherwise, results are unpredictable.
4. In your C program, argument *iseed* must be passed by reference.

Function: The pseudo-random numbers uniformly distributed in the interval [0,1) are generated using the Tausworthe exclusive-or algorithm. This is based on a linear-feedback shift-register sequence. The very long period of the generator, $2^{1279}-1$, makes it useful in modern statistical simulations where the shorter period of other generators could be exhausted during a single run. If you need a large number of random numbers, you can use these subroutines, because with this generator you do not request more than a small percentage of the entire period of the generator.

This generator is based on two feedback positions to generate a new binary digit:

$$z_k = z_{(k-p)} \oplus z_{(k-q)}$$

where:

$p > q$
$k = 1, 2, \ldots$
z is a bit vector.
and where:

$\oplus$ is the bitwise exclusive-or operation.

For details, see references [50], [68], and [88]. The values of *p* and *q* are selected according to the criteria stated in reference [94].

The algorithm initializes a seed vector of length *p*, starting with *iseed*. The seed vector is stored in *vseed* for use in subsequent calls to this subroutine with $iseed = 0$.

If *n* is 0, no computation is performed, and the initial seed is unchanged.

Special Usage: For some specialized applications, if you need multiple sources of random numbers, you can specify different *vseed* areas, which are initialized with different seeds on multiple calls to this subroutine. You then get multiple sequences of the random number sequence provided by the generator that are sufficiently far apart for most purposes.

Error Conditions

Computational Errors: None.

Input-Argument Errors

1. *n* < 0
2. *iseed* = 0 and *vseed* does not contain valid data.

Example 1: This example shows a call to SURXOR to generate 10 random numbers.

Call Statement and Input

```
             ISEED    N   X   VSEED
               |      |   |     |
CALL SURXOR( ISEED , 10 , X , VSEED )

ISEED    =  137
```

Output

```
ISEED    =  0

X        =  (0.6440868,
             0.5105118,
             0.4878680,
             0.3209075,
             0.6624528,
             0.2499877,
             0.0056630,
             0.7329214,
             0.7486335,
             0.8050517)
```

Example 2: This example shows a call to SURXOR to generate 10 random numbers. This example specifies *iseed* = 0 and uses the *vseed* output generated from Example 1.

Call Statement and Input

```
             ISEED    N   X   VSEED
               |      |   |     |
CALL SURXOR( ISEED , 10 , X , VSEED )

ISEED    =  0
```

Output

```
ISEED    =  0

X        =  (0.9930249,
             0.0441873,
             0.6891295,
             0.3101060,
             0.6324178,
             0.3299408,
             0.3553145,
             0.0100013,
             0.0214620,
             0.8059390)
```

Example 3: This example shows a call to DURXOR to generate 20 random numbers. This sequence of numbers generated are like those generated in Examples 1 and 2.

Call Statement and Input

```
              ISEED    N   X   VSEED
                |      |   |     |
CALL DURXOR( ISEED , 20 , X , VSEED )

ISEED    =  137
```

Output

```
ISEED    =  0

X        =   (0.64408693438956721,
              0.51051182536460882,
              0.48786801310787142,
              0.32090755617007050,
              0.66245283144861666,
              0.24998782843358081,
              0.00566308101257373,
              0.73292147005172925,
              0.74863359794102236,
              0.80505169697755319,
              0.99302499462139138,
              0.04418740640269125,
              0.68912952155409579,
              0.31010611495627916,
              0.63241786342211936,
              0.32994081459690583,
              0.35531452631408911,
              0.01000134413132581,
              0.02146199494672940,
              0.80593898487597615)
```

Chapter 17. Utilities

The utility subroutines are described in this chapter.

Overview of the Utility Subroutines

The utility subroutines perform general service functions that support ESSL, rather than mathematical computations (Table 166).

Table 166. List of Utility Subroutines

Descriptive Name	Subroutine	Page
ESSL Error Information-Handler Subroutine	EINFO	968
ESSL ERRSAV Subroutine for ESSL	ERRSAV	971
ESSL ERRSET Subroutine for ESSL	ERRSET	972
ESSL ERRSTR Subroutine for ESSL	ERRSTR	974
Set the Vector Section Size (VSS) for the ESSL/370 Scalar Library	IVSSET‡	
Set the Extended Vector Operations Indicator for the ESSL/370 Scalar Library	IEVOPS‡	
Determine the Level of ESSL Installed	IESSL	975
Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines	STRIDE	977
Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode	DSRSM	986
For a General Sparse Matrix, Convert Between Diagonal-Out and Profile-In Skyline Storage Mode	DGKTRN	990
For a Symmetric Sparse Matrix, Convert Between Diagonal-Out and Profile-In Skyline Storage Mode	DSKTRN	996

‡ This subroutine is provided for migration from earlier releases of ESSL and is not intended for use in new programs. Documentation for this subroutines is no longer provided.

Use Considerations

This section describes what you use the utility subroutines for.

Determining the Level of ESSL Installed

IESSL gets the level of ESSL and returns it to your program. The level consists of the following: version number, release number, modification number, and number of the most recently installed ESSL PTF. You can use this function to verify that you are running on or using the capabilities of the desired level.

© Copyright IBM Corp. 1997

Finding the Optimal Stride(s) for Your Fourier Transforms

STRIDE is used to determine optimal stride values for your Fourier transforms when using any of the Fourier transform subroutines, except _RCFT and _CRFT. You must invoke STRIDE for each optimal stride you want computed. Sometimes you need a separate stride for your input and output data. For the three-dimensional Fourier transforms, you need an optimal stride for both the second and third dimensions of the array. The examples provided for STRIDE explain how it is used for each of the subroutines listed above.

After obtaining the optimal strides from STRIDE, you should arrange your data using these stride values. After the data is set up, call the Fourier transform subroutine. For additional information on how to set up your data, see "Setting Up Your Data" on page 744.

Converting Sparse Matrix Storage

DSRSM is used to migrate your existing program from sparse matrices stored by rows to sparse matrices stored in compressed-matrix storage mode. This converts the matrices into a storage format that is compatible with the input requirements for some ESSL sparse matrix subroutines, such as DSMMX.

DGKTRN and DSKTRN are used to convert your sparse matrix from one skyline storage mode to another, if necessary, before calling the subroutines DGKFS/DGKFSP or DSKFS/DSKFSP, respectively.

Utility Subroutines

This section contains the utility subroutine descriptions.

EINFO—ESSL Error Information-Handler Subroutine

This subroutine returns information to your program about the data involved in a computational error that occurred in an ESSL subroutine. This is the same information that is provided in the ESSL messages; however, it allows you to check the information in your program at run time and continue processing. You pass the computational error code of interest to this subroutine in *icode*, and it passes back one or more pieces of information in the output arguments *inf1* and, optionally, *inf2*, as defined in Table 167. **You should use this subroutine only for those computational errors listed in the table. It does not apply to computational errors that do not return information.**

For multithreaded application programs, if you want the error handling capabilities that this subroutine provides to be implemented on each thread created by your program, this subroutine must be called from each thread. If your application creates multiple threads, the action performed by a call to this subroutine applies to the thread that this subroutine was invoked from. For an example, see "Example of Handling Errors in a Multithreaded Application Program" on page 127.

Table 167 (Page 1 of 2). Computational Error Information Returned by EINFO

Error Code	Receiver	Type of Information
2100	*inf1*	Lower range of a vector
	inf2	Upper range of a vector
2101	*inf1*	Index of the eigenvalue that failed to converge
	inf2	Number of iterations after which it failed to converge
2102	*inf1*	Index of the last eigenvector that failed to converge
	inf2	Number of iterations after which it failed to converge
2103	*inf1*	Index of the pivot with zero value
2104	*inf1*	Index of the last pivot with nonpositive value
2105	*inf1*	Index of the pivot element near zero causing factorization to fail
2107	*inf1*	Index of the singular value that failed to converge
	inf2	Number of iterations after which it failed to converge
2109	*inf1*	Iteration count when it was determined that the matrix was not definite
2114	*inf1*	Index of the last eigenvalue that failed to converge
	inf2	Number of iterations after which it failed to converge
2115	*inf1*	Order of the leading minor that was discovered to have a nonpositive determinant
2117	*inf1*	Column number for which pivot value was near zero
2118	*inf1*	Row number for which pivot value was near zero
2120	*inf1*	Row number of empty row where factorization failed

Table 167 (Page 2 of 2). Computational Error Information Returned by EINFO

Error Code	Receiver	Type of Information
2121	*inf1*	Column number of empty column where factorization failed
2126	*inf1*	Row number for which pivot value was unacceptable
2145	*inf1*	First diagonal element with zero value

Syntax

Fortran	CALL EINFO (*icode*[, *inf1*[, *inf2*]])
C and C++	einfo (*icode*, *inf1*, *inf2*);
PL/I	CALL EINFO (*icode*[, *inf1*[, *inf2*]]);

On Entry

icode
has the following meaning, where:

If *icode* = 0, this indicates that the ESSL error option table is to be initialized. (You specify this value once in the beginning of your program before calls to ERRSET.)

If *icode* has any of the allowable error code values listed in Table 167 on page 968, this is the computational error code of interest. (You specify one of these values whenever you want information returned about a computational error.)

Specified as: a fullword integer; *icode* = 0 or an error code value indicated in Table 167 on page 968.

inf1
See On Return.

inf2
See On Return.

On Return

inf1
has the following meaning, where:

If *icode* = 0, this argument is not used in the computation. In this case, *inf1* is an optional argument, except in C and C++ programs.

If *icode* ≠ 0, then *inf1* is the first information receiver, containing numerical information related to the computational error.

Returned as: a fullword integer.

inf2
has the following meaning, where:

If *icode* = 0, this argument is not used in the computation.

If *icode* ≠ 0, then *inf2* is the second information receiver, containing numerical information related to the computational error. It should be specified when the error code provides a second piece of information, and you want the information.

In both of these cases, *inf2* is an optional argument, except in C and C++ programs. For more details, see "Notes" on page 970.

Returned as: a fullword integer.

Notes

1. If *icode* is not 0 and is not one of the error codes specified in Table 167 on page 968, this subroutine returns to the caller, and no information is provided in *inf1* and *inf2*.
2. If there are two pieces of information for the error and you specify one output argument, the second piece of information is not returned to the caller.
3. If there is one piece of information for the error and you specify two output arguments, the second output argument is not set by this subroutine.
4. In C and C++ programs you must code the *inf1* and *inf2* arguments, because they are not optional arguments.
5. In Fortran programs, *inf1* and *inf2* are optional arguments. This is an exception to the rule, because other ESSL subroutines do not allow optional arguments.
6. Examples of how to use EINFO are provided in Chapter 4 on page 111.

ERRSAV—ESSL ERRSAV Subroutine for ESSL

The ERRSAV subroutine copies an ESSL error option table entry into an 8-byte storage area that is accessible to your program.

For multithreaded application programs, if you want the error handling capabilities that this subroutine provides to be implemented on each thread created by your program, this subroutine must be called from each thread. If your application creates multiple threads, the action performed by a call to this subroutine applies to the thread that this subroutine was invoked from. For an example, see "Example of Handling Errors in a Multithreaded Application Program" on page 127.

Syntax

Fortran	CALL ERRSAV (*ierno*, *tabent*)
C and C++	errsav (*ierno*, *tabent*);
PL/I	CALL ERRSAV (*ierno*, *tabent*);

On Entry

ierno
is the error number in the option table. The entry for *ierno* in the ESSL error option table is stored in the 8-byte storage area *tabent*. Specified as: a fullword integer; *ierno* must be one of the error numbers in the option table. For a list of these numbers, see Table 26 on page 51.

tabent
is the storage area where the option table entry is stored. Specified as: an area of storage of length 8-bytes.

Note: Examples of how to use ERRSAV are provided in Chapter 4 on page 111.

ERRSET—ESSL ERRSET Subroutine for ESSL

The ERRSET subroutine allows you to control execution when error conditions occur. It modifies the information in the ESSL error option table for the error number indicated. For a range of error messages, you can specify the following:

- How many times a particular error is allowed to occur before the program is terminated
- How many times a particular error message is printed before printing is suppressed
- Whether the ESSL error exit routine is to be invoked

For multithreaded application programs, if you want the error handling capabilities that this subroutine provides to be implemented on each thread created by your program, this subroutine must be called from each thread. If your application creates multiple threads, the action performed by a call to this subroutine applies to the thread that this subroutine was invoked from. For an example, see "Example of Handling Errors in a Multithreaded Application Program" on page 127.

Syntax

Fortran	CALL ERRSET (*ierno*, *inoal*, *inomes*, *itrace*, *iusadr*, *irange*)
C and C++	errset (*ierno*, *inoal*, *inomes*, *itrace*, *iusadr*, *irange*);
PL/I	CALL ERRSET (*ierno*, *inoal*, *inomes*, *itrace*, *iusadr*, *irange*);

On Entry

ierno
is the error number in the option table. The entry for *ierno* in the ESSL error option table is updated as indicated by the other arguments. Specified as: a fullword integer; *ierno* must be one of the error numbers in the option table. For a list of these numbers, see Table 26 on page 51.

inoal
indicates the number of errors allowed before each execution is terminated, where:

If *inoal* ≤ 0, the specification is ignored, and the number-of-errors option is not changed.

If *inoal* = 1, execution is terminated after one error.

If 2 ≤ *inoal* ≤ 255, then *inoal* specifies the number of errors allowed before each execution is terminated.

If *inoal* > 255, an unlimited number of errors is allowed.

Specified as: a fullword integer, where:

If *iusadr* = ENOTRM, then 2 ≤ *inoal* ≤ 255.

inomes
indicates the number of messages to be printed, where:

If *inomes* < 0, all messages are suppressed.

If *inomes* = 0, the number-of-messages option is not changed.

If 0 < *inomes* ≤ 255, then *inomes* specifies the number of messages to be printed.

If *inomes* > 255, an unlimited number of error messages is allowed.

Specified as: a fullword integer.

itrace

this argument is ignored for ESSL/6000, but must be specified. Specified as: a fullword integer where, *itrace* = 0, 1, or 2 (for migration purposes).

iusadr

indicates whether or not the ESSL error exit routine is to be invoked, where:

If *iusadr* = 0, the option table is not altered.

If *iusadr* = 1, the option table is set to show no exit routine. Therefore, standard corrective action is to be used when continuing execution.

If *iusadr* = ENOTRM, the option table entry is set to the ESSL error exit routine ENOTRM. Therefore, the ENOTRM subroutine is to be invoked after the occurrence of the indicated errors. (ENOTRM must appear in an EXTERNAL statement in your program.)

Specified as: a fullword integer or the name of a subroutine; *iusadr* = 0, 1, or ENOTRM.

irange

indicates the range of errors to be updated in the ESSL error option table, where:

If *irange* < *ierno*, the parameter is ignored.

If *irange* ≥ *ierno*, the options specified for the other parameters are to be applied to the entire range of error conditions encompassed by *ierno* and *irange*.

Specified as: a fullword integer.

Notes

1. Examples of how to use ERRSET are provided in Chapter 4 on page 111.
2. If you specify ENOTRM for *iusadr*, then *inoal* must be in the following range: 2 ≤ *inoal* ≤ 255.

ERRSTR—ESSL ERRSTR Subroutine for ESSL

The ERRSTR subroutine stores an entry in the ESSL error option table.

For multithreaded application programs, if you want the error handling capabilities that this subroutine provides to be implemented on each thread created by your program, this subroutine must be called from each thread. If your application creates multiple threads, the action performed by a call to this subroutine applies to the thread that this subroutine was invoked from. For an example, see "Example of Handling Errors in a Multithreaded Application Program" on page 127.

Syntax

Fortran	CALL ERRSTR (*ierno*, *tabent*)
C and C++	errstr (*ierno*, *tabent*);
PL/I	CALL ERRSTR (*ierno*, *tabent*);

On Entry

ierno
is the error number in the option table. The information in the 8-byte storage area *tabent* is stored into the entry for *ierno* in the ESSL error option table. Specified as: a fullword integer; *ierno* must be one of the error numbers in the option table. For a list of these numbers, see Table 26 on page 51.

tabent
is the storage area containing the table entry data. Specified as: an area of storage of length 8-bytes.

Note: Examples of how to use ERRSTR are provided in Chapter 4 on page 111.

IESSL—Determine the Level of ESSL Installed

This function returns the level of ESSL installed on your system, where the level consists of a version number, release number, and modification number, plus the fix number of the most recent PTF installed.

Syntax

Fortran	CALL IESSL ()
C and C++	iessl ();
PL/I	CALL IESSL ();

On Return

Function value

is the level of ESSL installed on your system. It is provided as a fullword integer in the form *vvrrmmff*, where each two digits represents a part of the level:

- *vv* is the version number.
- *rr* is the release number.
- *mm* is the modification number.
- *ff* is the fix number of the most recent PTF installed.

Returned as: a fullword integer; *vvrrmmff* > 0.

Notes

1. To use IESSL effectively, you must install your ESSL PTFs in their proper sequential order. As part of the result, IESSL returns the value *ff* of the **most recent** PTF installed, rather than the **highest number** PTF installed. Therefore, if you do not install your PTFs sequentially, the *ff* value returned by IESSL does not reflect the actual level of ESSL.
2. Declare the IESSL function in your program as returning a fullword integer value.

Function: The IESSL function enables you to determine the current level of ESSL installed on your system. It is useful to you in those instances where your program is using a subroutine or feature that exists only in certain levels of ESSL. It is also useful when your program is dependent upon certain PTFs being applied to ESSL.

Example: This example shows several ways to use the IESSL function. Most typically, you use IESSL for checking the version and release level of ESSL. Suppose you are dependent on a new capability in ESSL, such as a new subroutine or feature, provided for the first time in ESSL Version 3. You can add the following check in your program before using the new capability:

```
IF IESSL() ≥ 3010000
```

By specifying 0000 for *mmff*, the modification and fix level, you are independent of the order in which your modifications and PTFs are installed.

Less typically, you use IESSL for checking the PTF level of ESSL. Suppose you are dependent on PTF 2 being installed on your ESSL Version 3 system. You want to know whether to call a different user-callable subroutine to set up your array data. You can add the following check in your program before making the call:

```
IF IESSL() ≥ 3010002
```

If your system support group installed the ESSL PTFs in their proper sequential order, this test works properly; otherwise, it is unpredictable.

STRIDE—Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines

This subroutine determines an optimal stride value for you to use for your input or output data when you are computing large row Fourier transforms in any of the Fourier transform subroutines, except _RCFT and _CRFT. The strides determined by this subroutine allow your arrays to fit comfortably in various levels of storage hierarchy on your particular processor, thus allowing you to improve your run-time performance.

Note: This subroutine returns a single stride value. Where you need multiple strides, you must invoke this subroutine multiple times; for example, in the multidimensional Fourier transforms and, also, when input and output data types differ. For more details, see "Function" on page 978.

Syntax

Fortran	CALL STRIDE (*n*, *incd*, *incr*, *dt*, *iopt*)
C and C++	stride (*n*, *incd*, *incr*, *dt*, *iopt*);
PL/I	CALL STRIDE (*n*, *incd*, *incr*, *dt*, *iopt*);

On Entry

n
is the length *n* of the Fourier transform for which the optimal stride is being determined. The transform corresponding to *n* is usually a row transform; that is, the data elements are stored using a stride value.

Specified as: a fullword integer; *n* > 0.

incd
is the minimum allowable stride for the Fourier transform for which the optimal stride is being determined. For each situation in each subroutine, there is a specific way to compute this minimum value. This is explained in the examples starting on page 979.

Specified as: a fullword integer; *incd* > 0 or *incd* < 0.

incr
See On Return.

dt
is the data type of the numbers for the Fourier transform for which the optimal stride is being determined, where:

If *dt* = 'S', the numbers are short-precision real.

If *dt* = 'D', the numbers are long-precision real.

If *dt* = 'C', the numbers are short-precision complex.

If *dt* = 'Z', the numbers are long-precision complex.

Specified as: a single character; *dt* = 'S', 'D', 'C', or 'Z'.

iopt
is provided only for migration purposes from ESSL Version 1 and is no longer used; however, you must still specify it as a dummy argument. Specified as: a fullword integer; *iopt* = 0, 1, or 2.

On Return

incr
is the stride that allows you to improve your run-time performance in your Fourier transform computation on your particular processor. In general, this value differs for each processor you are running on.

Returned as: a fullword integer; $incr > 0$ or $incr < 0$ and $|incr| \geq |incd|$, where *incr* has the same sign (+ or −) as *incd*.

Notes

1. In your C program, argument *incr* must be passed by reference.
2. All subroutines accept lowercase letters for the *dt* argument.
3. For each situation in each of the Fourier transform subroutines, there is a specific way to compute the value you should specify for the *incd* argument. Details on how to compute each of these values is given in the examples starting on page 979. See the example corresponding to the Fourier transform subroutine you are using.
4. Where different data types are specified for the input and output data in your Fourier transform subroutine, you should be careful to indicate the correct data type in the *dt* argument in this subroutine.
5. For additional information on how to set up your data, see "Setting Up Your Data" on page 744.

Function: This subroutine determines an optimal stride, *incr*, for you to use for your input or output data when computing large row Fourier transforms. The stride value returned by this subroutine is based on the size and structure of your transform data, using:

- The size of each data item (*dt*)
- The minimum allowable stride for this transform (*incd*)
- The length of the transform (*n*)

This information is used in determining the optimal stride for the processor you are currently running on. The stride determined by this subroutine allows your arrays to fit comfortably in various levels of storage hierarchy for that processor, thus giving you the ability to improve your run-time performance.

