IBM Engineering and Scientific Subroutine Library for AIX, Version 5 Release 3, and
IBM Engineering and Scientific Subroutine Library for Linux on POWER, Version 5 Release 5
Version 5 Release 5

ESSL Guide and Reference

IBM
Version 5 Release 5

ESSL Guide and Reference
Note

Before using this information and the product it supports, read the information in "Notices" on page 1359.

This edition applies to:

- Version 5 Release 3 of the IBM Engineering and Scientific Subroutine Library (ESSL) for AIX licensed program, program number 5765-H25
- Version 5 Release 5 of the IBM Engineering and Scientific Subroutine Library (ESSL) for Linux on POWER licensed program, program number 5765-L51

and to all subsequent releases and modifications until otherwise indicated by new edition.

In this document ESSL refers to both of the above products (unless a differentiation between ESSL for AIX and ESSL for Linux is explicitly specified).

Significant changes or additions to the text and illustrations are marked by a vertical line (|) to the left of the change.

IBM welcomes your comments. see the topic “How to Send Your Comments” on page xxix. 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 IBM Corporation 1986, 2016.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
Contents

Tables ............................................. xiii

About this information ........................... xvii
  How to Use This Information .................... xvii
  Where to Find Related Publications ............ xviii
  Using Bibliography References ................ xix
  IBM Request for Enhancement (RFE) Community xix
  How to Find a Subroutine Description .......... xix
  How to Interpret the Subroutine Names with a 
      Prefix Underscore .......................... xix
  Special Terms .................................. xix
      Short and Long Precision.................... xix
      Subroutines and Subprograms ................. xx
  Abbreviated Names ............................. xx
  Conventions and terminology used ............... xxi
      Fonts ...................................... xxi
      Special Notations and Conventions .......... xxi
      Special Characters, Symbols, Expressions, and 
          Abbreviations ........................... xxiv
  How to Interpret the Subroutine Descriptions .... xxi
      Description ................................ xxvi
      Syntax .................................... xxvi
      On Entry .................................. xxviii
      On Return ................................ xxviii
      Notes ..................................... xxviii
      Function .................................. xxviii
      Special Usage ............................... xxviii
      Error Conditions ............................ xxix
      Examples .................................. xxix
  How to Send Your Comments ..................... xxix

Summary of Changes ............................. xxxi
  Future Migration ............................... xxxvii

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
  Accuracy of the Computations ................ 6
  High Performance of ESSL ..................... 6
  The Fortran Language Interface to the Subroutines 8
  Software and Hardware Products That Can Be Used 
      with ESSL ................................ 8
      Hardware Products Supported by ESSL ......... 8
      Operating Systems Supported by ESSL ......... 8
      Software Products Required by ESSL .......... 9
      Software Products for Installing and Customizing 
          ESSL ................................ 10
      Software Products for Displaying ESSL 
          Documentation ........................... 10

List of ESSL Subroutines ........................ 11
  Linear Algebra Subprograms ..................... 11
  Matrix Operations .............................. 16
  Linear Algebraic Equations ..................... 17
  Eigensystem Analysis ........................... 25
  Fourier Transforms, Convolutions and 
      Correlations, and Related Computations .. 26
  Sorting and Searching .......................... 28
  Interpolation .................................. 28
  Numerical Quadrature ........................... 29
  Random Number Generation ........................ 29
  Utilities ..................................... 30