You get only one stride value returned by this subroutine on each invocation. Therefore, in many instances, you may need to invoke this subroutine multiple times to obtain several stride values to use in your Fourier transform computation:

- For multidimensional Fourier transforms using several strides, this subroutine must be called once for each optimal stride you want to obtain. Successive invocations should go from the lower (earlier) dimensions to the higher (later) dimensions, because the results from the lower dimensions are used to calculate the *incd* values for the higher dimensions.
- Where input and output data have different data types and you want to obtain optimal strides for each, this subroutine must be called once for each data type.

Where multiple invocations are necessary, they are explained in the examples starting on page 979. The examples also explain how to calculate the *incd* values for each invocation. There are nine examples to cover the Fourier transform subroutines that can use the STRIDE subroutine.

After calling this subroutine and obtaining the optimal stride value, you then set up your input or output array accordingly. This may involve movement of data for input arrays or increasing the sizes of input or output arrays. To accomplish this, you may want to set up a separate subroutine with the stride values passed into it as arguments. You can then dimension your arrays in that subroutine, depending on the values calculated by STRIDE. For additional information on how to set up your data, see "Setting Up Your Data" on page 744.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. $n \leq 0$
2. $incd = 0$
3. $iopt \neq 0$, 1, or 2
4. $dt \neq$ S, D, C, or Z

Example 1—SCFT: This example shows the use of the STRIDE subroutine in computing one-dimensional row transforms using the SCFT subroutine.

If $inc2x = 1$, the input sequences are stored in the transposed form as rows of a two-dimensional array `X(INC1X,N)`. In this case, the STRIDE subroutine helps in determining a good value of *inc1x* for this array. The required minimum value of *inc1x* is *m*, the number of Fourier transforms being computed. To find a good value of *inc1x*, use STRIDE as follows:

```
             N  INCD  INCR    DT   IOPT
             |   |     |       |    |
CALL STRIDE( N , M  , INC1X , 'C' , 0  )
```

Here, the arguments refer to the SCFT subroutine. In the following table, values of *inc1x* are given (as obtained from the STRIDE subroutine) for some combinations of *n* and *m* and for POWER2-P2SC with 64KB level 1 cache using the POWER2 library for ESSL:

```
     N      M    INC1X

     128    64     64
     240    32     32
     240    64     65
     256   256    264
     512    60     60
    1024    64     65
```

The above example also applies when the output sequences are stored in the transposed form ($inc2y = 1$). In that case, in the above example, *inc1x* is replaced by *inc1y*.

In computing column transforms ($inc1x = inc1y = 1$), the values of *inc2x* and *inc2y* are not very important. For these, any value over the required minimum of *n* can be used.

Example 2—DCOSF: This example shows the use of the STRIDE subroutine in computing one-dimensional row transforms using the DCOSF subroutine.

If *inc2x* = 1, the input sequences are stored in the transposed form as rows of a two-dimensional array X(INC1X,N/2+1). In this case, the STRIDE subroutine helps in determining a good value of *inc1x* for this array. The required minimum value of *inc1x* is *m*, the number of Fourier transforms being computed. To find a good value of *inc1x*, use STRIDE as follows:

```
              N     INCD  INCR    DT  IOPT
              |      |     |      |    |
CALL STRIDE( N/2+1 , M  , INC1X , 'D' , 0  )
```

Here, the arguments refer to the DCOSF subroutine. In the following table, values of *inc1x* are given (as obtained from the STRIDE subroutine) for some combinations of *n* and *m* and for POWER2-P2SC with 64KB level 1 cache using the POWER2 library for ESSL:

```
     N      M    INC1X

     128    64     64
     240    32     32
     240    64     64
     256   256    264
     512    60     60
    1024    64     65
```

The above example also applies when the output sequences are stored in the transposed form (*inc2y* = 1). In that case, in the above example, *inc1x* is replaced by *inc1y*.

In computing column transforms (*inc1x* = *inc1y* = 1), the values of *inc2x* and *inc2y* are not very important. For these, any value over the required minimum of *n*/2+1 can be used.

Example 3—DSINF: This example shows the use of the STRIDE subroutine in computing one-dimensional row transforms using the DSINF subroutine.

If *inc2x* = 1, the input sequences are stored in the transposed form as rows of a two-dimensional array X(INC1X,N/2). In this case, the STRIDE subroutine helps in determining a good value of *inc1x* for this array. The required minimum value of *inc1x* is *m*, the number of Fourier transforms being computed. To find a good value of *inc1x*, use STRIDE as follows:

```
              N   INCD  INCR    DT   IOPT
              |    |     |       |    |
CALL STRIDE( N/2 , M  , INC1X , 'D' , 0  )
```

Here, the arguments refer to the DSINF subroutine. In the following table, values of *inc1x* are given (as obtained from the STRIDE subroutine) for some combinations of *n* and *m* and for POWER2-P2SC with 64KB level 1 cache using the POWER2 library for ESSL:

```
     N      M    INC1X

     128    64     64
     240    32     32
     240    64     64
     256   256    264
     512    60     60
    1024    64     65
```

The above example also applies when the output sequences are stored in the transposed form (*inc2y* = 1). In that case, in the above example, *inc1x* is replaced by *inc1y*.

In computing column transforms (*inc1x* = *inc1y* = 1), the values of *inc2x* and *inc2y* are not very important. For these, any value over the required minimum of *n*/2 can be used.

Example 4—SCFT2: This example shows the use of the STRIDE subroutine in computing two-dimensional transforms using the SCFT2 subroutine.

If *inc1y* = 1, the two-dimensional output array is stored in the normal form. In this case, the output array can be declared as `Y(INC2Y,N2)`, where the required minimum value of *inc2y* is *n1*. The STRIDE subroutine helps in picking a good value of *inc2y*. To find a good value of *inc2y*, use STRIDE as follows:

```
             N   INCD  INCR    DT   IOPT
             |    |     |       |    |
CALL STRIDE( N2 , N1 , INC2Y , 'C' , 0  )
```

Here, the arguments refer to the SCFT2 subroutine. In the following table, values of *inc2y* are given (as obtained from the STRIDE subroutine) for some two-dimensional arrays with *n1* = *n2* and for POWER2-P2SC with 64KB level 1 cache using the POWER2 library for ESSL:

```
     N1      N2   INC2Y

      64     64     64
     128    128    136
     240    240    240
     512    512    520
     840    840    848
```

If the input array is stored in the normal form (*inc1x* = 1), the value of *inc2x* is not important. However, if you want to use the same array for input and output, you should use *inc2x* = *inc2y*.

If *inc2y* = 1, the two-dimensional output array is stored in the transposed form. In this case, the output array can be declared as `Y(INC1Y,N1)`, where the required minimum value of *inc1y* is *n2*. The STRIDE subroutine helps in picking a good value of *inc1y*. To find a good value of *inc1y*, use STRIDE as follows:

```
             N   INCD  INCR    DT   IOPT
             |    |     |       |    |
CALL STRIDE( N1 , N2 , INC1Y , 'C' , 0  )
```

Here, the arguments refer to the SCFT2 subroutine. In the following table, values of *inc1y* are given (as obtained from the STRIDE subroutine) for some combinations of *n1* and *n2* and for POWER2-P2SC with 64K level 1 cache using the POWER2 library for ESSL:

```
     N1     N2   INC1Y

     60     64     64
    120    128    136
    256    240    240
    512    512    520
    840    840    848
```

If the input array is stored in the transposed form (*inc2x* = 1), the value of *inc1x* is also important. The above example can be used to find a good value of *inc1x*, by replacing *inc1y* with *inc1x*. If both arrays are stored in the transposed form, a good value for *inc1y* is also a good value for *inc1x*. In that situation, the two arrays can also be made equivalent.

Example 5—SRCFT2: This example shows the use of the STRIDE subroutine in computing two-dimensional transforms using the SRCFT2 subroutine.

For this subroutine, the output array is declared as `Y(INC2Y,N2)`, where the required minimum value of *inc2y* is *n1*/2+1. The STRIDE subroutine helps in picking a good value of *inc2y*. To find a good value of *inc2y*, use STRIDE as follows:

```
             N      INCD     INCR    DT   IOPT
             |        |        |      |    |
CALL STRIDE( N2 , N1/2 + 1 , INC2Y , 'C' , 0  )
```

Here, the arguments refer to the SRCFT2 subroutine. In the following table, values of *inc2y* are given (as obtained from the STRIDE subroutine) for some two-dimensional arrays with *n1* = *n2* and for POWER2-P2SC with 64KB level 1 cache using the POWER2 library for ESSL:

N1	N2	INC2Y
240	240	121
420	420	211
512	512	257
840	840	421
1024	1024	513
2048	2048	1032

For this subroutine, the leading dimension of the input array (*inc2x*) is not important. If you want to use the same array for input and output, you should use *inc2x* ≥ 2(*inc2y*).

Example 6—SCRFT2: This example shows the use of the STRIDE subroutine in computing two-dimensional transforms using the SCRFT2 subroutine.

For this subroutine, the output array is declared as `Y(INC2Y,N2)`, where the required minimum value of *inc2y* is *n1*+2. The STRIDE subroutine helps in picking a good value of *inc2y*. To find a good value of *inc2y*, use STRIDE as follows:

```
             N     INCD    INCR    DT   IOPT
             |       |       |      |    |
CALL STRIDE( N2 , N1 + 2 , INC2Y , 'S' , 0  )
```

Here, the arguments refer to the SCRFT2 subroutine. In the following table, values of *inc2y* are given (as obtained from the STRIDE subroutine) for some two-dimensional arrays with *n1* = *n2* and for POWER2-P2SC with 64KB level 1 cache using the POWER2 library for ESSL:

```
     N1      N2    INC2Y

     240     240    242
     420     420    422
     512     512    514
     840     840    842
     1024   1024   1026
     2048   2048   2064
```

For this subroutine, the leading dimension of the input array (*inc2x*) is also important. In general, *inc2x* = *inc2y*/2 is a good choice. This is also the requirement if you want to use the same array for input and output.

Example 7—SCFT3: This example shows the use of the STRIDE subroutine in computing three-dimensional transforms using the SCFT3 subroutine.

For this subroutine, the strides for the input array are not important. They are important for the output array. The STRIDE subroutine helps in picking good values of *inc2y* and *inc3y*. This requires two calls to the STRIDE subroutine as shown below. First, you should find a good value for *inc2y*. The minimum acceptable value for *inc2y* is *n1*.

```
             N   INCD  INCR    DT   IOPT
             |    |      |      |    |
CALL STRIDE( N2 , N1 , INC2Y , 'C' , 0  )
```

Here, the arguments refer to the SCFT3 subroutine. Next, you should find a good value for *inc3y*. The minimum acceptable value for *inc3y* is (*n2*)(*inc2y*).

```
             N      INCD    INCR    DT   IOPT
             |       |       |      |    |
CALL STRIDE( N3 , N2*INC2Y, INC3Y , 'C' , 0  )
```

If *inc3y* turns out to be a multiple of *inc2y*, then `Y` can be declared a three-dimensional array as `Y(INC2Y,INC3Y/INC2Y,N3)`. For large problems, this may not happen. In that case, you can declare the `Y` array as a two-dimensional array `Y(0:INC3Y-1,0:N3-1)` or a one-dimensional array `Y(0:INC3Y*N3-1)`. Using zero-based indexing, the element *y(k1,k2,k3)* is stored in the following location in these arrays:

- For the two-dimensional array, location *(k1+k2*inc2y,k3)*
- For the one-dimensional array, location *(k1+k2*inc2y+k3*inc3y)*

In the following table, values of *inc2y* and *inc3y* are given (as obtained from the STRIDE subroutine) for some three-dimensional arrays with *n1* = *n2* = *n3* and for POWER2-P2SC with 64KB level 1 cache using the POWER2 library for ESSL:

```
  N1,N2,N3    INC2Y    INC3Y

     30         30       900
     32         32      1032
     64         64      4112
    120        120     14400
    128        136     17416
    240        240     57608
    256        264     67592
    420        420    176400
```

As mentioned before, the strides of the input array are not important. The array can be declared as a three-dimensional array. If you want to use the same array for input and output, the requirements are *inc2x* ≥ *inc2y* and *inc3x* ≥ *inc3y*. A simple thing to do is to use *inc2x* = *inc2y* and make *inc3x* a multiple of *inc2x* not smaller than *inc3y*. Then X can be declared as a three-dimensional array X(INC2X,INC3X/INC2X,N3).

Example 8—SRCFT3: This example shows the use of the STRIDE subroutine in computing three-dimensional transforms using the SRCFT3 subroutine.

For this subroutine, the strides for the input array are not important. They are important for the output array. The STRIDE subroutine helps in picking good values of *inc2y* and *inc3y*. This requires two calls to the STRIDE subroutine as shown below. First, you should find a good value for *inc2y*. The minimum acceptable value for *inc2y* is *n1*/2+1.

```
             N       INCD    INCR    DT   IOPT
             |        |       |      |    |
CALL STRIDE( N2 , N1/2 + 1 , INC2Y , 'C' , 0  )
```

Here, the arguments refer to the SRCFT3 subroutine. Next, you should find a good value for *inc3y*. The minimum acceptable value for *inc3y* is (*n2*)(*inc2y*).

```
             N      INCD     INCR    DT   IOPT
             |      |        |       |    |
CALL STRIDE( N3 , N2*INC2Y , INC3Y , 'C' , 0  )
```

If *inc3y* turns out to be a multiple of *inc2y*, then Y can be declared a three-dimensional array as Y(INC2Y,INC3Y/INC2Y,N3). For large problems, this may not happen. In that case, you can declare the Y array as a two-dimensional array Y(0:INC3Y-1,0:N3-1) or a one-dimensional array Y(0:INC3Y*N3-1). Using zero-based indexing, the element *y(k1,k2,k3)* is stored in the following location in these arrays:

- For the two-dimensional array, location *(k1+k2*inc2y,k3)*
- For the one-dimensional array, location *(k1+k2*inc2y+k3*inc3y)*

In the following table, values of *inc2y* and *inc3y* are given (as obtained from the STRIDE subroutine) for some three-dimensional arrays with *n1* = *n2* = *n3* and for POWER2-P2SC with 64KB level 1 cache using the POWER2 library for ESSL:

N1,N2,N3	INC2Y	INC3Y
32	17	544
64	33	2112
120	61	7320
128	65	8328
240	121	29064
256	129	33032

As mentioned before, the strides of the input array are not important. The array can be declared as a three-dimensional array. If you want to use the same array for input and output, the requirements are *inc2x* ≥ 2(*inc2y*) and *inc3x* ≥ 2(*inc3y*). A simple thing to do is to use *inc2x* = 2(*inc2y*) and make *inc3x* a multiple of *inc2x* not smaller than 2(*inc3y*). Then X can be declared as a three-dimensional array X(INC2X,INC3X/INC2X,N3).

Example 9—SCRFT3: This example shows the use of the STRIDE subroutine in computing three-dimensional transforms using the SCRFT3 subroutine.

The STRIDE subroutine helps in picking good values of *inc2y* and *inc3y*. This requires two calls to the STRIDE subroutine as shown below. First, you should find a good value for *inc2y*. The minimum acceptable value for *inc2y* is *n1*+2.

```
             N      INCD     INCR     DT    IOPT
             |        |        |       |     |
CALL STRIDE( N2 , N1 + 2 , INC2Y , 'S' , 0  )
```

Here, the arguments refer to the SCRFT3 subroutine. Next, you should find a good value for *inc3y*. The minimum acceptable value for *inc3y* is (*n2*)(*inc2y*).

```
             N       INCD      INCR     DT    IOPT
             |         |         |       |     |
CALL STRIDE( N3 , N2*INC2Y , INC3Y , 'S' , 0  )
```

If *inc3y* turns out to be a multiple of *inc2y*, then `Y` can be declared a three-dimensional array as `Y(INC2Y,INC3Y/INC2Y,N3)`. For large problems, this may not happen. In that case, you can declare the `Y` array as a two-dimensional array `Y(0:INC3Y-1,0:N3-1)` or a one-dimensional array `Y(0:INC3Y*N3-1)`. Using zero-based indexing, the element *y(k1,k2,k3)* is stored in the following location in these arrays:

- For the two-dimensional array, location *(k1+k2*inc2y,k3)*
- For the one-dimensional array, location *(k1+k2*inc2y+k3*inc3y)*

In the following table, values of *inc2y* and *inc3y* are given (as obtained from the STRIDE subroutine) for some three-dimensional arrays with *n1 = n2 = n3* and for POWER2-P2SC with 64KB level 1 cache using the POWER2 library for ESSL:

N1,N2,N3	INC2Y	INC3Y
32	34	1088
64	66	4224
120	122	14640
128	130	16656
240	242	58128
256	258	66064

For this subroutine, the strides (*inc2x* and *inc3x*) of the input array are also important. In general, *inc2x = inc2y*/2 and *inc3x = inc3y*/2 are good choices. These are also the requirement if you want to use the same array for input and output.

DSRSM—Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode

This subroutine converts either *m* by *n* general sparse matrix ***A*** or symmetric sparse matrix ***A*** of order *n* from storage-by-rows to compressed-matrix storage mode, where matrix ***A*** contains long-precision real numbers.

Syntax

Fortran	CALL DSRSM (*iopt*, *ar*, *ja*, *ia*, *m*, *nz*, *ac*, *ka*, *lda*)
C and C++	dsrsm (*iopt*, *ar*, *ja*, *ia*, *m*, *nz*, *ac*, *ka*, *lda*);
PL/I	CALL DSRSM (*iopt*, *ar*, *ja*, *ia*, *m*, *nz*, *ac*, *ka*, *lda*);

On Entry

iopt
indicates the storage variation used for sparse matrix ***A*** storage-by-rows:

If *iopt* = 0, matrix ***A*** is a general sparse matrix, where all the nonzero elements in matrix ***A*** are used to set up the storage arrays.

If *iopt* = 1, matrix ***A*** is a symmetric sparse matrix, where only the upper triangle and diagonal elements are used to set up the storage arrays.

Specified as: a fullword integer; *iopt* = 0 or 1.

ar
is the sparse matrix ***A***, stored by rows in an array, referred to as AR. The *iopt* argument indicates the storage variation used for storing matrix ***A***. Specified as: a one-dimensional array, containing long-precision real numbers. The number of elements, *ne*, in this array can be determined by subtracting 1 from the value in IA(*m*+1).

ja
is the array, referred to as JA, containing the column numbers of each nonzero element in sparse matrix ***A***. Specified as: a one-dimensional array, containing fullword integers; 1 ≤ (JA elements) ≤ *n*. The number of elements, *ne*, in this array can be determined by subtracting 1 from the value in IA(*m*+1).

ia
is the row pointer array, referred to as IA, containing the starting positions of each row of matrix ***A*** in array AR and one position past the end of array AR. Specified as: a one-dimensional array of (at least) length *m*+1, containing fullword integers; IA(*i*+1) ≥ IA(*i*) for *i* = 1, *m*+1.

m
is the number of rows in sparse matrix ***A***. Specified as: a fullword integer; *m* ≥ 0.

nz
is the number of columns in output arrays AC and KA that are available for use. Specified as: a fullword integer; *nz* > 0.

ac
See On Return.

ka
See On Return.

lda
is the size of the leading dimension of the arrays specified for *ac* and *ka*. Specified as: a fullword integer; 0 < *lda* ≤ *m*.

On Return

nz
is the maximum number of nonzero elements, *nz*, in each row of matrix ***A***, which is stored in compressed-matrix storage mode. Returned as: a fullword integer; (input argument) *nz* ≤ (output argument) *nz*.

ac
is the *m* by *n* general sparse matrix ***A*** or symmetric matrix ***A*** of order *n* stored in compressed-matrix storage mode in an array, referred to as AC. Returned as: an *lda* by at least (input argument) *nz* array, containing long-precision real numbers, where only the first (output argument) *nz* columns are used to store the matrix.

ka
is the array, referred to as KA, containing the column numbers of the matrix ***A*** elements that are stored in the corresponding positions in array AC. Returned as: an *lda* by at least (input argument) *nz* array, containing fullword integers, where only the first (output argument) *nz* columns are used to store the column numbers.

Notes

1. In your C program, argument *nz* must be passed by reference.
2. The value specified for input argument *nz* should be greater than or equal to the number of nonzero elements you estimate to be in each row of sparse matrix ***A***. The value returned in output argument *nz* corresponds to the *nz* value defined for compressed-matrix storage mode. This value is less than or equal to the value specified for input argument *nz*.
3. For a description of the storage modes for sparse matrices, see "Compressed-Matrix Storage Mode" on page 93 and "Storage-by-Rows" on page 99.

Function: A sparse matrix ***A*** is converted from storage-by-rows (using arrays AR, JA, and IA) to compressed-matrix storage mode (using arrays AC and KA). The argument *iopt* indicates whether the input matrix ***A*** is stored by rows using the storage variation for general sparse matrices or for symmetric sparse matrices. See reference [67].

This subroutine is meant for existing programs that need to convert their sparse matrices to a storage mode compatible with some of the ESSL sparse matrix subroutines, such as DSMMX.

Error Conditions

Computational Errors: None

Input-Argument Errors

1. *iopt* ≠ 0 or 1
2. *m* < 0
3. *lda* < 1
4. *lda* < *m*
5. *nz* ≤ 0
6. IA(*m*+1) < 1
7. IA(*i*+1)−IA(*i*) < 0, for any *i* = 1, *m*
8. *nz* is too small to store matrix ***A*** in array AC, where:
   - If *iopt* = 0 , AC and KA are not modified.
   - If *iopt* = 1 , AC and KA are modified.

Example 1: This example shows a general sparse matrix ***A***, which is stored by rows and converted to compressed-matrix storage mode, where sparse matrix ***A*** is:

```
⎡                        ⎤
⎢ 11.0   0.0   0.0  14.0 ⎥
⎢  0.0  22.0   0.0  24.0 ⎥
⎢  0.0   0.0  33.0  34.0 ⎥
⎢  0.0   0.0   0.0  44.0 ⎥
⎣                        ⎦
```

Because there is a maximum of only two nonzero elements in each row of ***A***, and argument *nz* is specified as 5, columns 3 through 5 of arrays AC and KA are not used.

Call Statement and Input

```
           IOPT  AR   JA   IA   M   NZ  AC   KA  LDA
            |    |    |    |    |   |   |    |    |
CALL DSRSM( 0  , AR , JA , IA , 4 , 5 , AC , KA , 4 )

AR       =  (11.0, 14.0, 22.0, 24.0, 33.0, 34.0, 44.0)
JA       =  (1, 4, 2, 4, 3, 4, 4)
IA       =  (1, 3, 5, 7, 8)
```

Output

```
NZ       =  2

         ⎡                     ⎤
         ⎢ 11.0  14.0  .  .  . ⎥
AC   =   ⎢ 22.0  24.0  .  .  . ⎥
         ⎢ 33.0  34.0  .  .  . ⎥
         ⎢ 44.0   0.0  .  .  . ⎥
         ⎣                     ⎦

         ⎡                  ⎤
         ⎢ 1  4  .  .  .  ⎥
KA   =   ⎢ 2  4  .  .  .  ⎥
         ⎢ 3  4  .  .  .  ⎥
         ⎢ 4  4  .  .  .  ⎥
         ⎣                  ⎦
```

Example 2: This example shows a symmetric sparse matrix ***A***, which is stored by rows and converted to compressed-matrix storage mode, where sparse matrix ***A*** is:

```
⎡                        ⎤
⎢ 11.0   0.0   0.0  14.0 ⎥
⎢  0.0  22.0   0.0  24.0 ⎥
⎢  0.0   0.0  33.0  34.0 ⎥
⎢ 14.0  24.0  34.0  44.0 ⎥
⎣                        ⎦
```

Because there is a maximum of only four nonzero elements in each row of ***A***, and argument *nz* is specified as 6, columns 5 and 6 of arrays AC and KA are not used.

Call Statement and Input

```
           IOPT  AR   JA   IA   M   NZ  AC   KA  LDA
            |    |    |    |    |   |   |    |    |
CALL DSRSM( 1  , AR , JA , IA , 4 , 6 , AC , KA , 4 )

AR       =  (11.0, 14.0, 22.0, 24.0, 33.0, 34.0, 44.0)
JA       =  (1, 4, 2, 4, 3, 4, 4)
IA       =  (1, 3, 5, 7, 8)
```

Output

```
NZ       =  4

        ⎡                          ⎤
        ⎢ 11.0  14.0   0.0   0.0  .  . ⎥
AC   =  ⎢ 22.0  24.0   0.0   0.0  .  . ⎥
        ⎢ 33.0  34.0   0.0   0.0  .  . ⎥
        ⎢ 44.0  24.0  34.0  14.0  .  . ⎥
        ⎣                          ⎦

        ⎡                    ⎤
        ⎢ 1  4  4  4  .  . ⎥
KA   =  ⎢ 2  4  4  4  .  . ⎥
        ⎢ 3  4  4  4  .  . ⎥
        ⎢ 4  2  3  1  .  . ⎥
        ⎣                    ⎦
```

DGKTRN—For a General Sparse Matrix, Convert Between Diagonal-Out and Profile-In Skyline Storage Mode

This subroutine converts general sparse matrix ***A*** of order *n* from one skyline storage mode to another—that is, between the following:

- Diagonal-out skyline storage mode
- Profile-in skyline storage mode

Syntax

Fortran	CALL DGKTRN (*n*, *au*, *nu*, *idu*, *al*, *nl*, *idl*, *itran*, *aux*, *naux*)
C and C++	dgktrn (*n*, *au*, *nu*, *idu*, *al*, *nl*, *idl*, *itran*, *aux*, *naux*);
PL/I	CALL DGKTRN (*n*, *au*, *nu*, *idu*, *al*, *nl*, *idl*, *itran*, *aux*, *naux*);

On Entry

n
is the order of general sparse matrix ***A***. Specified as: a fullword integer; $n \geq 0$.

au
is the array, referred to as AU, containing the upper triangular part of general sparse matrix ***A***, stored as follows, where:

If ITRAN(1) = 0, ***A*** is stored in diagonal-out skyline storage mode.

If ITRAN(1) = 1, ***A*** is stored in profile-in skyline storage mode.

Specified as: a one-dimensional array of (at least) length *nu*, containing long-precision real numbers.

nu
is the length of array AU. Specified as: a fullword integer; $nu \geq 0$ and $nu \geq (IDU(n+1)-1)$.

idu
is the array, referred to as IDU, containing the relative positions of the diagonal elements of matrix ***A*** in input array AU. Specified as: a one-dimensional array of (at least) length $n+1$, containing fullword integers.

al
is the array, referred to as AL, containing the lower triangular part of general sparse matrix ***A***, stored as follows, where:

If ITRAN(1) = 0, ***A*** is stored in diagonal-out skyline storage mode.

If ITRAN(1) = 1, ***A*** is stored in profile-in skyline storage mode.

Note: Entries in AL for diagonal elements of ***A*** are assumed not to have meaningful values.

Specified as: a one-dimensional array of (at least) length *nl*, containing long-precision real numbers.

nl
is the length of array AL. Specified as: a fullword integer; $nl \geq 0$ and $nl \geq (IDL(n+1)-1)$.

idl
is the array, referred to as IDL, containing the relative positions of the diagonal elements of matrix ***A*** in input array AL. Specified as: a one-dimensional array of (at least) length $n+1$, containing fullword integers.

itran
is an array of parameters, ITRAN(*i*), where:

- ITRAN(1) indicates the input storage mode used for matrix ***A***. This determines the arrangement of data in arrays AU, IDU, AL, and IDL on input, where:

  If ITRAN(1) = 0, diagonal-out skyline storage mode is used.

  If ITRAN(1) = 1, profile-in skyline storage mode is used.

- ITRAN(2) indicates the output storage mode used for matrix ***A***. This determines the arrangement of data in arrays AU, IDU, AL, and IDL on output, where:

  If ITRAN(2) = 0, diagonal-out skyline storage mode is used.

  If ITRAN(2) = 1, profile-in skyline storage mode is used.

- ITRAN(3) indicates the direction of sweep that ESSL uses through the matrix ***A***, allowing you to optimize performance (see "Notes" on page 992), where:

  If ITRAN(3) = 1, matrix ***A*** is transformed in the positive direction, starting in row or column 1 and ending in row or column *n*.

  If ITRAN(3) = −1, matrix ***A*** is transformed in the negative direction, starting in row or column *n* and ending in row or column 1.

Specified as: a one-dimensional array of (at least) length 3, containing fullword integers, where:
ITRAN(1) = 0 or 1
ITRAN(2) = 0 or 1
ITRAN(3) = −1 or 1

aux
has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is the storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing *naux* long-precision real numbers.

naux
is the size of the work area specified by *aux*—that is, the number of elements in *aux*.
Specified as: a fullword integer, where:

If *naux* = 0 and error 2015 is unrecoverable, DGKTRN dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux* ≥ 2*n*.

On Return

au
is the array, referred to as AU, containing the upper triangular part of general sparse matrix ***A***, stored as follows, where:

If ITRAN(2) = 0, ***A*** is stored in diagonal-out skyline storage mode.

If ITRAN(2) = 1, ***A*** is stored in profile-in skyline storage mode.

Returned as: a one-dimensional array of (at least) length *nu*, containing long-precision real numbers.

idu
is the array, referred to as IDU, containing the relative positions of the diagonal elements of matrix ***A*** in output array AU. Returned as: a one-dimensional array of (at least) length *n*+1, containing fullword integers.

al
is the array, referred to as AL, containing the lower triangular part of general sparse matrix ***A***, stored as follows, where:

If ITRAN(2) = 0, ***A*** is stored in diagonal-out skyline storage mode.

If ITRAN(2) = 1, ***A*** is stored in profile-in skyline storage mode.

Note: You should assume that entries in AL for diagonal elements of ***A*** do not have meaningful values.

Returned as: a one-dimensional array of (at least) length *nl*, containing long-precision real numbers.

idl
is the array, referred to as IDL, containing the relative positions of the diagonal elements of matrix ***A*** in output array AL. Returned as: a one-dimensional array of (at least) length *n*+1, containing fullword integers.

Notes

1. Your various arrays must have no common elements; otherwise, results are unpredictable.
2. The ITRAN(3) argument allows you to specify the direction of travel through matrix ***A*** that ESSL takes during the transformation. By properly specifying ITRAN(3), you can optimize the performance of the transformation, which is especially beneficial when transforming large matrices.

   The direction specified by ITRAN(3) should be opposite the most recent direction of access through the matrix performed by the DGKFS or DGKFSP subroutine, as indicated in the following table:

Most Recent Computation Performed by DGKFS/DGKFSP	Direction Used by DGKFS/DGKFSP	Direction to Specify in ITRAN(3)
Factor and Solve	Negative	Positive (ITRAN(3) = 1)
Factor Only	Positive	Negative (ITRAN(3) = −1)
Solve Only	Negative	Positive (ITRAN(3) = 1)

3. For a description of how sparse matrices are stored in skyline storage mode, see "Profile-In Skyline Storage Mode" on page 103 and "Diagonal-Out Skyline Storage Mode" on page 100.
4. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: A general sparse matrix ***A***, stored in diagonal-out or profile-in skyline storage mode is converted to either of these same two storage modes. (Generally, you convert from one to the other, but the capability exists to specify the same storage mode

for input and output.) The argument `ITRAN(3)` indicates the direction in which you want the transformation performed on matrix ***A***, allowing you to optimize your performance in this subroutine. This is especially beneficial for large matrices.

This subroutine is meant to be used in conjunction with DGKFS and DGKFSP, which process matrices stored in these skyline storage modes.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors: None

Input-Argument Errors

1. $n < 0$
2. $nu < 0$
3. `IDU`(n+1) > nu+1
4. `IDU`(i+1) ≤ `IDU`(i) for i = 1, n
5. `IDU`(i+1) > `IDU`(i)+i and `ITRAN(1)` = 0 for i = 1, n
6. `IDU`(i) > `IDU`(i−1)+i and `ITRAN(1)` = 1 for i = 2, n
7. $nl < 0$
8. `IDL`(n+1) > nl+1
9. `IDL`(i+1) ≤ `IDL`(i) for i = 1, n
10. `IDL`(i+1) > `IDL`(i)+i and `ITRAN(1)` = 0 for i = 1, n
11. `IDL`(i) > `IDL`(i−1)+i and `ITRAN(1)` = 1 for i = 2, n
12. `ITRAN(1)` ≠ 0 or 1
13. `ITRAN(2)` ≠ 0 or 1
14. `ITRAN(3)` ≠ −1 or 1
15. Error 2015 is recoverable or *naux*≠0, and *naux* is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example 1: This example shows how to convert a 9 by 9 general sparse matrix ***A*** from diagonal-out skyline storage mode to profile-in skyline storage mode. Matrix ***A*** is:

```
[                                                                  ]
| 11.0  12.0  13.0   0.0   0.0   0.0   0.0   0.0   0.0 |
| 21.0  22.0  23.0  24.0  25.0   0.0   0.0   0.0  29.0 |
| 31.0  32.0  33.0  34.0  35.0   0.0  37.0   0.0  39.0 |
| 41.0  42.0  43.0  44.0  45.0  46.0  47.0   0.0  49.0 |
|  0.0   0.0   0.0  54.0  55.0  56.0  57.0  58.0  59.0 |
|  0.0  62.0  63.0  64.0  65.0  66.0  67.0  68.0  69.0 |
|  0.0   0.0   0.0  74.0  75.0  76.0  77.0  78.0  79.0 |
|  0.0   0.0   0.0  84.0  85.0  86.0  87.0  88.0  89.0 |
| 91.0  92.0  93.0  94.0  95.0  96.0  97.0  98.0  99.0 |
[                                                                  ]
```

Assuming that DGKFS last performed a solve on matrix ***A***, the direction of the transformation is positive; that is, `ITRAN(3)` is 1. This provides the best performance here.

Note: On input and output, the diagonal elements in `AL` do not have meaningful values.

Call Statement and Input

```
             N   AU   NU   IDU   AL   NL   IDL   ITRAN   AUX   NAUX
             |   |    |    |     |    |    |     |       |     |
CALL DGKTRN( 9 , AU , 33 , IDU , AL , 35 , IDL , ITRAN , AUX ,  18


AU       =  (11.0, 22.0, 12.0, 33.0, 23.0, 13.0, 44.0, 34.0, 24.0,
             55.0, 45.0, 35.0, 25.0, 66.0, 56.0, 46.0, 77.0, 67.0,
             57.0, 47.0, 37.0, 88.0, 78.0, 68.0, 58.0, 99.0, 89.0,
             79.0, 69.0, 59.0, 49.0, 39.0, 29.0)
IDU      =  (1, 2, 4, 7, 10, 14, 17, 22, 26, 34)
AL       =  ( . , . , 21.0, . , 32.0, 31.0, . , 43.0, 42.0, 41.0, . ,
             54.0, . , 65.0, 64.0, 63.0, 62.0, . , 76.0, 75.0, 74.0,
             . , 87.0, 86.0, 85.0, 84.0, . , 98.0, 97.0, 96.0, 95.0,
             94.0, 93.0, 92.0, 91.0)
IDL      =  (1, 2, 4, 7, 11, 13, 18, 22, 27, 36)
ITRAN    =  (0, 1, 1)
```

Output

```
AU       =  (11.0, 12.0, 22.0, 13.0, 23.0, 33.0, 24.0, 34.0, 44.0,
             25.0, 35.0, 45.0, 55.0, 46.0, 56.0, 66.0, 37.0, 47.0,
             57.0, 67.0, 77.0, 58.0, 68.0, 78.0, 88.0, 29.0, 39.0,
             49.0, 59.0, 69.0, 79.0, 89.0, 99.0)
IDU      =  (1, 3, 6, 9, 13, 16, 21, 25, 33, 34)
AL       =  ( . , 21.0, . , 31.0, 32.0, . , 41.0, 42.0, 43.0, . , 54.0,
             . , 62.0, 63.0, 64.0, 65.0, . , 74.0, 75.0, 76.0, . ,
             84.0, 85.0, 86.0, 87.0, . , 91.0, 92.0, 93.0, 94.0, 95.0,
             96.0, 97.0, 98.0, . )
IDL      =  (1, 3, 6, 10, 12, 17, 21, 26, 35, 36)
```

Example 2: This example shows how to convert the same 9 by 9 general sparse matrix ***A*** in Example 1 from profile-in skyline storage mode to diagonal-out skyline storage mode.

Assuming that DGKFS last performed a factorization on matrix ***A***, the direction of the transformation is negative; that is, `ITRAN(3)` is –1. This provides the best performance here.

Note: On input and output, the diagonal elements in `AL` do not have meaningful values.

Call Statement and Input

```
             N   AU   NU   IDU   AL   NL   IDL   ITRAN   AUX   NAUX
             |   |    |    |     |    |    |     |       |     |
CALL DGKTRN( 9 , AU , 33 , IDU , AL , 35 , IDL , ITRAN , AUX ,  18


AU       =(same as output AU in Example 1)
IDU      =(same as output IDU in Example 1)
AL       =(same as output AL in Example 1)
IDL      =(same as output IDL in Example 1)
ITRAN    =  (1, 0, -1)
```

Output

AU	=(same as input AU in Example 1)
IDU	=(same as input IDU in Example 1)
AL	=(same as input AL in Example 1)
IDL	=(same as input IDL in Example 1)

DSKTRN—For a Symmetric Sparse Matrix, Convert Between Diagonal-Out and Profile-In Skyline Storage Mode

This subroutine converts symmetric sparse matrix ***A*** of order *n* from one skyline storage mode to another—that is, between the following:

- Diagonal-out skyline storage mode
- Profile-in skyline storage mode

Syntax

Fortran	CALL DSKTRN (*n*, *a*, *na*, *idiag*, *itran*, *aux*, *naux*)
C and C++	dsktrn (*n*, *a*, *na*, *idiag*, *itran*, *aux*, *naux*);
PL/I	CALL DSKTRN (*n*, *a*, *na*, *idiag*, *itran*, *aux*, *naux*);

On Entry

n
is the order of symmetric sparse matrix ***A***. Specified as: a fullword integer; $n \geq 0$.

a
is the array, referred to as A, containing the upper triangular part of symmetric sparse matrix ***A***, stored as follows, where:

If ITRAN(1) = 0, ***A*** is stored in diagonal-out skyline storage mode.

If ITRAN(1) = 1, ***A*** is stored in profile-in skyline storage mode.

Specified as: a one-dimensional array of (at least) length *na*, containing long-precision real numbers.

na
is the length of array A. Specified as: a fullword integer; $na \geq 0$ and $na \geq (IDIAG(n+1)-1)$.

idiag
is the array, referred to as IDIAG, containing the relative positions of the diagonal elements of matrix ***A*** in input array A. Specified as: a one-dimensional array of (at least) length *n*+1, containing fullword integers.

itran
is an array of parameters, ITRAN(*i*), where:

- ITRAN(1) indicates the input storage mode used for matrix ***A***. This determines the arrangement of data in arrays A and IDIAG on input, where:

  If ITRAN(1) = 0, diagonal-out skyline storage mode is used.

  If ITRAN(1) = 1, profile-in skyline storage mode is used.

- ITRAN(2) indicates the output storage mode used for matrix ***A***. This determines the arrangement of data in arrays A and IDAIG on output, where:

  If ITRAN(2) = 0, diagonal-out skyline storage mode is used.

  If ITRAN(2) = 1, profile-in skyline storage mode is used.

- ITRAN(3) indicates the direction of sweep that ESSL uses through the matrix ***A***, allowing you to optimize performance (see "Notes" on page 997), where:

  If ITRAN(3) = 1, matrix ***A*** is transformed in the positive direction, starting in row or column 1 and ending in row or column *n*.

If ITRAN(3) = −1, matrix ***A*** is transformed in the negative direction, starting in row or column *n* and ending in row or column 1.

Specified as: a one-dimensional array of (at least) length 3, containing fullword integers, where:

ITRAN(1) = 0 or 1
ITRAN(2) = 0 or 1
ITRAN(3) = −1 or 1

aux

has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is the storage work area used by this subroutine. Its size is specified by *naux*.

Specified as: an area of storage, containing *naux* long-precision real numbers.

naux

is the size of the work area specified by *aux*—that is, the number of elements in *aux*. Specified as: a fullword integer, where:

If *naux* = 0 and error 2015 is unrecoverable, DSKTRN dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux* ≥ *n*.

On Return

a

is the array, referred to as A, containing the upper triangular part of symmetric sparse matrix ***A***, stored as follows, where:

If ITRAN(2) = 0, ***A*** is stored in diagonal-out skyline storage mode.

If ITRAN(2) = 1, ***A*** is stored in profile-in skyline storage mode.

Returned as: a one-dimensional array of (at least) length *na*, containing long-precision real numbers.

idiag

is the array, referred to as IDIAG, containing the relative positions of the diagonal elements of matrix ***A*** in output array A. Returned as: a one-dimensional array of (at least) length *n*+1, containing fullword integers.

Notes

1. Your various arrays must have no common elements; otherwise, results are unpredictable.
2. The ITRAN(3) argument allows you to specify the direction of travel through matrix ***A*** that ESSL takes during the transformation. By properly specifying ITRAN(3), you can optimize the performance of the transformation, which is especially beneficial when transforming large matrices.

   The direction specified by ITRAN(3) should be opposite the most recent direction of access through the matrix performed by the DSKFS or DSKFSP subroutine, as indicated in the following table:

Most Recent Computation Performed by DSKFS/DSKFSP	Direction Used by DSKFS/DSKFSP	Direction to Specify in ITRAN(3)
Factor and Solve	Negative	Positive (ITRAN(3) = 1)
Factor Only	Positive	Negative (ITRAN(3) = −1)
Solve Only	Negative	Positive (ITRAN(3) = 1)

3. For a description of how sparse matrices are stored in skyline storage mode, see "Profile-In Skyline Storage Mode" on page 103 and "Diagonal-Out Skyline Storage Mode" on page 100.
4. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 31.

Function: A symmetric sparse matrix ***A***, stored in diagonal-out or profile-in skyline storage mode is converted to either of these same two storage modes. (Generally, you convert from one to the other, but the capability exists to specify the same storage mode for input and output.) The argument ITRAN(3) indicates the direction in which you want the transformation performed on matrix ***A***, allowing you to optimize your performance in this subroutine. This is especially beneficial for large matrices.

This subroutine is meant to be used in conjunction with DSKFS and DSKFSP, which process matrices stored in these skyline storage modes.

Error Conditions

Resource Errors: Error 2015 is unrecoverable, *naux* = 0, and unable to allocate work area.

Computational Errors: None

Input-Argument Errors

1. $n < 0$
2. $na < 0$
3. IDIAG(n+1) > na+1
4. IDIAG(i+1) ≤ IDIAG(i) for $i = 1, n$
5. IDIAG(i+1) > IDIAG(i)+i and ITRAN(1) = 0 for $i = 1, n$
6. IDIAG(i) > IDIAG(i−1)+i and ITRAN(1) = 1 for $i = 2, n$
7. ITRAN(1) ≠ 0 or 1
8. ITRAN(2) ≠ 0 or 1
9. ITRAN(3) ≠ −1 or 1
10. *naux* Error 2015 is recoverable or *naux*≠0, and is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Example 1: This example shows how to convert a 9 by 9 symmetric sparse matrix ***A*** from diagonal-out skyline storage mode to profile-in skyline storage mode. Matrix ***A*** is:

```
⎡                                                                  ⎤
| 11.0  12.0  13.0  14.0   0.0   0.0   0.0   0.0   0.0 |
| 12.0  22.0  23.0  24.0  25.0  26.0   0.0  28.0   0.0 |
| 13.0  23.0  33.0  34.0  35.0  36.0   0.0  38.0   0.0 |
| 14.0  24.0  34.0  44.0  45.0  46.0   0.0  48.0   0.0 |
|  0.0  25.0  35.0  45.0  55.0  56.0  57.0  58.0   0.0 |
|  0.0  26.0  36.0  46.0  56.0  66.0  67.0  68.0  69.0 |
|  0.0   0.0   0.0   0.0  57.0  67.0  77.0  78.0  79.0 |
|  0.0  28.0  38.0  48.0  58.0  68.0  78.0  88.0  89.0 |
|  0.0   0.0   0.0   0.0   0.0  69.0  79.0  89.0  99.0 |
⎣                                                                  ⎦
```

Assuming that DSKFS last performed a factorization on matrix ***A***, the direction of the transformation is negative; that is, `ITRAN(3)` is –1. This provides the best performance here.

Call Statement and Input

```
             N   A   NA   IDIAG   ITRAN   AUX   NAUX
             |   |   |      |       |      |     |
CALL DSKTRN( 9 , A , 33 , IDIAG , ITRAN , AUX ,  9  )


A        =  (11.0, 22.0, 12.0, 33.0, 23.0, 13.0, 44.0, 34.0, 24.0,
             14.0, 55.0, 45.0, 35.0, 25.0, 66.0, 56.0, 46.0, 36.0,
             26.0, 77.0, 67.0, 57.0, 88.0, 78.0, 68.0, 58.0, 48.0,
             38.0, 28.0, 99.0, 89.0, 79.0, 69.0)
IDIAG    =  (1, 2, 4, 7, 11, 15, 20, 23, 30, 34)
ITRAN    =  (0, 1, -1)
```

Output

```
A        =  (11.0, 12.0, 22.0, 13.0, 23.0, 33.0, 14.0, 24.0, 34.0,
             44.0, 25.0, 35.0, 45.0, 55.0, 26.0, 36.0, 46.0, 56.0,
             66.0, 57.0, 67.0, 77.0, 28.0, 38.0, 48.0, 58.0, 68.0,
             78.0, 88.0, 69.0, 79.0, 89.0, 99.0)
IDIAG    =  (1, 3, 6, 10, 14, 19, 22, 29, 33, 34)
```

Example 2: This example shows how to convert the same 9 by 9 symmetric sparse matrix ***A*** in Example 1 from profile-in skyline storage mode to diagonal-out skyline storage mode.

Assuming that DSKFS last performed a solve on matrix ***A***, the direction of the transformation is positive; that is, `ITRAN(3)` is 1. This provides the best performance here.

Call Statement and Input

```
             N   A   NA   IDIAG   ITRAN   AUX   NAUX
             |   |   |      |       |      |     |
CALL DSKTRN( 9 , A , 33 , IDIAG , ITRAN , AUX ,  9
```

```
A       =(same as output A in Example 1)
IDIAG   =(same as output IDIAG in Example 1)
ITRAN   =  (1, 0, 1)
```

Output

```
A       =(same as input A in Example 1)
IDIAG   =(same as input IDIAG in Example 1)
```

Part 3. Appendixes

© Copyright IBM Corp. 1997

Appendix A. Basic Linear Algebra Subprograms (BLAS)

This appendix lists the ESSL subprograms corresponding to a subprogram in the standard set of BLAS.

Level 1 BLAS

Table 168. Level 1 BLAS Included in ESSL

Descriptive Name	Short-Precision Subprogram	Long-Precision Subprogram
Position of the First or Last Occurrence of the Vector Element Having the Largest Magnitude	ISAMAX ICAMAX	IDAMAX IZAMAX
Sum of the Magnitudes of the Elements in a Vector	SASUM SCASUM	DASUM DZASUM
Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in the Vector Y	SAXPY CAXPY	DAXPY ZAXPY
Copy a Vector	SCOPY CCOPY	DCOPY ZCOPY
Dot Product of Two Vectors	SDOT CDOTU CDOTC	DDOT ZDOTU ZDOTC
Euclidean Length of a Vector with Scaling of Input to Avoid Destructive Underflow and Overflow	SNRM2 SCNRM2	DNRM2 DZNRM2
Construct a Givens Plane Rotation	SROTG CROTG	DROTG ZROTG
Apply a Plane Rotation	SROT CROT CSROT	DROT ZROT ZDROT
Multiply a Vector X by a Scalar and Store in the Vector X	SSCAL CSCAL CSSCAL	DSCAL ZSCAL ZDSCAL
Interchange the Elements of Two Vectors	SSWAP CSWAP	DSWAP ZSWAP

© Copyright IBM Corp. 1997

Level 2 BLAS

Table 169. Level 2 BLAS Included in ESSL

Descriptive Name	Short-Precision Subprogram	Long-Precision Subprogram
Matrix-Vector Product for a General Matrix, Its Transpose, or Its Conjugate Transpose	SGEMV CGEMV	DGEMV ZGEMV
Rank-One Update of a General Matrix	SGER CGERU CGERC	DGER ZGERU ZGERC
Matrix-Vector Product for a Real Symmetric or Complex Hermitian Matrix	SSPMV CHPMV SSYMV CHEMV	DSPMV ZHPMV DSYMV ZHEMV
Rank-One Update of a Real Symmetric or Complex Hermitian Matrix	SSPR CHPR SSYR CHER	DSPR ZHPR DSYR ZHER
Rank-Two Update of a Real Symmetric or Complex Hermitian Matrix	SSPR2 CHPR2 SSYR2 CHER2	DSPR2 ZHPR2 DSYR2 ZHER2
Matrix-Vector Product for a General Band Matrix, Its Transpose, or Its Conjugate Transpose	SGBMV CGBMV	DGBMV ZGBMV
Matrix-Vector Product for a Real Symmetric or Complex Hermitian Band Matrix	SSBMV CHBMV	DSBMV ZHBMV
Matrix-Vector Product for a Triangular Matrix, Its Transpose, or Its Conjugate Transpose	STPMV CTPMV STRMV CTRMV	DTPMV ZTPMV DTRMV ZTRMV
Solution of a Triangular System of Equations with a Single Right-Hand Side	STPSV CTPSV STRSV CTRSV	DTPSV ZTPSV DTRSV ZTRSV
Matrix-Vector Product for a Triangular Band Matrix, Its Transpose, or Its Conjugate Transpose	STBMV CTBMV	DTBMV ZTBMV
Triangular Band Equation Solve	STBSV CTBSV	DTBSV ZTBSV

Level 3 BLAS

Table 170. Level 3 BLAS Included in ESSL

Descriptive Name	Short-Precision Subprogram	Long-Precision Subprogram
Combined Matrix Multiplication and Addition for General Matrices, Their Transposes, or Conjugate Transposes	SGEMM CGEMM	DGEMM ZGEMM
Matrix-Matrix Product Where One Matrix is Real or Complex Symmetric or Complex Hermitian	SSYMM CSYMM CHEMM	DSYMM ZSYMM ZHEMM
Triangular Matrix-Matrix Product	STRMM CTRMM	DTRMM ZTRMM
Rank-K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix	SSYRK CSYRK CHERK	DSYRK ZSYRK ZHERK
Rank-2K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix	SSYR2K CSYR2K CHER2K	DSYR2K ZSYR2K ZHER2K
Solution of Triangular Systems of Equations with Multiple Right-Hand Sides	STRSM CTRSM	DTRSM ZTRSM

Appendix B. LAPACK

This appendix lists the ESSL subroutines corresponding to subroutines in the standard set of LAPACK.

LAPACK

Table 171. LAPACK Included in ESSL

Descriptive Name	Short-Precision Subprogram	Long-Precision Subprogram
General Matrix Factorization	SGETRF CGETRF	DGETRF ZGETRF
General Matrix, Its Transpose, or Its Conjugate Transpose Multiple Right-Hand Side Solve	SGETRS CGETRS	DGETRS ZGETRS

© Copyright IBM Corp. 1997

Glossary

This glossary defines terms and abbreviations used in this publication. If you do not find the term you are looking for, refer to the index portion of this book. This glossary includes terms and definitions from:

- *IBM Dictionary of Computing,* New York: McGraw Hill (1-800-2MC-GRAW), 1994.
- *American National Standard Dictionary for Information Systems,* ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42nd Street, New York, New York 10036. Definitions are identified by the symbol (A) after the definition.
- *Information Technology Vocabulary,* developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions from published sections of these vocabularies are identified by the symbol (I) after the definition. Definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

APAR. Authorized Program Analysis Report. A report of a problem caused by a suspected defect in a current unaltered release of a program.

argument. A parameter passed between a calling program and a SUBROUTINE subprogram, a FUNCTION subprogram, or a statement function.

array. An ordered set of data items identified by a single name.

array element. A data item in an array, identified by the array name followed by a subscript indicating its position in the array.

array name. The name of an ordered set of data items that make up an array.

assignment statement. A statement that assigns a value to a variable or array element. It is made up of a variable or array element, followed by an equal sign (=), followed by an expression. The variable, array element, or expression can be character, logical, or arithmetic. When the assignment statement is processed, the expression to the right of the equal sign replaces the value of the variable or array element to the left.

Basic Linear Algebra Subprograms (BLAS). A standard, public domain, set of mathematical subroutines that perform linear algebra operations.

BLAS. Basic Linear Algebra Subprograms.

cache. A special-purpose buffer storage, smaller and faster than main storage, used to hold a copy of instructions and data obtained from main storage and likely to be needed next by the processor. (T)

character constant. A string of one or more alphanumeric characters enclosed in apostrophes. The delimiting apostrophes are not part of the value of the constant.

character expression. An expression in the form of a single character constant, variable, array element, substring, function reference, or another expression enclosed in parentheses. A character expression is always of type character.

character type. The data type for representing strings of alphanumeric characters; in storage, one byte is used for each character.

column-major order. A sequencing method used for storing multidimensional arrays according to the subscripts of the array elements. In this method the leftmost subscript position varies most rapidly and completes a full cycle before the next subscript position to the right is incremented.

complex conjugate even data. Complex data that has its real part even and its imaginary part odd.

complex constant. An ordered pair of real or integer constants separated by a comma and enclosed in parentheses. The first real constant of the pair is the real part of the complex number; the second is the imaginary part.

complex type. The data type for representing an approximation of the value of a complex number. A data item of this type consists of an ordered pair of real data items separated by a comma and enclosed in parentheses. The first item represents the real part of the complex number; the second represents the imaginary part.

constant. An unvarying quantity. The four classes of constants specify numbers (arithmetic), truth values (logical), character data (character), and hexadecimal data.

data type. The structural characteristics, features and properties of data that may be directly specified by a programming language; for example, integers, real numbers in Fortran; arrays in APL; linked lists in LISP; character string in SNOBOL.

© Copyright IBM Corp. 1997

decimation. The formation of a sequence containing every n-th element of another sequence.

dimension of an array. One of the subscript expression positions in a subscript for an array. In Fortran, an array may have from one to seven dimensions. Graphically, the first dimension is represented by the rows, the second by the columns, and the third by the planes. Contrast with rank. See also extent of a dimension.

direct access storage. A storage device in which the access time is in effect independent of the location of the data. (A)

divide-by-zero exception. The condition recognized by a processor that results from running a program that attempts to divide by zero.

double precision. Synonym for long-precision.

expression. A notation that represents a value: a primary appearing alone, or combinations of primaries and operators. An expression can be arithmetic, character, logical, or relational.

extent of a dimension. The number of different integer values that may be represented by subscript expressions for a particular dimension in a subscript for an array.

external function. A function defined outside the program unit that refers to it. It may be referred to in a procedure subprogram or in the main program, but it must not refer to itself, either directly or indirectly. Contrast with statement function.

function. In Fortran, a procedure that is invoked by referring to it in an expression and that supplies a value to the expression. The value supplied is the value of the function. See also external function, intrinsic function, and statement function. Contrast with subroutine.

function reference. A Fortran source program reference to an intrinsic function, to an external function, or to a statement function.

general matrix. A matrix with no assumed special properties such as symmetry. Synonym for matrix.

integer constant. A string of decimal digits containing no decimal point and expressing a whole number.

integer expression. An arithmetic expression whose values are of integer type.

integer type. An arithmetic data type capable of expressing the value of an integer. It can have a positive, negative, or 0 value. It must not include a decimal point.

intrinsic function. A function, supplied by Fortran, that performs mathematical or character operations.

leading dimension. For a two-dimensional array, an increment used to find the starting point for the matrix elements in each successive column of the array.

logical constant. A constant that can have one of two values: true or false. The form of these values in Fortran is: .TRUE. and .FALSE. respectively.

logical expression. A logical primary alone or a combination of logical primaries and logical operators. A logical expression can have one of two values: true or false.

logical type. The data type for data items that can have the value true or false and upon which logical operations such as .NOT. and .OR. can be performed. See also “data type.”

long-precision. Real type of data of length 8. Contrast with single precision and short-precision.

main program. In Fortran, a program unit, required for running, that can call other program units but cannot be called by them.

mask. To use a pattern of characters to control the retention or elimination of portions of another pattern of characters. (I)

matrix. A rectangular array of elements, arranged in rows and columns, that may be manipulated according to the rules of matrix algebra. (A) (I)

multithreaded. There may be one or more threads in a process, and each thread is executed by the operating system concurrently. An application program is multithreaded if more than one thread is executed concurrently.

overflow exception. A condition caused by the result of an arithmetic operation having a magnitude that exceeds the largest possible number.

platform. A mainframe or a workstation.

primary. An irreducible unit of data; a single constant, variable, array element, function reference, or expression enclosed in parentheses.

program exception. The condition recognized by a processor that results from running a program that improperly specifies or uses instructions, operands, or control information.

PTF. Program Temporary Fix. A temporary solution or by-pass of a problem diagnosed by IBM as resulting from a defect in a current unaltered release of the program. A report of a problem caused by a suspected defect in a current unaltered release of a program.

pthread. A thread that conforms to the POSIX Threads Programming Model.

real constant. A string of decimal digits that expresses a real number. A real constant must contain either a decimal point or a decimal exponent and may contain both. For example, the real constant 0.36819E+2 has the value +36.819.

real type. An arithmetic data type, capable of approximating the value of a real number. It can have a positive, negative, or 0 value.

row-major order. A sequencing method used for storing multidimensional arrays according to the subscripts of the array elements. In this method the rightmost subscript position varies most rapidly and completes a full cycle before the next subscript position to the left is incremented.

scalar. (1) A quantity characterized by a single number. (A) (I) (2) Contrast with vector.

shape of an array. The extents of all the dimensions of an array listed in order. For example, the shape of a three-dimensional array that has four rows, five columns, and three planes is (4,5,3) or 4 by 5 by 3.

short-precision. Real type data of length 4. Contrast with double precision and long-precision.

single precision. Synonym for short-precision.

size of an array. The number of elements in an array. This is the product of the extents of its dimensions.

SMP. Symmetric Multi-Processing.

statement. The basic unit of a program, that specifies an action to be performed, or the nature and characteristics of the data to be processed, or information about the program itself. Statements fall into two broad classes: executable and nonexecutable.

statement function. A procedure specified by a single statement that is similar in form to an arithmetic, logical, or character assignment statement. The statement must appear after the specification statements and before the first executable statement. In the remainder of the program it can be referenced as a function. A statement function may be referred to only in the program unit in which it is defined. Contrast with external function.

statement label. A number of from one through five decimal digits that is used to identify a statement. Statement labels can be used to transfer control, to define the range of a DO, or to refer to a FORMAT statement.

statement number. See “statement label.”

stride. The increment used to step through array storage to select the vector or matrix elements from the array.

subprogram. A program unit that is invoked by another program unit in the same program. In Fortran, a subprogram has a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement.

subscript. (1) A symbol that is associated with the name of a set to identify a particular subset or element. (A) (2) A subscript expression or set of subscript expressions, enclosed in parentheses and used with an array name to identify a particular array element.

subscript expression. An integer expression in a subscript whose value and position in the subscript determine the index number for the corresponding dimension in the referenced array.

thread. A thread is the element that is scheduled, and to which resources such as execution time, locks, and queues may be assigned. There may be one or more threads in a process, and each thread is executed by the operating system concurrently.

thread-safe. A subroutine which may be called from multiple threads of the same process simultaneously.

type declaration. The explicit specification of the type of a constant, variable, array, or function by use of an explicit type specification statement.

underflow exception. A condition caused by the result of an arithmetic operation having a magnitude less than the smallest possible nonzero number.

variable. (1) A quantity that can assume any of a given set of values. (A) (2) A data item, identified by a name, that is not a named constant, array, or array element, and that can assume different values at different times during program processing.

vector. A one-dimensional ordered collection of numbers.

working storage. A storage area provided by the application program for the use of an ESSL subroutine.

workstation. A workstation is a single-user, high-performance microcomputer (or even a minicomputer) which has been specialized in some way, usually for graphics output. Such a machine has a screen and a keyboard, but is also capable of extensive processing of your input before it is passed to the host. Likewise, the host's responses may be extensively processed before being passed along to your screen. A workstation may be intelligent enough to do much or all the processing itself.

Bibliography

This bibliography lists the publications that you may need to use with ESSL and describes how to obtain them.

References

Text books and articles covering the mathematical aspects of ESSL are listed in this section, as well as several software libraries available from other companies. They are listed alphabetically as follows:

- Publications are listed by the author's name. IBM publications that include an order number, other than an *IBM Technical Report* can be ordered through the Subscription Library Services System (SLSS). The non-IBM publications listed here should be obtained through publishers, bookstores, or professional computing organizations.
- Software libraries are listed by their product name. Each reference includes the names, addresses, and phone numbers of the companies from which they can be obtained.

Each citation in the text of this book is shown as a number enclosed in square brackets. It indicates the number of the item listed in the bibliography. For example, reference [1] cites the first item listed below.

1. Agarwal, R. C. Dec. 1984. "An Efficient Formulation of the Mixed-Radix FFT Algorithm." *Proceedings of the International Conference on Computers, Systems, and Signal Processing*, 769–772. Bangalore, India.
2. Agarwal, R. C. August 1988. "A Vector and Parallel Implementation of the FFT Algorithm on the IBM 3090." *Proceedings from the IFIP WG 2.5 (International Federation for Information Processing Working Conference 5)*, Stanford University.
3. Agarwal, R. C. 1989. "A Vector and Parallel Implementation of the FFT Algorithm on the IBM 3090." *Aspects of Computation on Asynchronous Parallel Processors*, 45–54. Edited by M. H. Wright. Elsevier Science Publishers, New York, N. Y.
4. Agarwal, R. C.; Cooley, J. W. March 1986. "Fourier Transform and Convolution Subroutines for the IBM 3090 Vector Facility." *IBM Journal of Research and Development*, 30(2):145–162 (Order no. G322-0146).
5. Agarwal, R. C.; Cooley, J. W. September 1987. "Vectorized Mixed-Radix Discrete Fourier Transform Algorithms" *IEEE Proceedings*, 75:1283–1292.
6. Agarwal, R.; Cooley, J.; Gustavson F.; Shearer J.; Slishman G.; Tuckerman B. March 1986. "New Scalar and Vector Elementary Functions for the IBM System/370." *IBM Journal of Research and Development*, 30(2):126–144 (Order no. G322-0146).
7. Agarwal, R.; Gustavson F.; Zubair, M. May 1994. "An Efficient Parallel Algorithm for the 3-D FFT NAS Parallel Benchmark." *Proceedings of IEEE SHPCC 94* :129–133.
8. Anderson, E.; Bai, Z.; Bischof, C.; Demmel, J.; Dongarra, J.; DuCroz, J.; Greenbaum, A.; Hammarling, S.; McKenney, A.; Ostrouchov, S.; Sorensen, D. 1995. *LAPACK User's Guide* (second edition), SIAM Publications, Philadelphia, Pa. (For more information, see http://www.netlib.org/lapack/index.html.)
9. Bathe, K.; Wilson, E. L. 1976. *Numerical Methods in Finite Element Analysis*, 249–258.
10. Brayton, R. K.; Gustavson F. G.; Willoughby, R. A.; 1970. "Some Results on Sparse Matrices." *Mathematics of Computation*, 24(112):937–954.
11. Borodin, A.; Munro, I. 1975. *The Computational Complexity of Algebraic and Numeric Problems* American Elsevier, New York, N. Y.
12. Carey, G. F.; Oden, J. T. 1984. *Finite Elements: Computational Aspects, Vol 3*, 144–147. Prentice Hall, Englewood Cliffs, N. J.
13. Chan, T. F. March 1982. "An Improved Algorithm for Computing the Singular Value Decomposition." *ACM Transactions on Mathematical Software* 8(1):72–83.
14. Cline, A. K.; Moler, C. B.; Stewart, G. W.; Wilkinson, J. H. 1979. "An Estimate for the Condition Number of a Matrix." *SIAM Journal of Numerical Analysis* 16:368–375.
15. Conte, S. D.; DeBoor, C. 1972. *Elementary Numerical Analysis: An Algorithmic Approach* (second edition), McGraw-Hill, New York, N. Y.
16. Cooley, J. W. 1976. "Fast Fourier Transform." *Encyclopedia of Computer Sciences* Edited by A. Ralston. Auerbach Publishers.
17. Cooley, J. W.; Lewis, P. A. W.; Welch, P. D. June 1967. "Application of the Fast Fourier Transform to Computation of Fourier Integrals, Fourier Series, and Convolution Integrals." *IEEE Transactions Audio Electroacoustics* AU-15:79–84.
18. Cooley, J. W.; Lewis, P. A. W.; Welch, P. D. June 1967. "Historical Notes on the Fast Fourier Transform." *IEEE Transactions Audio Electroacoustics* AU-15:76–79. (Also published Oct. 1967 in *Proceedings of IEEE* 55(10):1675–1677.)
19. Cooley, J. W.; Lewis, P. A. W.; Welch, P. D. March 1969. "The Fast Fourier Transform Algorithm and its

© Copyright IBM Corp. 1997

Applications." *IEEE Transactions on Education* E12:27–34.

20. Cooley, J. W.; Lewis, P. A. W.; Welch, P. D. June 1969. "The Finite Fast Fourier Transform." *IEEE Transactions Audio Electroacoustics* AU-17:77–85.

21. Cooley, J. W.; Lewis, P. A. W.; Welch, P. D. July 1970. "The Fast Fourier Transform: Programming Considerations in the Calculation of Sine, Cosine, and LaPlace Transforms." *Journal of Sound Vibration and Analysis* 12(3):315–337.

22. Cooley, J. W.; Lewis, P. A. W.; Welch, P. D. July 1970. "The Application of the Fast Fourier Transform Algorithm to the Estimation of Spectra and Cross-Spectra." *Journal of Sound Vibration and Analysis* 12(3):339–352.

23. Cooley, J. W.; Lewis, P. A. W.; Welch, P. D. 1977. "Statistical Methods for Digital Computers." *Mathematical Methods for Digital Computers* Chapter 14. Edited by Ensein, Ralston and Wilf, Wiley-Interscience. John Wiley, New York.

24. Cooley, J. W.; Tukey, J. W. April 1965. "An Algorithm for the Machine Calculation of Complex Fourier Series." *Mathematics of Computation* 19:297.

25. Dahlquist, G.; Bjorck, A.; (Translated by Anderson, N.). 1974. *Numerical Methods*, Prentice Hall, Englewoods Cliffs, N. J. (For skyline subroutines, see 169–170.)

26. Davis, P. J.; Rabinowitz, P. 1984. *Methods of Numerical Integration*, (second edition), Academic Press, Orlando, Florida.

27. Delsarte, P.; Genin, Y. V. June 1986. "The Split Levinson Algorithm." *IEEE Transactions on Acoustics, Speech, and Signal Processing* ASSP-34(3):472.

28. Di Chio, P.; Filippone, S. January 1992. "A Stable Partition Sorting Algorithm." *Report No. ICE-0045* IBM European Center for Scientific and Engineering Computing, Rome, Italy.

29. Dodson, D. S.; Lewis, J. G. Jan. 1985. "Proposed Sparse Extensions to the Basic Linear Algebra Subprograms." *ACM SIGNUM Newsletter*, 20(1).

30. Dongarra, J. J. July 1997. "Performance of Various Computers Using Standard Linear Equations Software." University of Tennessee, CS-89-85. (You can download this document from http://www.netlib.org/benchmark/performance.ps.)

31. Dongarra, J. J.; Bunch, J. R.; Moler C. B.; Stewart, G. W. 1986. *LINPACK User's Guide*, SIAM Publications, Philadelphia, Pa. (For more information, see http://www.netlib.org/linpack/index.html.)

32. Dongarra, J. J.; DuCroz, J.; Hammarling, S.; Duff, I. March 1990. "A Set of Level 3 Basic Linear Algebra Subprograms." *ACM Transactions on Mathematical Software*, 16(1):1–17.

33. Dongarra, J. J.; DuCroz, J.; Hammarling, S.; Duff, I. March 1990. "Algorithm 679. A Set of Level 3 Basic Linear Algebra Subprograms: Model Implementation and Test Programs." *ACM Tranactions on Mathematical Software*, 16(1):18–28.

34. Dongarra, J. J.; DuCroz, J.; Hammarling, S.; Hanson, R. J. March 1988. "An Extended Set of Fortran Basic Linear Algebra Subprograms." *ACM Transactions on Mathematical Software*, 14(1):1–17.

35. Dongarra, J. J.; DuCroz, J.; Hammarling, S.; Hanson, R. J. March 1988. "Algorithm 656. An Extended Set of Basic Linear Algebra Subprograms: Model Implementation and Test Programs." *ACM Tranactions on Mathematical Software*, 14(1):18–32.

36. Dongarra, J. J.; Duff, I. S.; Sorensen, D. C.; Van der Vorst, H. 1991. *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM Publications, ISBN 0-89871-270-X.

37. Dongarra, J. J.; Eisenstat, S. C. May 1983. "Squeezing the Most Out of an Algorithm in Cray Fortran." *Technical Memorandum 9* Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439.

38. Dongarra, J. J.; Gustavson, F. G.; Karp, A. Jan. 1984. "Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine." *SIAM Review*, 26(1).

39. Dongarra, J. J.; Kaufman, L.; Hammarling, S. Jan. 1985. "Squeezing the Most Out of Eigenvalue Solvers on High-Performance Computers." *Technical Memorandum 46* Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439.

40. Dongarra, J. J.; Kolatis M. October 1994. "Call Conversion Interface (CCI) for LAPACK/ESSL." LAPACK Working Note 82, Department of Computer Science University of Tennessee, Knoxville, Tennessee. (You can download this document from http://www.netlib.org/lapack/lawns/lawn82.ps.)

41. Dongarra, J. J.; Kolatis M. May 1994. "IBM RS/6000-550 & -590 Performance for Selected Routines in ESSL/LAPACK/NAG/IMSL," LAPACK Working Note 71, Department of Computer Science University of Tennessee, Knoxville, Tennessee. (You can download this document from http://www.netlib.org/lapack/lawns/lawn71.ps.)

42. Dongarra, J. J; Meuer, H. W.; Strohmaier, E. June 1997. "Top500 Supercomputer Sites." University of Tennessee, UT-CS-97-365.; University of Mannheim, RUM 50/97, (You can view this document from http://www.netlib.org/benchmark/top500.html.)

43. Dongarra, J. J.; Moler, C. B. August 1983. "EISPACK—A Package for Solving Matrix Eigenvalue Problems." *Technical Memorandum 12* Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439.

44. Dongarra, J. J.; Moler, C. B; Bunch, J. R.; Stewart, G. W. 1979. *LINPACK Users' Guide*, SIAM, Philadelphia, Pa.

45. Dubrulle, A. A. 1971. "QR Algorithm with Implicit Shift." IBM licensed program: PL/MATH.

46. Dubrulle, A. A. November 1979. "The Design of Matrix Algorithms for Fortran and Virtual Storage." *IBM Palo Alto Scientific Center Technical Report* (Order no. G320-3396).

47. Duff, I. S.; Erisman, A. M.; Reid, J. K. 1986. *Direct Methods for Sparse Matrices* Oxford University Press (Claredon), Oxford. (For skyline subroutines, see 151–153.)

48. Eisenstat, S. C. March 1981. "Efficient Implementation of a Class of Preconditioned Conjugate Gradient Methods." *SIAM Journal of Scientific Statistical Computing*, 2(1).

49. EISPACK software library; National Energy Software Center, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439 (312-972-7250); International Mathematical and Statistical Libraries, Inc., Sixth Floor, GNB Building, 7500 Bellaire Boulevard, Houston, Texas 77036 (713-772-1927)

50. Filippone, S.; Santangelo, P.; Vitaletti M. Nov. 1990. "A Vectorized Long-Period Shift Register Random Number Generation." *Proceedings of Supercomputing '90*, 676–684, New York.

51. Forsythe, G. E.; Malcolm, M. A. 1977. *Computer Methods for Mathematical Computations*, Prentice Hall, Englewoods Cliffs, N. J.

52. Forsythe, G.E.; Moler, C. 1967. *Computer Solution of Linear Algebra Systems*, Prentice Hall, Englewoods Cliffs, N. J.

53. Freund, R. W. July 28, 1992. "Transpose-Free Quasi-Minimal Residual Methods for Non-Hermitian Linear Systems." *Numerical Analysis Manuscript 92-07* AT&T Bell Laboratories. (To appear in *SIAM Journal of Scientific Statistical Computing*, 1993, Vol. 14.)

54. Gans, D. 1969. *Transformations and Geometries* Appleton Century Crofts, New York.

55. Garbow, B. S.; Boyle, J. M.; Dongarra, J. J.; Moler, C. B. 1977. "Matrix Eigensystem Routines." *EISPACK Guide Extension Lecture Notes in Computer Science, Vol. 51* Springer-Verlag, New York, Heidelberg, Berlin.

56. George, A.; Liu, J. W. 1981. "Computer Solution of Large Sparse Positive Definite Systems." *Series in Computational Mathematics* Prentice-Hall, Englewood Cliffs, New Jersey.

57. Gerald, C. F.; Wheatley, P. O. 1985. *Applied Numerical Analysis* (third edition), Addison-Wesley, Reading, Mass.

58. Gill, P. E.; Miller, G. R. 1972. "An Algorithm for the Integration of Unequally Spaced Data." *Computer Journal* 15:80–83.

59. Golub, G. H.; Van Loan, C. F. 1996. *Matrix Computations*, John Hopkins University Press, Baltimore, Maryland.

60. Gregory, R. T.; Karney, D. L. 1969. *A Collection of Matrices for Testing Computational Algorithms*, Wiley-Interscience, New York, London, Sydney, Toronto.

61. Grimes, R. C.; Kincaid, D. R.; Young, D. M. 1979. *ITPACK 2.0 User's Guide*, CNA-150. Center for Numerical Analysis, University of Texas at Austin.

62. Hageman, L. A.; Young, D. M.. 1981. *Applied Iterative Methods* Academic Press, New York, N. Y.

63. Higham, N. J. 1996. *Accuracy and Stability of Numerical Algorithms*, SIAM Publications, Philadelphia, Pa.

64. Higham, N. J. December 1988. *Fortran Codes for Estimating the One-Norm of a Real or Complex Matrix, with Application to Condition Estimating* ACM Transactions on Mathematical Software, 14(4):381–396.

65. Jennings, A. 1977. *Matrix Computation for Engineers and Scientists*, 153–158, John Wiley and Sons, Ltd., New York, N. Y.

66. Kagstrom, B.; Ling, P.; Van Loan, C. 1993. "Portable High Performance GEMM-Based Level 3 BLAS," *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, 339–346. Edited by: R. Sincovec, D. Keyes, M. Leize, L. Petzold, and D. Reed. SIAM Publications.

67. Kincaid, D. R.; Oppe, T. C.; Respess, J. R.; Young, D. M. 1984. *ITPACKV 2C User's Guide*, CNA-191. Center for Numerical Analysis, University of Texas at Austin.

68. Kirkpatrick, S.; Stoll, E. P. 1981. "A Very Fast Shift-Register Sequence Random Number Generation." *Journal of Computational Physics*, 40:517–526.

69. Knuth, D. E. 1973. *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, Mass.

70. Knuth, D. E. 1981. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, (second edition), Addison-Wesley, Reading, Mass.

71. Lambiotte, J. J.; Voigt, R. G. December 1975. "The Solution of Tridiagonal Linear Systems on the CDC STAR-100 Computer." *ACM Transactions on Mathematical Software* 1(4):308–329.

72. Lawson, C. L.; Hanson, R. J. 1974. *Solving Least Squares Problems* Prentice-Hall, Englewood Cliffs, New Jersey.

73. Lawson, C. L.; Hanson, R. J.; Kincaid, D. R.; Krough, F. T. Sept. 1979. "Basic Linear Algebra Subprograms

for Fortran Usage." *ACM Transactions on Mathematical Software* 5(3):308–323.

74. Lewis, P. A. W.; Goodman, A. S.; Miller, J. M. 1969. "A Pseudo-Random Number Generator for the System/360." *IBM System Journal*, 8(2).
75. McCracken, D. D.; Dorn, W. S. 1964. *Numerical Methods and Fortran Programming*, John Wiley and Sons, New York.
76. Melhem, R. 1987. "Toward Efficient Implementation of Preconditioned Conjugate Gradient Methods on Vector Supercomputers." *Journal of Supercomputer Applications*, Vol. 1.
77. Moler, C. B.; Stewart, G. W. 1973. "An Algorithm for the Generalized Matrix Eigenvalue Problem." *SIAM Journal of Numerical Analysis*, 10:241–256.
78. Oppenheim, A. V.; Schafer, R. W. 1975. *Digital Signal Processing* Prentice-Hall, Englewood Cliffs, New Jersey.
79. Oppenheim, A. V.; Weinstein, C. August 1972. "Effects of Finite Register Length in Digital Filtering and the Fast Fourier Transform." *IEEE Proceedings*, AU-17:209–215.
80. Saad, Y.; Schultz, M. H. 1986. "GMRES: A Generalized Minimum Residual Algorithm for Solving Nonsymmetric Linear Systems." *SIAM Journal of Scientific and Statistical Computing*, 7:856–869. Philadelphia, Pa.
81. Smith, B. T.; Boyle, J. M.; Dongarra, J. J.; Garbow, B. S.; Ikebe, Y.; Klema, V. C.; Moler, C. B. 1976. "Matrix Eigensystem Routines." *EISPACK Guide Lecture Notes in Computer Science, Vol. 6* Springer-Verlag, New York, Heidelberg, Berlin.
82. Sonneveld; Wesseling; DeZeeuv. 1985. *Multigrid and Conjugate Gradient Methods as Convergence Acceleration Techniques in Multigrid Methods for Integral and Differential Equations*, 117–167. Edited by D.J. Paddon and M. Holstein. Oxford University Press (Claredon), Oxford.
83. Sonneveld, P. January 1989. "CGS, a Fast Lanczos-Type Solver for Nonsymmetric Linear Systems." *SIAM Journal of Scientific and Statistical Computing*, 10(1):36–52.
84. Stewart, G. 1973. *Introduction to Matrix Computations* Academic Press, New York, N. Y.
85. Stewart, G. W. 1976. "The Economical Storage of Plane Rotations." *Numerische Mathematik*, 25(2):137–139.
86. Stroud, A. H.; Secrest, D. 1966. *Gaussian Quadrature Formulas* Prentice-Hall, Englewood Cliffs, New Jersey.
87. Suhl, U. H.; Aittoniemi, L. 1987. "Computing Sparse LU-Factorization for Large-Scale Linear Programming Bases." *Report Number 58* Freie University, Berlin.
88. Tausworthe, R. C. 1965. "Random Numbers Generated by Linear Recurrence Modulo Two." *Mathematical Computing, Vol. 19*
89. Van der Vorst, H. A. 1992. "Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems." *SIAM Journal of Scientific Statistical Computing*, 13:631–644.
90. Weinstein, C. September 1969. "Round-off Noise in Floating Point Fast Fourier Transform Calculation." *IEEE Transactions on Audio Electroacoustics* AU-17:209–215.
91. Wilkinson, J. H. 1965. *The Algebraic Eigenvalue Problem*, Oxford University Press (Claredon), Oxford.
92. Wilkinson, J. H. 1963. *Rounding Errors in Algebraic Processes*, Prentice-Hall, Englewood Cliffs, New Jersey.
93. Wilkinson, J. H.; Reinsch, C. 1971. *Handbook for Automatic Computation, Vol. II, Linear Algebra*, Springer-Verlag, New York, Heidelberg, Berlin.
94. Zierler, N. 1969 "Primitive Trinomials Whose Degree Is a Mersenne Exponent." *Information and Control*, 15:67–69.
95. Zlatev, Z. 1980. "On Some Pivotal Strategies in Gaussian Elimination by Sparse Technique." *SIAM Journal of Numerical Analysis*, 17(1):18–30.

ESSL Publications

This section lists the ESSL publications for each major task that you perform.

You can order the full set of ESSL for AIX publications through the Subscription Library Services System (SLSS) by specifying:

- The subject code 82
- The program number: 5765-C42

You can also order individual publications by specifying the individual order numbers. For example, to order a copy of the *ESSL Version 3 Guide and Reference*, specify the order number SA22-7272. Contact your IBM Marketing Representative or Systems Engineer to order manuals through SLSS.

Evaluation and Planning

ESSL Products General Information, GC23-0529—provides detailed information helpful in evaluating and planning for all the ESSL products: Parallel ESSL, ESSL for AIX, and ESSL/370.

Installation

ESSL Version 3 Installation Memo, GI10-0604—describes how to install ESSL on AIX. It is a packing list for the ESSL product when it is shipped. (One copy is delivered with each ESSL product.)

Application Programming

ESSL Version 3 Guide and Reference, SA22-7272—contains ESSL guidance information for designing, coding, and running programs using ESSL, and contains complete reference information for coding calls to the ESSL subroutines. This manual is available in HTML and PostScript format on the product medium.

ESSL Version 2 Release 2 Guide and Reference, SC23-0526—contains ESSL guidance information for designing, coding, and running programs using ESSL/370, and contains complete reference information for coding calls to the ESSL/370 subroutines.

Parallel ESSL Version 2 Guide and Reference, SA22-7273—contains Parallel ESSL guidance information for designing, coding, and running programs using Parallel ESSL, and contains complete reference information for coding calls to the Parallel ESSL subroutines. This manual is available in HTML and PostScript format on the Parallel ESSL product medium.

Related Publications

The related publications listed below may be useful to you when using ESSL.

AIX Version 4 for the RS/6000

IBM AIX Version 4 for the RS/6000 Calls and Subroutines Reference, SC23-2198

IBM AIX Version 4 Commands Reference, SBOF-1851

IBM AIX Version 4 General Programming Concepts: Writing and Debugging Programs, SC23-2533

IBM AIX Version 4 System Management Guide: Operating System and Devices, SC23-2525

XL Fortran

IBM XL Fortran for AIX User's Guide Version 5, SC09-1354

IBM XL Fortran for AIX Language Reference Version 5, SC09-1353

XL C and C++

IBM C Set ++ for AIX/6000 User's Guide, SC09-1605

IBM C Set ++ for AIX/6000 C Language Reference, SC09-1730

IBM C Set ++ for AIX/6000 C++ Language Reference, SC09-1606

IBM C Set ++ for AIX/6000 Standard Class Library Reference Guide, SC09-1604

PL/I

PL/I Set for AIX Programming Guide, SC26-8456

PL/I Set for AIX Language Reference, SC26-8455

Workstation Processors

RS/6000 POWERstation and POWERserver Hardware Technical Reference—General Information, SA23-2643

IBM 3838 Array Processor

IBM 3838 Array Processor Functional Characteristics GA24-3639

OS/VS1 and OS/VS2 MVS Vector Processing Subsystem Programmer's Guide, GC24-5125

MVS Extended Architecture Vector Processing Subsystem Application Programmer's Guide, SC28-1202

Index

Numerics

3838 Array Processor
  general signal processing routines 740
3838 Array Processor publications BIB-5

A

abbreviations xvi
  for product names xxi
  in the Glossary GLOS-1
  interpreting math and programming xxv
absolute value xvi
  maximum 202
  minimum 205
  notation xxv
  sum of all absolute values 214
accuracy xvi
  considerations for dense and banded linear algebraic equations 463
  considerations for eigensystem analysis 695
  considerations for Fourier transforms, convolutions, and correlations 743
  considerations for interpolation 905
  considerations for linear algebra subprograms 199
  considerations for matrix operations 381
  considerations for numerical quadrature 925
  considerations for related computations 747
  considerations for sorting and searching 885
  error of computation 44
  of results 6, 44
  precisions 44
  what accuracy means 44
  where to find information on 44
acronyms xvi
  associated with programming values xxv
  in the Glossary GLOS-1
  product names xxi
adding xvi
  absolute values 214
  general matrices or their transposes 384
  vector x to vector y and store in vector z 260
address notation xxv
advantages of ESSL 3
AIX xvi
  publications BIB-5
AIX processing procedures
  for processing your Fortran program 164
  processing your C program 165
  processing your C++ program 166
algebra xvi, 459
  See also linear algebra subprograms
algebra *(continued)*
  See also linear algebraic equations
Announcing ESSL brochure BIB-4
ANSI definitions in Glossary GLOS-1
APAR xvi
application programming, publication for BIB-5
applications in the industry 4
architecture supported by ESSL on the workstations 8
arguments xvi
  coding rules 30
  conventions used in the subroutine descriptions xxviii
  diagnosing ESSL input-argument errors 175
  font for ESSL calling xxii
  list of ESSL input-argument errors 179
  passing in C programs 131
  passing in C++ programs 147
array xvi
  coding in C programs 134
  coding in C++ programs 150
  coding in Fortran programs 112
  conventions for xxiv
  definition of 29
  real and complex elements 112
  setting up data structures inside 55
  storage techniques overview 28
array data xvi
  storage and performance tradeoffs 45
arrow notation, what it means xxv
attention error messages, interpreting 177
audience of this book xvii
autocorrelation of one or more sequences 856, 860
auxiliary working storage xvi
  calculating 33
  dynamic allocation 32
  list of subroutines using 31
  provided by the user 33

B

background books BIB-1
band matrix
  definition of 77
  storage layout 78, 80, 83, 84, 88, 89
band matrix subroutines, names of 459
band width 77, 82
banded linear algebraic equation subroutines
  SGBF and DGBF 548
  SGBS and DGBS 552
  SGTF and DGTF 562
  SGTNP, DGTNP, CGTNP, and ZGTNP 567
  SGTNPF, DGTNPF, CGTNPF, and ZGTNPF 570
  SGTNPS, DGTNPS, CGTNPS, and ZGTNPS 573

© Copyright IBM Corp. 1997

banded linear algebraic equation subroutines *(continued)*
SGTS and DGTS 565
SPBF, DPBF, SPBCHF, and DPBCHF 555
SPBS, DPBS, SPBCHS, and DPBCHS 559
SPTF and DPTF 576
SPTS and DPTS 578
STBSV, DTBSV, CTBSV, and ZTBSV 580
base program, processing your
under AIX 163
Basic Linear Algebra Subprograms xvi
See also BLAS
bibliography BIB-1
binary search 896
BLAS (Basic Linear Algebra Subprograms) xvi, 195
ESSL subprograms 195, APA-1
Level 1 APA-1
Level 2 APA-2
Level 3 APA-3
migrating from 171
migrating to ESSL 25
BLAS-general-band storage mode 80
bold letters, usage of xxii
books BIB-1
See also publications

C

C (C programming language) xvi
coding programs 130
ESSL header file 130, 133
function reference 130
handling errors in your program 136
how to code arrays 134
passing character arguments 131
procedures for processing your program under AIX 163
program calling interface 130
setting up complex and logical data 133
C++ (C++ programming language) xvi
coding programs 146
ESSL header file 146, 148
function reference 146
handling errors in your program 152
how to code arrays 150
passing character arguments 147
procedures for processing your program under AIX 163
program calling interface 146
publications BIB-5
setting up complex and logical data 148
calculating auxiliary working storage 33
calculating transform lengths 38
CALL statement 111
See also calling sequence
calling sequence xvi
for C programs 130
for C++ programs 146
for Fortran programs 111
calling sequence *(continued)*
for PL/I programs 162
specifying the arguments 30
subroutines versus functions 111, 130, 146
syntax description xxviii
cataloged procedures, ESSL 163
CAXPY 217
CAXPYI 289
CCOPY 220
CDOTC 223
CDOTCI 292
CDOTU 223
CDOTUI 292
ceiling notation and meaning xxv
CGBMV 341
CGEADD 384
CGEEV 698
CGEF 468
CGEMM 413
CGEMMS 408
CGEMUL 398
CGEMV 297
CGERC 308
CGERU 308
CGES 471
CGESM 475
CGESUB 391
CGETMI 452
CGETMO 455
CGETRF 481
CGETRS 485
CGTHR 283
CGTHRZ 286
CGTNP 567
CGTNPF 570
CGTNPS 573
character data xvi
conventions xxii, 28
characters, special usage of xxv
CHBMV 348
CHEMM 422
CHEMV 316
CHER 324
CHER2 332
CHER2K 444
CHERK 437
choosing the ESSL library 25
choosing the ESSL subroutine 25
CHPEV 710
CHPMV 316
CHPR 324
CHPR2 332
CHPSV 719
citations BIB-1
See also references, math background

CNORM2 240
coding your program xvi
  arguments in ESSL calling sequences 30
  CALL sequence for C programs 130
  CALL sequence for C++ programs 146
  CALL sequence for Fortran programs 111
  CALL sequence for PL/I programs 162
  calls to ESSL in C programs 130
  calls to ESSL in C++ programs 146
  calls to ESSL in Fortran programs 111
  data types used in your program 28
  handling errors with ERRSET, EINFO, ERRSAV, ERRSTR, and return codes 118, 136, 152
  restrictions for application programs 28
  techniques that affect performance 45
column vector 55
comparison of accuracy for libraries 6
compilers, required by ESSL on the workstations 8
compiling your program xvi
  under AIX 163
complex and real array elements 112
complex conjugate notation xxv
complex data xvi
  conventions xxii, 28
  setting up for C 133
  setting up for C++ 148
complex Hermitian band matrix
  definition of 85
  storage layout 85
complex Hermitian matrix
  definition of 70
  storage layout 70
complex Hermitian Toeplitz matrix
  definition of 72
complex matrix 62
complex vector 55
compressed-diagonal storage mode for sparse matrices 94
compressed-matrix storage mode for sparse matrices 93
compressed-vector, definition and storage mode 61
computational areas, overview 4
computational errors xvi
  diagnosing 175
  list of messages for 187
  overview 48
condition number, reciprocal of xvi
  general matrix 490, 516
  positive definite real symmetric matrix 510, 521
conjugate notation xxv
conjugate transpose xvi
  of matrix operation results for multiply 401, 412, 416
conjugate transpose of a matrix 63
conjugate transpose of a vector 57
continuation, convention for numerical data xxii
conventions xvi, xxii
  for messages 178
  mathematical and programming notations xxv
conventions *(continued)*
  subroutine descriptions xxvii
convolution and correlation xvi
  autocorrelation of one or more sequences 856
  direct method xvi
    one sequence with another sequence 837
    with decimated output 852
  mixed radix Fourier method xvi
    autocorrelation of one or more sequences 860
    one sequence with one or more sequences 843
  one sequence with one or more sequences 831
convolution and correlation subroutines
  accuracy considerations 743
  performance and accuracy considerations 745
  performance considerations 743
  SACOR 856
  SACORF 860
  SCON and SCOR 831
  SCOND and SCORD 837
  SCONF and SCORF 843
  SDCON, DDCON, SDCOR, and DDCOR 852
  usage considerations 741
copy a vector 220
correlation 831
  See also convolution and correlation
cosine notation xxv
cosine transform 774
courier font usage xxii
CPOF 494
CPOSM 505
CROT 250
CROTG 243
CSCAL 254
CSCTR 280
CSROT 250
CSSCAL 254
CSWAP 257
CSYAX 272
CSYMM 422
CSYR2K 444
CSYRK 437
CTBMV 359
CTBSV 580
CTPMV 353
CTPSV 528
CTRMM 430
CTRMV 353
CTRSM 534
CTRSV 528
cubic spline interpolating 915, 921
customer service, IBM 173
customer support, IBM 173
CVEA 260
CVEM 268
CVES 264

CWLEV 880
CYAX 272
CZAXPY 275

D

DASUM 214
data xvi
  array data 29
  conventions for scalar data xxii, 28
data structures (vectors and matrices) 55
DAXPY 217
DAXPYI 289
DBSRCH 896
DCFT 749
DCFT2 790
DCFT3 812
DCOPY 220
DCOSF 774
DCRFT 766
DCRFT2 804
DCRFT3 824
DCSIN2 921
DCSINT 915
DDCON 852
DDCOR 852
DDOT 223
DDOTI 292
default values in the ESSL error option table 51
definitions of terms in the Glossary GLOS-1
dense and banded subroutines
  performance and accuracy considerations 463
dense linear algebraic equation subroutines
  SGEF, DGEF, CGEF, and ZGEF 468
  SGEFCD and DGEFCD 490
  SGEICD and DGEICD 516
  SGES, DGES, CGES, and ZGES 471
  SGESM, DGESM, CGESM, and ZGESM 475
  SGETRF, DGETRF, CGETRF and ZGETRF 481
  SGETRS, DGETRS, CGETRS, and ZGETRS 485
  SPOFCD and DPOFCD 510
  SPOSM, DPOSM, CPOSM, and ZPOSM 505
  SPPF, DPPF, SPOF, DPOF, CPOF, and ZPOF 494
  SPPFCD and DPPFCD 510
  SPPICD, DPPICD, SPOICD, and DPOICD 521
  SPPS and DPPS 502
  STPSV, DTPSV, CTPSV, and ZTPSV 528
  STRI, DTRI, STPI, and DPTI 542
  STRSM, DTRSM, CTRSM, and ZTRSM 534
  STRSV, DTRSV, CTRSV, and ZTRSV 528
dense matrix, definition 92
descriptions, conventions used in the subroutine xxvii
designing your program xvi
  accuracy of results 44
  choosing the ESSL library 25
  choosing the ESSL subroutine 25
designing your program *(continued)*
  error considerations 47
  performance considerations 45
  storage considerations 28
determinant xvi
  general matrix 490, 516
  general skyline sparse matrix 597
  matrix notation xxv
  positive definite real symmetric matrix 510, 521
  symmetric skyline sparse matrix 615
DGBF 548
DGBMV 341
DGBS 552
DGEADD 384
DGEEV 698
DGEF 468
DGEFCD 490
DGEGV 727
DGEICD 516
DGELLS 689
DGEMM 413
DGEMMS 408
DGEMTX 297
DGEMUL 398
DGEMV 297
DGEMX 297
DGER 308
DGES 471
DGESM 475
DGESUB 391
DGESVF 676
DGESVS 684
DGETMI 452
DGETMO 455
DGETRF 481
DGETRS 485
DGHMQ 949
DGKFS 597
DGKTRN 990
DGLGQ 941
DGLNQ 932
DGLNQ2 935
DGRAQ 945
DGSF 587
DGSS 593
DGTF 562
DGTHR 283
DGTHRZ 286
DGTNP 567
DGTNPF 570
DGTNPS 573
DGTS 565
diagnosis procedures xvi
  attention error messages 177
  computational errors 175
  ESSL messages, list of 178

diagnosis procedures *(continued)*
in your program 173
informational error messages 177
initial problem diagnosis procedures (symptom index) 174
input-argument errors 175
miscellaneous error messages 178
program exceptions 175
resource error messages 176
diagnostics 178
See also messages
diagonal-out skyline storage mode 100
dimensions of arrays xvi
storage layout 112
direct access storage xvi
direct method xvi
general skyline sparse matrix 597
general sparse matrix 587
symmetric skyline sparse matrix 615
direct sparse matrix solvers
usage considerations 463
DIZC 870
DNAXPY 227
DNDOT 232
DNORM2 240
DNRAND 957
DNRM2 237
documentation BIB-1
See also publications
dot product xvi
notation xxv
of dense vectors 223
of sparse vectors 292
special (compute N times) 232
DPBCHF 555
DPBCHS 559
DPBF 555
DPBS 559
DPINT 907
DPOF 494
DPOFCD 510
DPOICD 521
DPOLY 867
DPOSM 505
DPPF 494
DPPFCD 510
DPPICD 521
DPPS 502
DPTF 576
DPTNQ 929
DPTS 578
DQINT 876
DRCFT 758
DRCFT2 797
DRCFT3 818
DROT 250
DROTG 243
DSBMV 348
DSCAL 254
DSCTR 280
DSDCG 653
DSDGCG 668
DSDMX 373
DSINF 782
DSKFS 615
DSKTRN 996
DSLMX 316
DSLR1 324
DSLR2 332
DSMCG 645
DSMGCG 661
DSMMX 366
DSMTM 369
DSORT 888
DSORTS 893
DSORTX 890
DSPEV 710
DSPMV 316
DSPR 324
DSPR2 332
DSPSV 719
DSRIS 634
DSRSM 986
DSSRCH 900
DSWAP 257
DSYGV 733
DSYMM 422
DSYMV 316
DSYR 324
DSYR2 332
DSYR2K 444
DSYRK 437
DTBMV 359
DTBSV 580
DTPI 542
DTPINT 912
DTPMV 353
DTPSV 528
DTREC 873
DTRI 542
DTRMM 430
DTRMV 353
DTRSM 534
DTRSV 528
DURAND 954
DURXOR 961
DVEA 260
DVEM 268
DVES 264
DWLEV 880

DYAX 272
dynamic allocation of auxiliary working storage 32
See also auxiliary working storage
DZASUM 214
DZAXPY 275
DZNRM2 237

E

efficiency of your program 45
eigensystem analysis subroutines
performance and accuracy considerations 695
SGEEV, DGEEV, CGEEV, and ZGEEV 698
SGEGV and DGEGV 727
SSPEV, DSPEV, CHPEV, and ZHPEV 710
SSPSV, DSPSV, CHPSV, and ZHPSV 719
SSYGV and DSYGV 733
eigenvalues and eigenvectors xvi, 698
See also extreme eigenvalues and eigenvectors
complex Hermitian matrix 710
general matrix 698
real general matrices 727
real symmetric matrix 710, 733
real symmetric positive definite matrix 733
EINFO, ESSL error information-handler xvi
considerations when designing your program 48
diagnosis procedures using 175
subroutine description 968
using EINFO in C programs 136
using EINFO in C++ programs 153
using EINFO in Fortran programs 119
element of a matrix notation xxv
element of a vector notation xxv
Engineering and Scientific Subroutine Library xvi
See also ESSL (Engineering and Scientific Subroutine Library)
error conditions, conventions used in the subroutine descriptions xxix
error messages 178
See also messages
error option table default values 51
error-handling subroutines xvi
EINFO 968
ERRSAV 971
ERRSET 972
ERRSTR 974
errors xvi
attention error messages, interpreting 177
attention messages, overview 51
calculating auxiliary storage 31
computational errors 48, 175
EINFO subroutine description 968
extended error-handling subroutines 23
handling errors in your C program 136
handling errors in your C++ program 152
handling errors in your Fortran program 118
errors *(continued)*
how errors affect output 47
informational error messages, interpreting 177
input-argument errors 175
input-argument errors, overview 47
miscellaneous error messages, interpreting 178
overview of 47
program exceptions 47, 175
resource error messages, interpreting 176
resource errors, overview 50
types of errors that you can encounter 47
using ERRSAV and ERRSTR 53
values returned for EINFO error codes 968
when to use ERRSET 51
where to find information on 47
ERRSAV xvi
in workstation environment 23
subroutine description 971
using with large applications 53
ERRSET xvi
diagnosis procedures using 175
ESSL default values for 51
handling errors in C 136
handling errors in C++ 152
in workstation environment 23
subroutine description 972
using EINFO in C programs 136
using EINFO in C++ programs 153
using EINFO in Fortran programs 119
using ERRSET, EINFO, and return codes in C 136
using ERRSET, EINFO, and return codes in C++ 153
using ERRSET, EINFO, and return codes in Fortran 119
when to use 48, 51
when to use ERRSAV and ERRSTR with ERRSET 53
ERRSTR xvi
in workstation environment 23
subroutine description 974
using with large applications 53
ESSL (Engineering and Scientific Subroutine Library) xvi
advantages of 3
attention error messages, interpreting 177
attention messages, overview 51
coding your program 111
computational areas, overview 4
computational errors 48
computational errors, diagnosing 175
designing your program 25
diagnosis procedures for ESSL errors 173
eigensystem analysis subroutines 695
error option table default values 51
extended error-handling subroutines 23
Fourier transform, convolutions and correlations, and related-computation subroutines 739
functional capability 4
informational error messages, interpreting 177
input-argument errors, diagnosing 175

ESSL (Engineering and Scientific Subroutine Library) *(continued)*
input-argument errors, overview 47
installation requirements 8
interpolation subroutines 905
introduction to 3
languages supported 7
linear algebra subprograms 195
linear algebraic equations subroutines 459
matrix operation subroutines 379
message conventions 178
messages, list of 178
migrating between RS/6000 processors 170
migrating programs 169
migrating to future releases or future hardware 170
miscellaneous error messages, interpreting 178
name xxi
names with an underscore, interpreting xxi
number of subroutines in each area 4
numerical quadrature subroutines 925
ordering publications BIB-4
overview 3
overview of the subroutines 4
packaging characteristics 8
parallel processing subroutines on the workstations 6
processing your program 163
program exceptions 47
program number for BIB-4
publications overview BIB-4
random number generation subroutines 953
reference information conventions xxvii
related publications BIB-5
resource error messages, interpreting 176
resource errors, overview 50
setting up your data structures 55
sorting and searching subroutines 885
usability of subroutines 3
utility subroutines 965
when coding large applications 53
when to use ERRSET for ESSL errors 51
ESSL messages 178
See also messages
ESSL/370, migrating from 171
ESSLPARM file xvi
migrating programs to use 169
Euclidean length xvi
with no scaling of input 240
with scaling of input 237
Euclidean norm notation xxv
evaluation and planning, publications for BIB-4
examples of matrices 62
examples of vectors 55
examples, conventions used in the subroutine descriptions xxix
exception xvi
See also program exception
exponential function notation xxv
expressions, special usage of xxv
extended error-handling subroutines xvi
handling errors in C 136
handling errors in C++ 152
handling errors in your Fortran program 118
how they work 47, 53
in ESSL and in Fortran, list of 23
using them in diagnosing problems 175
extended-error-handling subroutines, using
in your C program 136
in your C++ program 152
in your Fortran program 118
extreme eigenvalues and eigenvectors xvi, 698, 719, 733
See also eigenvalues and eigenvectors
complex Hermitian matrix 719
real symmetric matrix 719

F

factoring xvi
general band matrix 548
general matrix 468, 481, 490
general skyline sparse matrix 597
general sparse matrix 587
general tridiagonal matrix 562
positive definite xvi
complex Hermitian matrix 494
real symmetric matrix 494, 510
symmetric band matrix 555
symmetric tridiagonal matrix 576
symmetric skyline sparse matrix 615
tridiagonal matrix 567, 570
fast Fourier transform (FFT) 745
See also Fourier transform
FFT 745
See also Fourier transform
filter 21
See also quadratic interpolation
See also Wiener-Levinson filter coefficients subroutine
floor notation and meaning xxv
fonts used in this book xxii
formula for transform lengths, interpreting 39
formulas for auxiliary storage, interpreting 33
Fortran xvi
languages required by ESSL on the workstations 8
publications BIB-5
Fortran considerations xvi
coding programs 111
function reference 195
handling errors in your program 119
procedures for processing your program under AIX 163
Fortran function reference 111
Fortran program calling interface 111
Fourier transform
one dimension xvi
complex 749

Fourier transform *(continued)*
one dimension *(continued)*
complex-to-real 766
cosine transform 774
real-to-complex 758
sine transform 782
three dimensions xvi
complex 812
complex-to-real 824
real-to-complex 818
two dimensions xvi
complex 790
complex-to-real 804
real-to-complex 797
Fourier transform subroutines
accuracy considerations 743
how they achieve high performance 745
performance considerations 743
SCFT and DCFT 749
SCFT2 and DCFT2 790
SCFT3 and DCFT3 812
SCOSF and DCOSF 774
SCRFT and DCRFT 766
SCRFT2 and DCRFT2 804
SCRFT3 and DCRFT3 824
SRCFT and DRCFT 758
SRCFT2 and DRCFT2 797
SRCFT3 and DRCFT3 818
SSINF and DSINF 782
terminology used for 741
usage considerations 741
Frobenius norm notation xxv
full-matrix storage mode 92
full-vector, definition and storage mode 61
function xvi
calling sequence in C programs 130
calling sequence in C++ programs 146
calling sequence in Fortran programs 111
function reference 195
functional capability of the ESSL subroutines 4
functional description, conventions used in the subroutine descriptions xxix
functions, ESSL 195
future migration considerations 170

G

gather vector elements 283, 286
Gaussian quadrature methods xvi
Gauss-Hermite Quadrature 949
Gauss-Laguerre Quadrature 941
Gauss-Legendre Quadrature 932
Gauss-Rational Quadrature 945
two-dimensional Gauss-Legendre Quadrature 935
general matrix subroutines, names of 459
general tridiagonal matrix
definition of 90
storage layout 90
general-band storage mode 78
generalized eigensystem xvi
real general matrices 727
real symmetric matrix 733
real symmetric positive definite matrix 733
generation of random numbers 953
Givens plane rotation, constructing 243
Glossary GLOS-1
greek letters notation xxv
guide information 1
guidelines for handling problems 173
See also diagnosis procedures

H

half band width 82
handling errors xvi
in your C program 136
in your C++ program 152
in your Fortran program 118
hardware xvi
publications BIB-5
required on the workstations 8
header file, ESSL, for C 130, 133
header file, ESSL, for C++ 146, 148
Hermitian band matrix
definition of 85
storage layout 85
Hermitian matrix
definition of 70
definition of, complex 72
storage layout 70
how to use this book xix, xx
Hypertext Markup Language, required products 8

I

i-th zero crossing 870
IBM products, migrating from 171
IBM publications BIB-4
See also publications
IBSRCH 896
ICAMAX 202
IDAMAX 202
IDAMIN 205
identifying problems 174
IDMAX 208
IDMIN 211
IESSL 975
industry areas 4
infinity notation xxv
informational error messages, interpreting 177

informational messages, for ESSL 178
input arguments, conventions used in the subroutine descriptions xxviii
input data, conventions for 28
input-argument errors xvi
  diagnosing 175
  list of messages for 179
  overview 47, 50, 51
installation documentation, Program Directory BIB-5
int notation and meaning xxv
integer data xvi
  conventions xxii, 28
integral notation xxv
interchange elements of two vectors 257
interface, ESSL xvi
  for C programs 130
  for C++ programs 146
  for Fortran programs 111
  for PL/I programs 162
interpolating xvi
  cubic spline 915
  local polynomial 912
  polynomial 907
  quadratic 876
  two-dimensional cubic spline 921
interpolation subroutines
  accuracy considerations 905
  performance considerations 905
  SCSIN2 and DCSIN2 921
  SCSINT and DCSINT 915
  SPINT and DPINT 907
  STPINT and DTPINT 912
  usage considerations 905
introduction to ESSL 3
inverse xvi
  general matrix 516
  matrix notation xxv
  positive definite real symmetric matrix 521
  triangular matrix 542
ISAMAX 202
ISAMIN 205
ISMAX 208
ISMIN 211
ISO definitions in Glossary GLOS-1
ISORT 888
ISORTS 893
ISORTX 890
ISSRCH 900
italic font usage xxii
iterative linear system solver xvi
  general sparse matrix 634, 661, 668
  sparse negative definite symmetric matrix 645, 653
  sparse positive definite symmetric matrix 645, 653
  symmetric sparse matrix 634
  usage considerations 465
IZAMAX 202

L

l(2) norm xvi
  with no scaling of input 240
  with scaling of input 237
languages supported by ESSL 7
LAPACK
  ESSL subprograms APB-1
LAPACK, migrating from 171
leading dimension for matrices xvi
  how it is used for matrices 64
  how it is used in three dimensions 107
least squares solution 684
  See also linear least squares solution
letters, fonts of xxii
Level 1 BLAS APA-1
Level 2 BLAS APA-2
Level 3 BLAS APA-3
level of ESSL, getting 975
library
  migrating from a non-IBM 171
  migrating from another IBM 171
  migrating from ESSL Version 2 to Version 3 169
  migrating from LAPACK 171
  overview 4
Licensed Program Specification, ESSL BIB-4
linear algebra 459
  See also linear algebraic equations
linear algebra subprograms xvi, 195
  See also sparse vector-scalar linear algebra subprograms
  See also vector-scalar linear algebra subprograms
  accuracy considerations 199
  list of matrix-vector linear algebra subprograms 197
  list of sparse matrix-vector linear algebra subprograms 198
  list of sparse vector-scalar linear algebra subprograms 197
  list of vector-scalar linear algebra subprograms 195
  overview 195
  performance considerations 199
  usage considerations 199
linear algebraic equations xvi, 459
  See also banded linear algebraic equation subroutines
  See also dense linear algebraic equation subroutines
  See also linear least squares subroutines
  See also sparse linear algebraic equation subroutines
  accuracy considerations 463
  list of banded linear algebraic equation subroutines 460
  list of dense linear algebraic equations 459
  list of linear least squares subroutines 462
  list of sparse linear algebraic equation subroutines 461
  overview 459
  performance considerations 463
  usage considerations 462

linear least squares solution xvi
preparing for 676
QR decomposition with column pivoting 689
singular value decomposition 684
linear least squares subroutines
SGELLS and DGELLS 689
SGESVF and DGESVF 676
SGESVS and DGESVS 684
linking and loading your program xvi
under AIX 163
loading your program xvi
logical data xvi
conventions xxii, 28
setting up for C 133
setting up for C++ 148
long precision xvi
accuracy statement 6
meaning of 44
lower band width 77
lower storage mode 66, 68
lower-band-packed storage mode 84
lower-packed storage mode 66
lower-storage-by-rows for symmetric sparse matrices 99
lower-triangular storage mode 74, 76
lower-triangular-band-packed storage mode 87, 89
lower-triangular-packed storage mode 74, 75

M

mailing list for ESSL customers 9
masking underflow xvi
for performance 45
why you should 45
math and programming notations xxv
math background publications BIB-1
See also references, math background
mathematical expressions, conventions for xxv
mathematical functions, overview 4
matrix xvi, 71, 459
See also general matrix
See also Toeplitz matrix
band matrix 77
complex Hermitian band matrix 85
complex Hermitian matrix 70, 72
complex Hermitian Toeplitz matrix 72
conventions for xxiii
description of 62
font for xxii
full or dense matrix 92
general tridiagonal matrix 90
leading dimension for 64
negative definite complex Hermitian matrix 71
negative definite symmetric matrix 69
positive definite complex Hermitian matrix 71
positive definite symmetric band matrix 85
positive definite symmetric matrix 69
matrix *(continued)*
positive definite symmetric tridiagonal matrix 92
sparse matrix 92
storage of 63
symmetric band matrix 82
symmetric matrix 65
symmetric tridiagonal matrix 91
Toeplitz matrix 71
triangular band matrices 86
triangular matrices 74
matrix operation subroutines
accuracy considerations 381
performance considerations 381
SGEADD, DGEADD, CGEADD, and ZGEADD 384
SGEMM, DGEMM, CGEMM, and ZGEMM 413
SGEMMS, DGEMMS, CGEMMS, and ZGEMMS 408
SGEMUL, DGEMUL, CGEMUL, and ZGEMUL 398
SGESUB, DGESUB, CGESUB, and ZGESUB 391
SGETMI, DGETMI, CGETMI, and ZGETMI 452
SGETMO, DGETMO, CGETMO, and ZGETMO 455
SSYMM, DSYMM, CSYMM, ZSYMM, CHEMM, and ZHEMM 422
SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, and ZHER2K 444
SSYRK, DSYRK, CSYRK, ZSYRK, CHERK, and ZHERK 437
STRMM, DTRMM, CTRMM, and ZTRMM 430
usage considerations 380
matrix-matrix product xvi
complex Hermitian matrix 422
complex symmetric matrix 422
general matrices, their transposes, or their conjugate transposes 413
real symmetric matrix 422
triangular matrix 430
matrix-vector linear algebra subprograms xvi
SGBMV, DGBMV, CGBMV, and ZGBMV 341
SGEMX, DGEMX, SGEMTX, DGEMTX, SGEMV, DGEMV, CGEMV, and ZGEMV 297
SGER, DGER, CGERU, ZGERU, CGERC, and ZGERC 308
SSBMV, DSBMV, CHBMV, and ZHBMV 348
SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV, SSLMX and DSLMX 316
SSPR, DSPR, CHPR, ZHPR, SSYR, DSYR, CHER, ZHER, SSLR1, and DSLR1 324
SSPR2, DSPR2, CHPR2, ZHPR2, SSYR2, DSYR2, CHER2, ZHER2, SSLR2, and DSLR2 332
STBMV, DTBMV, CTBMV, and ZTBMV 359
STPMV, DTPMV, CTPMV, ZTPMV, STRMV, DTRMV, CTRMV, and ZTRMV 353
matrix-vector product xvi
complex Hermitian band matrix 348
complex Hermitian matrix 316
general band matrix, its transpose, or its conjugate transpose 341

matrix-vector product *(continued)*
general matrix, its transpose, or its conjugate transpose 297
real symmetric band matrix 348
real symmetric matrix 316
sparse matrix 366
sparse matrix or its transpose 373
triangular band matrix, its transpose, or its conjugate transpose 359
triangular matrix, its transpose, or its conjugate transpose 353
max notation and meaning xxv
maximum xvi
absolute value 202
value 208
meanings of words in the Glossary GLOS-1
messages xvi
ESSL and attention messages, interpreting 177
ESSL informational messages, interpreting 177
ESSL miscellaneous messages, interpreting 178
ESSL resource messages, interpreting 176
list of ESSL messages 179
message conventions 178
migrating xvi
from ESSL Version 2 to Version 3 169
from ESSL/370 171
from LAPACK 171
from non-IBM libraries 171
from other IBM subroutine libraries 171
future migration considerations 170
programs to ESSL 169
RS/6000 processors considerations 170
min notation and meaning xxv
minimum xvi
absolute value 205
value 211
miscellaneous error messages, interpreting 178
mod notation and meaning xxv
modification level of ESSL, getting 975
Modifying C language AIX commands 165
Modifying C++ language AIX commands 166
Modifying Fortran language AIX commands 164
modulo notation xxv
multiplying xvi
See also product
compute SAXPY or DAXPY N times 227
general matrices using Strassen's algorithm 408
general matrices, their transposes, or their conjugate transposes 398
notation xxv
sparse vector x by a scalar, add sparse vector y, and store in vector y 289
vector x by a scalar and store in vector x 254
vector x by a scalar and store in vector y 272
vector x by a scalar, add to vector y, and store in vector y 217
multiplying *(continued)*
vector x by a scalar, add to vector y, and store in vector z 275
vector x by vector y, and store in vector z 268
multithreaded xvi
definition GLOS-2
ESSL subroutines 25

N

name usage restrictions 28
names in ESSL with an underscore (_) prefix, how to interpret xxi
names of xvi
products and acronyms xxi
the eigensystem analysis subroutines 695
the Fourier transform, convolution and correlation, and related-computation subroutines 739
the interpolation subroutines 905
the linear algebra subprograms 195
the linear algebraic equations subroutines 459
the matrix operations subroutines 379
the numerical quadrature subroutines 925
the random number generation subroutines 953
the sorting and searching subroutines 885
the utility subroutines 965
National Language Support 174
negative definite complex Hermitian matrix
definition of 71
negative definite complex Hermitian Toeplitz matrix
definition of 72
negative definite Hermitian matrix
storage layout 71
negative definite symmetric matrix
definition of 69
storage layout 70
negative definite symmetric Toeplitz matrix
definition of 71
negative stride, for vectors 60
NLS, National Language Support 174
non-IBM library, migrating from 171
norm notation xxv
normally distributed random numbers, generate 957
notations and conventions xxii
notes, conventions used in the subroutine descriptions xxix
number of subroutines in each area 4
numbers 23
See also random number generation
accuracy of computations 44
accuracy of computations, for ESSL 6
numerical quadrature xvi
accuracy considerations 925
performance considerations 925
programming considerations for SUBF 926
usage considerations 925

numerical quadrature performed xvi
on a function xvi
using Gauss-Hermite Quadrature 949
using Gauss-Laguerre Quadrature 941
using Gauss-Legendre Quadrature 932
using Gauss-Rational Quadrature 945
using two-dimensional Gauss-Legendre Quadrature 935
on a set of points 929
numerical quadrature subroutines
SGHMQ and DGHMQ 949
SGLGQ and DGLGQ 941
SGLNQ and DGLNQ 932
SGLNQ2 and DGLNQ2 935
SGRAQ and DGRAQ 945
SPTNQ and DPTNQ 929

O

objectives for this manual xvii
one norm notation xxv
online documentation
online Guide and Reference manual BIB-5
required Hypertext Markup Language products 8
option table, default values for ESSL errors 51
order numbers of the publications BIB-4
ordering IBM publications BIB-4
output xvi
accuracy on different processors 6
how errors affect output 47
output arguments, conventions used in the subroutine descriptions xxviii
overflow, avoiding 237
overview
of eigensystem analysis 695
of ESSL 3
of Fourier transforms, convolutions and correlations, and related computations 739
of interpolation 905
of linear algebra subprograms 195
of linear algebraic equations 459
of matrix operations 379
of numerical quadrature 925
of random number generation 953
of sorting and searching 885
of the documentation BIB-4
of utility subroutines 965

P

packed band storage mode 78
packed-Hermitian-Toeplitz storage mode 73
packed-symmetric-Toeplitz storage mode 72
Parallel Fortran xvi
parallel processing xvi
introduction to 6
performance xvi
achieving better performance in your program 45
aspects of parallel processing on the workstations 6
coding techniques that affect performance 45
considerations for dense and banded linear algebraic equations 463
considerations for eigensystem analysis 695
considerations for Fourier transforms, convolutions, and correlations 743
considerations for interpolation 905
considerations for linear algebra subprograms 199
considerations for matrix operations 381
considerations for numerical quadrature 925
considerations for related computations 747
considerations for sorting and searching 885
how the Fourier transforms achieve high performance 745
information on ESSL run-time performance 46
tradeoffs for convolution and correlation subroutines 745
where to find information on 46
pi notation xxv
PL/I (Programming Language/I) xvi
coding programs 162
handling errors in your program 136, 153
publications BIB-5
plane rotation xvi
applying a 250
constructing a Givens 243
planning your program 25
planning, publications for BIB-4
polynomial
evaluating 867
interpolating 907, 912
positive definite complex Hermitian matrix
definition of 71
positive definite complex Hermitian Toeplitz matrix
definition of 72
positive definite Hermitian matrix
storage layout 71
positive definite symmetric band matrix
definition of 85
storage layout 85
positive definite symmetric band matrix subroutines, names of 459
positive definite symmetric matrix
definition of 69
storage layout 70
positive definite symmetric matrix subroutines, names of 459
positive definite symmetric Toeplitz matrix
definition of 71
positive definite symmetric tridiagonal matrix 92
definition of 92
storage layout 92

positive stride, for vectors 58
precision, meaning of 44
precision, short and long 6
problems, handling 173
  See also diagnosis procedures
problems, IBM support for 173
procedures, job processing xvi
  setting up your own AIX 163
processing your program xvi
  requirements for ESSL on the workstations 8
  setting up your AIX procedures 163
  steps involved in 163
  using parallel subroutines on the workstations 6
processor-independent formulas for auxiliary storage, interpreting 33
product xvi, 217
  See also multiplying
  matrix-matrix xvi
    complex Hermitian matrix 422
    complex symmetric matrix 422
    general matrices, their transposes, or their conjugate transposes 413
    real symmetric matrix 422
    triangular matrix 430
  matrix-vector xvi
    complex Hermitian band matrix 348
    complex Hermitian matrix 316
    general band matrix, its transpose, or its conjugate transpose 341
    general matrix, its transpose, or its conjugate transpose 297
    real symmetric band matrix 348
    real symmetric matrix 316
    sparse matrix 366
    sparse matrix or its transpose 373
    triangular band matrix, its transpose, or its conjugate transpose 359
    triangular matrix, its transpose, or its conjugate transpose 353
product names, acronyms for xxi
products, programming xvi
  migrating from LAPACK 171
  migrating from other IBM 171
  required by ESSL on the workstations, programming 8
profile-in skyline storage mode 103
program xvi
  attention messages, overview 51
  coding 111
  computational errors 48
  design 25
  errors 47
  handling errors in your C program 136
  handling errors in your C++ program 152
  handling errors in your Fortran program 118
  input-argument errors, overview 47
  interface for C programs 130
program *(continued)*
  interface for C++ programs 146
  interface for Fortran programs 111
  interface for PL/I programs 162
  migrated to ESSL 169
  performance, achieving high 45
  processing your program 163
  resource errors, overview 50
  setting up your data structures 55
  types of data in your program 28
  when coding large applications 53
program exceptions xvi
  description of ESSL 47
program exceptions, diagnosing 175
program number for ESSL BIB-4
programming considerations for SUBF in numerical quadrature 926
programming items, font for xxii
Programming Language/I xvi
  See also PL/I (Programming Language/I)
programming products xvi
  required by ESSL on the workstations 8
programming publications BIB-5
PTF xvi
  getting the most recent level applied 975
publications xvi
  list of ESSL BIB-4
  math background BIB-1
  related BIB-5

Q

QR decomposition with column pivoting 689
quadratic interpolation 876

R

random number generation xvi
  long period uniformly distributed 961
  normally distributed 957
  uniformly distributed 954
  usage considerations 953
random number generation subroutines
  SNRAND and DNRAND 957
  SURAND and DURAND 954
  SURXOR and DURXOR 961
rank-2k update xvi
  complex Hermitian matrix 444
  complex symmetric matrix 444
  real symmetric matrix 444
rank-k update xvi
  complex Hermitian matrix 437
  complex symmetric matrix 437
  real symmetric matrix 437
rank-one update xvi
  complex Hermitian matrix 324

rank-one update *(continued)*
general matrix 308
real symmetric matrix 324
rank-two update xvi
complex Hermitian matrix 332
real symmetric matrix 332
readers of this book xvii
real and complex array elements 112
real data xvi
conventions xxii, 28
real general matrix eigensystem analysis subroutine 695
real symmetric matrix eigensystem analysis subroutine 695
reciprocal of the condition number xvi
general matrix 490, 516
positive definite real symmetric matrix 510, 521
recursive filter 21
See also time-varying recursive filter
reference for ESSL, online BIB-5
reference information xvi
ESSL online information BIB-5
math background texts and reports BIB-1
organization of 193
what is in each subroutine description and the conventions used xxvii
references, math background xvi
related publications BIB-5
related-computation subroutines
accuracy considerations 747
CWLEV and ZWLEV 880
performance considerations 747
SIZC and DIZC 870
SPOLY and DPOLY 867
SQINT and DQINT 876
STREC and DTREC 873
SWLEV and DWLEV 880
release of ESSL, getting 975
reporting problems to IBM 173
required publications BIB-4
requirements xvi
auxiliary working storage 32, 33
for ESSL workstation product 8
software products on the workstations 8
transforms in storage, lengths of 38
workstation hardware 8
resource error messages, interpreting 176
restrictions, ESSL coding 28
results xvi
accuracy on different processors 6
how accuracy is affected by the nature of the computation 44
in C programs 130
in C++ programs 146
in Fortran programs 111
multiplication of NaN 44
results transposed and conjugate transposed for matrix multiplication 401, 412, 416
results transposed for matrix addition 386
results transposed for matrix subtraction 393
return code xvi
in C programs 136
in C++ programs 153
in Fortran programs 119
using during diagnosis 175
rotation xvi
applying a plane 250
constructing a Givens plane 243
routine names 28
row vector 55
run-time performance xvi
optimizing in your program 45
run-time problems, diagnosing xvi
attention error messages, interpreting 177
computational errors 175
informational error messages, interpreting 177
input-argument errors 175
miscellaneous error messages, interpreting 178
resource error messages, interpreting 176
running your program xvi

S

SACOR 856
SACORF 860
SASUM 214
SAXPY 217
SAXPYI 289
SBSRCH 896
scalar data xvi
conventions xxii, 28
scalar items, font for xxii
Scalar Library xvi
scalar processing xvi
scale argument used for Fourier transform subroutines 744
scaling, when to use 45
SCASUM 214
scatter vector elements 280
SCFT 749
SCFT2 790
SCFT3 812
Scientific Subroutine Package xvi
See also SSP (Scientific Subroutine Package)
SCNRM2 237
SCON 831
SCOND 837
SCONF 843
SCOPY 220
SCOR 831
SCORD 837
SCORF 843
SCOSF 774
SCOSFT, no documentation provided for 739

SCRFT 766
SCRFT2 804
SCRFT3 824
SCSIN2 921
SCSINT 915
SDCON 852
SDCOR 852
SDOT 223
SDOTI 292
searching xvi
  binary 896
  sequential 900
selecting an ESSL library 25
selecting an ESSL subroutine 25
sequences xvi
  conventions for xxiii
  description of 105
  storage layout 105
sequential search 900
service, IBM 173
setting up
  AIX procedures 163
setting up your data 28
SGBF 548
SGBMV 341
SGBS 552
SGEADD 384
SGEEV 698
SGEF 468
SGEFCD 490
SGEGV 727
SGEICD 516
SGELLS 689
SGEMM 413
SGEMMS 408
SGEMTX 297
SGEMUL 398
SGEMV 297
SGEMX 297
SGER 308
SGES 471
SGESM 475
SGESUB 391
SGESVF 676
SGESVS 684
SGETMI 452
SGETMO 455
SGETRF 481
SGETRS 485
SGHMQ 949
SGLGQ 941
SGLNQ 932
SGLNQ2 935
SGRAQ 945
SGTF 562
SGTHR 283
SGTHRZ 286
SGTNP 567
SGTNPF 570
SGTNPS 573
SGTS 565
short precision xvi
  accuracy statement 6
  meaning of 44
SIGN notation and meaning xxv
signal processing subroutines 740
simple formulas for auxiliary storage, interpreting 33
sin notation xxv
sine transform 782
singular value decomposition for a general matrix 676, 684
SIZC 870
size of array xvi
  required for a vector 57
skyline solvers
  usage considerations 464
skyline storage mode for sparse matrices, diagonal-out 100
skyline storage mode for sparse matrices, profile-in 103
SL MATH (Subroutine Library—Mathematics) xvi
  migrating from 171
SLSS (Subscription Library Services System) BIB-4
SMP xvi
  definition GLOS-3
  ESSL Library, why use it 25
  ESSL multithreaded subroutines 25
  performance 6
SNAXPY 227
SNDOT 232
SNORM2 240
SNRAND 957
SNRM2 237
software products
  required by ESSL on the workstations 8
  required by Hypertext Markup Language 8
solving xvi
  general band matrix 552
  general matrix or its transpose 471, 485
  general skyline sparse matrix 597
  general sparse matrix or its transpose 593
  general tridiagonal matrix 565, 567, 573
  iterative linear system solver xvi
    general sparse matrix 634, 661, 668
    sparse negative definite symmetric matrix 645, 653
    sparse positive definite symmetric matrix 645, 653
    symmetric sparse matrix 634
  multiple right-hand sides xvi
    general matrix, its transpose, or its conjugate transpose 475, 485
    positive definite complex Hermitian matrix 505
    positive definite real symmetric matrix 505
    triangular matrix 534
  positive definite xvi
    real symmetric matrix 502

solving *(continued)*
positive definite *(continued)*
symmetric band matrix 559
symmetric tridiagonal matrix 578
symmetric skyline sparse matrix 615
triangular band matrix 580
triangular matrix 528
some eigenvalues and eigenvectors xvi, 719
See also extreme eigenvalues and eigenvectors
sorting xvi
elements of a sequence 888
index 890
stable sort 893
sorting and searching subroutines
accuracy considerations 885
IBSRCH, SBSRCH, and DBSRCH 896
ISORT, SSORT, and DSORT 888
ISORTS, SSORTS, and DSORTS 893
ISORTX, SSORTX, and DSORTX 890
ISSRCH, SSSRCH, and DSSRCH 900
performance considerations 885
usage considerations 885
sparse linear algebraic equation subroutines
DGKFS 597
DGSF 587
DGSS 593
DSDCG 653
DSDGCG 668
DSKFS 615
DSMCG 645
DSMGCG 661
DSRIS 634
sparse matrix subroutines
direct solvers 463
iterative linear system solvers 465
performance and accuracy considerations 463, 465
skyline solvers 464
sparse matrix-vector linear algebra subprograms xvi
DSDMX 373
DSMMX 366
DSMTM 369
sparse matrix, definition and storage modes 92
sparse vector-scalar linear algebra subprograms xvi
SAXPYI, DAXPYI, CAXPYI, and ZAXPYI 289
SDOTI, DDOTI, CDOTUI, ZDOTUI, CDOTCI, and, ZDOTCI 292
SGTHR, DGTHR, CGTHR, and ZGTHR 283
SGTHRZ, DGTHRZ, CGTHRZ, and ZGTHRZ 286
SSCTR, DSCTR, CSCTR, and ZSCTR 280
sparse vector, definition and storage modes 61
SPBCHF 555
SPBCHS 559
SPBF 555
SPBS 559
special usage xvi
of matrix addition 386
special usage *(continued)*
of matrix multiplication 401, 412, 416
of matrix subtraction 393
spectral norm notation xxv
SPINT 907
SPOF 494
SPOFCD 510
SPOICD 521
SPOLY 867
SPOSM 505
SPPF 494
SPPFCD 510
SPPICD 521
SPPS 502
SPTF 576
SPTNQ 929
SPTS 578
SQINT 876
square root notation xxv
SRCFT 758
SRCFT2 797
SRCFT3 818
SROT 250
SROTG 243
SSBMV 348
SSCAL 254
SSCTR 280
SSINF 782
SSLMX 316
SSLR1 324
SSLR2 332
SSORT 888
SSORTS 893
SSORTX 890
SSP (Scientific Subroutine Package) xvi
migrating from 171
SSPEV 710
SSPMV 316
SSPR 324
SSPR2 332
SSPSV 719
SSSRCH 900
SSWAP 257
SSYGV 733
SSYMM 422
SSYMV 316
SSYR 324
SSYR2 332
SSYR2K 444
SSYRK 437
stable sort 893
STBMV 359
STBSV 580
stepping through storage, for matrices 63
stepping through storage, for vectors 58

storage xvi
array storage techniques overview 28
auxiliary working storage requirements 32, 33
compressed-diagonal storage mode for sparse matrices 94
compressed-matrix storage mode for sparse matrices 93
considerations when designing your program 28
diagonal-out skyline storage mode for sparse matrices 100
for matrices 63
for vectors 57
layout for a complex Hermitian band matrix 85
layout for a complex Hermitian matrix 70
layout for a general tridiagonal matrix 90
layout for a negative definite Hermitian matrix 71
layout for a negative definite symmetric matrix 70
layout for a positive definite Hermitian matrix 71
layout for a positive definite symmetric matrix 70
layout for a positive definite symmetric tridiagonal matrix 92
layout for a sequence 105, 106
layout for a symmetric tridiagonal matrix 91
layout for a Toeplitz matrix 72, 73
layout for band matrices 78, 80
layout for positive definite symmetric band matrices 85
layout for sparse matrices 92
layout for sparse vectors 61
layout for symmetric band matrices 83
layout for symmetric matrices 66
layout for triangular band matrices 87, 88, 89
layout for triangular matrices 74
list of subroutines using auxiliary storage 31
list of subroutines using transforms 38
of arrays in Fortran 112
profile-in skyline storage mode for sparse matrices 103
storage-by-columns for sparse matrices 98
storage-by-indices for sparse matrices 97
storage-by-rows for sparse matrices 99
tradeoffs for input 45
transform length requirements 38
storage conversion subroutine xvi
general skyline sparse matrix 990
sparse matrix 986
symmetric skyline sparse matrix 996
storage-by-columns for sparse matrices 98
storage-by-indices for sparse matrices 97
storage-by-rows for sparse matrices 99
STPI 542
STPINT 912
STPMV 353
STPSV 528
Strassen's algorithm, multiplying general matrices 408
STREC 873
STRI 542
stride xvi
defining vectors in arrays 58
stride *(continued)*
how it is used in three dimensions 107
negative 60
optimizing for your Fourier transforms 744
positive 58
subroutine for optimizing Fourier transforms 977
zero 59
STRIDE 977
STRMM 430
STRMV 353
STRSM 534
STRSV 528
structures, data (vectors and matrices) 55
subject code for ESSL documentation BIB-4
subprogram xvi
See also subroutine
linear algebra 195
meaning of xxi, 195
subprogram, definition xxi
subroutine xvi
calling sequence format for C programs 130
calling sequence format for C++ programs 146
calling sequence format for Fortran programs 111
choose of 25
conventions used in the description of xxvii
overview of ESSL 4
Subroutine Library—Mathematics xvi
See also SL MATH (Subroutine Library—Mathematics)
subroutine, definition xxi
subroutines, ESSL
CAXPY 217
CAXPYI 289
CCOPY 220
CDOTC 223
CDOTCI 292
CDOTU 223
CDOTUI 292
CGBMV 341
CGEADD 384
CGEEV 698
CGEF 468
CGEMM 413
CGEMMS 408
CGEMUL 398
CGEMV 297
CGERC 308
CGERU 308
CGES 471
CGESM 475
CGESUB 391
CGETMI 452
CGETMO 455
CGETRF 481
CGETRS 485
CGTHR 283
CGTHRZ 286

subroutines, ESSL *(continued)*
CGTNP 567
CGTNPF 570
CGTNPS 573
CHBMV 348
CHEMM 422
CHEMV 316
CHER 324
CHER2 332
CHER2K 444
CHERK 437
CHPEV 710
CHPMV 316
CHPR 324
CHPR2 332
CHPSV 719
CNORM2 240
CPOF 494
CPOSM 505
CROT 250
CROTG 243
CSCAL 254
CSCTR 280
CSROT 250
CSSCAL 254
CSWAP 257
CSYAX 272
CSYMM 422
CSYR2K 444
CSYRK 437
CTBMV 359
CTBSV 580
CTPMV 353
CTPSV 528
CTRMM 430
CTRMV 353
CTRSM 534
CTRSV 528
CVEA 260
CVEM 268
CVES 264
CWLEV 880
CYAX 272
CZAXPY 275
DASUM 214
DAXPY 217
DAXPYI 289
DBSRCH 896
DCFT 749
DCFT2 790
DCFT3 812
DCOPY 220
DCOSF 774
DCRFT 766
DCRFT2 804
DCRFT3 824

subroutines, ESSL *(continued)*
DCSIN2 921
DCSINT 915
DDCON 852
DDCOR 852
DDOT 223
DDOTI 292
DGBF 548
DGBMV 341
DGBS 552
DGEADD 384
DGEEV 698
DGEF 468
DGEFCD 490
DGEGV 727
DGEICD 516
DGELLS 689
DGEMM 413
DGEMMS 408
DGEMTX 297
DGEMUL 398
DGEMV 297
DGEMX 297
DGER 308
DGES 471
DGESM 475
DGESUB 391
DGESVF 676
DGESVS 684
DGETMI 452
DGETMO 455
DGETRF 481
DGETRS 485
DGHMQ 949
DGKFS 597
DGKTRN 990
DGLGQ 941
DGLNQ 932
DGLNQ2 935
DGRAQ 945
DGSF 587
DGSS 593
DGTF 562
DGTHR 283
DGTHRZ 286
DGTNP 567
DGTNPF 570
DGTNPS 573
DGTS 565
DIZC 870
DNAXPY 227
DNDOT 232
DNORM2 240
DNRAND 957
DNRM2 237
DPBCHF 555

subroutines, ESSL *(continued)*
DPBCHS 559
DPBF 555
DPBS 559
DPINT 907
DPOF 494
DPOFCD 510
DPOICD 521
DPOLY 867
DPOSM 505
DPPF 494
DPPFCD 510
DPPICD 521
DPPS 502
DPTF 576
DPTNQ 929
DPTS 578
DQINT 876
DRCFT 758
DRCFT2 797
DRCFT3 818
DROT 250
DROTG 243
DSBMV 348
DSCAL 254
DSCTR 280
DSDCG 653
DSDGCG 668
DSDMX 373
DSINF 782
DSKFS 615
DSKTRN 996
DSLMX 316
DSLR1 324
DSLR2 332
DSMCG 645
DSMGCG 661
DSMMX 366
DSMTM 369
DSORT 888
DSORTS 893
DSORTX 890
DSPEV 710
DSPMV 316
DSPR 324
DSPR2 332
DSPSV 719
DSRIS 634
DSRSM 986
DSSRCH 900
DSWAP 257
DSYGV 733
DSYMM 422
DSYMV 316
DSYR 324
DSYR2 332

subroutines, ESSL *(continued)*
DSYR2K 444
DSYRK 437
DTBMV 359
DTBSV 580
DTPI 542
DTPINT 912
DTPMV 353
DTPSV 528
DTREC 873
DTRI 542
DTRMM 430
DTRMV 353
DTRSM 534
DTRSV 528
DURAND 954
DURXOR 961
DVEA 260
DVEM 268
DVES 264
DWLEV 880
DYAX 272
DZASUM 214
DZAXPY 275
DZNRM2 237
EINFO 968
ERRSAV 971
ERRSET 972
ERRSTR 974
IBSRCH 896
ICAMAX 202
IDAMAX 202
IDAMIN 205
IDMAX 208
IDMIN 211
IESSL 975
ISAMAX 202
ISAMIN 205
ISMAX 208
ISMIN 211
ISORT 888
ISORTS 893
ISORTX 890
ISSRCH 900
IZAMAX 202
SACOR 856
SACORF 860
SASUM 214
SAXPY 217
SAXPYI 289
SBSRCH 896
SCASUM 214
SCFT 749
SCFT2 790
SCFT3 812
SCNRM2 237

subroutines, ESSL *(continued)*
SCON 831
SCOND 837
SCONF 843
SCOPY 220
SCOR 831
SCORD 837
SCORF 843
SCOSF 774
SCRFT 766
SCRFT2 804
SCRFT3 824
SCSIN2 921
SCSINT 915
SDCON 852
SDCOR 852
SDOT 223
SDOTI 292
SGBF 548
SGBMV 341
SGBS 552
SGEADD 384
SGEEV 698
SGEF 468
SGEFCD 490
SGEGV 727
SGEICD 516
SGELLS 689
SGEMM 413
SGEMMS 408
SGEMTX 297
SGEMUL 398
SGEMV 297
SGEMX 297
SGER 308
SGES 471
SGESM 475
SGESUB 391
SGESVF 676
SGESVS 684
SGETMI 452
SGETMO 455
SGETRF 481
SGETRS 485
SGHMQ 949
SGLGQ 941
SGLNQ 932
SGLNQ2 935
SGRAQ 945
SGTF 562
SGTHR 283
SGTHRZ 286
SGTNP 567
SGTNPF 570
SGTNPS 573
SGTS 565

subroutines, ESSL *(continued)*
SIZC 870
SNAXPY 227
SNDOT 232
SNORM2 240
SNRAND 957
SNRM2 237
SPBCHF 555
SPBCHS 559
SPBF 555
SPBS 559
SPINT 907
SPOF 494
SPOFCD 510
SPOICD 521
SPOLY 867
SPOSM 505
SPPF 494
SPPFCD 510
SPPICD 521
SPPS 502
SPTF 576
SPTNQ 929
SPTS 578
SQINT 876
SRCFT 758
SRCFT2 797
SRCFT3 818
SROT 250
SROTG 243
SSBMV 348
SSCAL 254
SSCTR 280
SSINF 782
SSLMX 316
SSLR1 324
SSLR2 332
SSORT 888
SSORTS 893
SSORTX 890
SSPEV 710
SSPMV 316
SSPR 324
SSPR2 332
SSPSV 719
SSSRCH 900
SSWAP 257
SSYGV 733
SSYMM 422
SSYMV 316
SSYR 324
SSYR2 332
SSYR2K 444
SSYRK 437
STBMV 359
STBSV 580

subroutines, ESSL *(continued)*
STPI 542
STPINT 912
STPMV 353
STPSV 528
STREC 873
STRI 542
STRIDE 977
STRMM 430
STRMV 353
STRSM 534
STRSV 528
SURAND 954
SURXOR 961
SVEA 260
SVEM 268
SVES 264
SWLEV 880
SYAX 272
SZAXPY 275
ZAXPY 217
ZAXPYI 289
ZCOPY 220
ZDOTC 223
ZDOTCI 292
ZDOTU 223
ZDOTUI 292
ZDROT 250
ZDSCAL 254
ZDYAX 272
ZGBMV 341
ZGEADD 384
ZGEEV 698
ZGEF 468
ZGEMM 413
ZGEMMS 408
ZGEMUL 398
ZGEMV 297
ZGERC 308
ZGERU 308
ZGES 471
ZGESM 475
ZGESUB 391
ZGETMI 452
ZGETMO 455
ZGETRF 481
ZGETRS 485
ZGTHR 283
ZGTHRZ 286
ZGTNP 567
ZGTNPF 570
ZGTNPS 573
ZHBMV 348
ZHEMM 422
ZHEMV 316
ZHER 324
subroutines, ESSL *(continued)*
ZHER2 332
ZHER2K 444
ZHERK 437
ZHPEV 710
ZHPMV 316
ZHPR 324
ZHPR2 332
ZHPSV 719
ZNORM2 240
ZPOF 494
ZPOSM 505
ZROT 250
ZROTG 243
ZSCAL 254
ZSCTR 280
ZSWAP 257
ZSYMM 422
ZSYR2K 444
ZSYRK 437
ZTBMV 359
ZTBSV 580
ZTPMV 353
ZTPSV 528
ZTRMM 430
ZTRMV 353
ZTRSM 534
ZTRSV 528
ZVEA 260
ZVEM 268
ZVES 264
ZWLEV 880
ZYAX 272
ZZAXPY 275
subscript notation, what it means xxv
subtracting xvi
general matrices or their transposes 391
vector y from vector x and store in vector z 264
sum of xvi
See also adding
absolute values 214
summation notation xxv
superscript notation, what it means xxv
support, IBM 173
SURAND 954
SURXOR 961
SVEA 260
SVEM 268
SVES 264
swap elements of two vectors 257
SWLEV 880
SYAX 272
symbols, special usage of xxv
symmetric band matrix
definition of 82
storage layout 83

symmetric matrix
definition of 65
storage layout 66
Symmetric Multi-Processing xvi
See also SMP
symmetric tridiagonal matrix 91
definition of 91
storage layout 91
symmetric-tridiagonal storage mode 91
symptoms, identifying problem 174
syntax rules for call statements and data 30
syntax, conventions used in the subroutine descriptions xxviii
SZAXPY 275

T

table, default values for ESSL error option 51
termination, program xvi
attention messages 51
computational errors 48
input-argument errors 47
resource errors 50
terminology in the Glossary GLOS-1
terminology used for Fourier transforms, convolutions, and correlations 741
terminology, names of products xxi
textbooks cited BIB-1
See also references, math background
thread-safe xvi
definition GLOS-3
ESSL Library, why use it 25
three-dimensional data structures, how stride is used for 107
time-varying recursive filter 873
times notation, multiply xxv
timings, achieving high performance in your program 45
Toeplitz matrix
definition of 71, 72
storage layout 72, 73
traceback map, using during diagnosis 175
transform lengths, calculating 38
transpose xvi
conjugate, of a matrix 63
conjugate, of a vector 57
notation xxv
of a matrix 62, 63
of a matrix inverse notation xxv
of a vector 56, 57
of a vector or matrix notation xxv
of matrix operation results for add 386
of matrix operation results for multiply 401, 412, 416
of matrix operation results for subtract 393
transposing xvi
general matrix (In-Place) 452
general matrix (Out-of-Place) 455
transposing *(continued)*
sparse matrix 369
triangular band matrices
storage layout 87
triangular band matrices, upper and lower
definition of 86
triangular matrices
storage layout 74
triangular matrices, upper and lower
definition of 74
tridiagonal matrix
definition of 90
storage layout 90
tridiagonal storage mode 90
truncation
how truncation affects output 44
type font usage xxii

U

underflow xvi
avoiding underflow 237
why mask it 45
uniformly distributed random numbers, generate 954, 961
upper band width 77
upper storage mode 66, 69
upper-band-packed storage mode 83
upper-packed storage mode 66, 67
upper-storage-by-rows for symmetric sparse matrices 99
upper-triangular storage mode 74, 76
upper-triangular-band-packed storage mode 87, 88
upper-triangular-packed storage mode 74, 75
usability of subroutines 3
usability of the ESSL subroutines 4
usage considerations
direct sparse matrix solvers 463
for Fourier transforms, convolutions, and correlations 741
for interpolation 905
for linear algebra subprograms 199
for linear algebraic equations 462
for matrix operations 380
for numerical quadrature 925
for random number generation 953
for sorting and searching 885
for utility subroutines 965
sparse matrix subroutines (iterative linear system solvers) 465
sparse matrix subroutines (skyline solvers) 464
usage, special xvi
conventions used in the subroutine description xxix
for matrix addition 386
for matrix multiplication 401, 412, 416
for matrix subtraction 393
user applications 4

users of ESSL xvii
using this book xix, xx
utility subroutines xvi
  DGKTRN 990
  DSKTRN 996
  DSRSM 986
  EINFO 968
  ERRSAV 971
  ERRSET 972
  ERRSTR 974
  IESSL 975
  STRIDE 977
  usage considerations 965

V

vector xvi
  compressed vector 61
  conventions for xxiii
  description of 55
  font for xxii
  full vector 61
  number of array elements needed for 57
  sparse vector 61
  storage of 57
  stride for 58
Vector Library xvi
vector processing xvi
Vector Processing Subsystem xvi
  See also VPSS (Vector Processing Subsystem)
vector register xvi
vector-scalar linear algebra subprograms xvi
  ISAMAX, ICAMAX, IDAMAX, and IZAMAX 202
  ISAMIN and IDAMIN 205
  ISMAX and IDMAX 208
  ISMIN and IDMIN 211
  SASUM, DASUM, SCASUM, and DZASUM 214
  SAXPY, DAXPY, CAXPY, and ZAXPY 217
  SCOPY, DCOPY, CCOPY, and ZCOPY 220
  SDOT, DDOT, CDOTU, ZDOTU, CDOTC, and ZDOTC 223
  SNAXPY and DNAXPY 227
  SNDOT and DNDOT 232
  SNORM2, DNORM2, CNORM2, and ZNORM2 240
  SNRM2, DNRM2, SCNRM2, and DZNRM2 237
  SROT, DROT, CROT, ZROT, CSROT, and ZDROT 250
  SROTG, DROTG, CROTG, and ZROTG 243
  SSCAL, DSCAL, CSCAL, ZSCAL, CSSCAL, and ZDSCAL 254
  SSWAP, DSWAP, CSWAP, and ZSWAP 257
  SVEA, DVEA, CVEA, and ZVEA 260
  SVEM, DVEM, CVEM, and ZVEM 268
  SVES, DVES, CVES, and ZVES 264
  SYAX, DYAX, CYAX, ZYAX, CSYAX, and ZDYAX 272
vector-scalar linear algebra subprograms *(continued)*
  SZAXPY, DZAXPY, CZAXPY, and ZZAXPY 275
version of ESSL, getting 975
versions of subroutines 4
VPSS (Vector Processing Subsystem) xvi

W

Wiener-Levinson filter coefficients 880
words in the Glossary GLOS-1
working auxiliary storage, list of subroutines using 31
working storage for band matrix 78
workstations xvi
  migrating between RS/6000 processors 170
  required for ESSL 8

Z

ZAXPY 217
ZAXPYI 289
ZCOPY 220
ZDOTC 223
ZDOTCI 292
ZDOTU 223
ZDOTUI 292
ZDROT 250
ZDSCAL 254
ZDYAX 272
zero crossing 740
  See also i-th zero crossing
zero stride, for vectors 59
ZGBMV 341
ZGEADD 384
ZGEEV 698
ZGEF 468
ZGEMM 413
ZGEMMS 408
ZGEMUL 398
ZGEMV 297
ZGERC 308
ZGERU 308
ZGES 471
ZGESM 475
ZGESUB 391
ZGETMI 452
ZGETMO 455
ZGETRF 481
ZGETRS 485
ZGTHR 283
ZGTHRZ 286
ZGTNP 567
ZGTNPF 570
ZGTNPS 573
ZHBMV 348
ZHEMM 422

ZHEMV 316
ZHER 324
ZHER2 332
ZHER2K 444
ZHERK 437
ZHPEV 710
ZHPMV 316
ZHPR 324
ZHPR2 332
ZHPSV 719
ZNORM2 240
ZPOF 494
ZPOSM 505
ZROT 250
ZROTG 243
ZSCAL 254
ZSCTR 280
ZSWAP 257
ZSYMM 422
ZSYR2K 444
ZSYRK 437
ZTBMV 359
ZTBSV 580
ZTPMV 353
ZTPSV 528
ZTRMM 430
ZTRMV 353
ZTRSM 534
ZTRSV 528
ZVEA 260
ZVEM 268
ZVES 264
ZWLEV 880
ZYAX 272
ZZAXPY 275