Chapter 2. Planning Your Program ................ 31
  Selecting an ESSL Subroutine ................. 31
      What ESSL Library Do You Want to Use? ..... 31
      Use of SIMD Algorithms by Some Subroutines in 
          the Libraries Provided by ESSL .......... 32
      Multithreaded Subroutines Provided by ESSL .. 38
  Using the ESSL SMP CUDA Library ............... 43
  NVIDIA GPU Power Capping ...................... 47
  What Type of Data Are You Processing in Your 
      Program? ................................... 47
  How Is Your Data Structured? And What Storage 
      Technique Are You Using? .................... 48
  What about Performance and Accuracy? .......... 48
  Avoiding Conflicts with Internal ESSL Routine 
      Names that are Exported ........................ 48
  Setting Up Your Data ............................ 48
      How Do You Set Up Your Scalar Data? ........ 48
      How Do You Set Up Your Arrays? ............... 48
      How Should Your Array Data Be Aligned? ....... 49
      What Storage Mode Should You Use for Your 
          Data? ................................... 49
      How Do You Convert from One Storage Mode to 
          Another? .................................. 49
  Setting Up Your ESSL Calling Sequences ......... 50
      What Is an Input-Output Argument? ............. 50
      What Are the General Rules to Follow when 
          Specifying Data for the Arguments? ......... 50
      What Happens When a Value of 0 Is Specified 
          for N? ................................... 51
      How Do You Specify the Beginning of the Data 
          Structure in the ESSL Calling Sequence? .. 51
  Using Auxiliary Storage in ESSL ................ 51
      Dynamic Allocation of Auxiliary Storage ........ 52
      Setting Up Auxiliary Storage When Dynamic 
          Allocation Is Not Used ..................... 53
  Providing a Correct Transform Length to ESSL .. 58
      Who Do You Want to Calculate the Transform 
          Length? You or ESSL? ..................... 58
      How Do You Calculate the Transform Length 
          Using the Table or Formula? ............... 59
      How Do You Get ESSL to Calculate the 
          Transform Length Using ESSL Error Handling? 59
Getting the Best Accuracy .......................... 63
What Precisions Do ESSL Subroutines Operate
On? ............................................. 63
How does the Nature of the ESSL Computation
Affect Accuracy? ................................ 64
What Data Type Standards Are Used by ESSL,
and What Exceptions Should You Know About? 64
How is Underflow Handled? ....................... 65
Where Can You Find More Information on
Accuracy? ...................................... 65
What about Bitwise-Identical Results? .......... 65
Getting the Best Performance ...................... 65
What General Coding Techniques Can You Use
to Improve Performance? ......................... 65
Where Can You Find More Information on
Performance? ................................... 67
Dealing with Errors when Using ESSL .......... 67
What Can You Do about Program Exceptions? 67
What Can You Do about ESSL Input-Argument
Errors? .......................................... 67
What Can You Do about ESSL Computational
Errors? ........................................... 68
What Can You Do about ESSL Resource Errors? 68
What Can You Do about ESSL Attention
Messages? ....................................... 70
How Do You Control Error Handling by Setting
Values in the ESSL Error Option Table? .... 71
How does Error Handling Work in a Threaded
Environment? .................................... 73
Where Can You Find More Information on
Errors? ........................................... 74

Chapter 3. Setting Up Your Data

Structures ........................................ 75
Concepts ........................................ 75
Vectors .......................................... 75
  Transpose of a Vector .......................... 76
  Conjugate Transpose of a Vector ............. 76
  Vector Storage Representation ............... 77
  How Stride Is Used for Vectors .............. 78
Sparse Vector .................................. 80
Matrices ........................................ 81
  Transpose of a Matrix ......................... 82
  Conjugate Transpose of a Matrix .......... 82
  Matrix Storage Representation .............. 82
  How Leading Dimension Is Used for Matrices. 83
Symmetric Matrix .............................. 85
  Positive Definite or Negative Definite Symmetric
  Matrix ....................................... 89
  Indefinite Symmetric Matrix ................. 89
  Complex Hermitian Matrix .................... 90
  Positive Definite or Negative Definite Complex
  Hermitian Matrix ............................ 91
  Indefinite Complex Hermitian Matrix ....... 91
  Positive Definite or Negative Definite Symmetric
  Toeplitz Matrix .............................. 91
  Positive Definite or Negative Definite Complex
  Hermitian Toeplitz Matrix .................. 92
  Triangular Matrix ....................... 93
  Trapezoidal Matrix ....................... 96
  General Band Matrix ....................... 100
  Symmetric Band Matrix ...................... 105
  Positive Definite Symmetric Band Matrix ... 107
  Complex Hermitian Band Matrix ............ 108
  Positive Definite Complex Hermitian Band
  Matrix ...................................... 108
  Triangular Band Matrix ..................... 109
  General Tridiagonal Matrix ................. 112
  Symmetric Tridiagonal Matrix .............. 114
  Positive Definite Symmetric Tridiagonal Matrix 115
  Complex Hermitian Tridiagonal Matrix .... 116
  Positive Definite Complex Hermitian Tridiagonal
  Matrix ..................................... 116
  Sparse Matrix ................................ 116

Sequences ...................................... 128
Real and Complex Elements in Storage ........ 128
One-Dimensional Sequences .................... 128
Two-Dimensional Sequences ..................... 128
Three-Dimensional Sequences .................... 129
How Stride Is Used for Three-Dimensional
Sequences ..................................... 131

Chapter 4. Coding Your Program ............ 133

Fortran Programs .................................. 133
  Calling ESSL Subroutines and Functions in
  Fortran ..................................... 133
  Setting Up a User-Supplied Subroutine for ESSL
  in Fortran .................................. 133
  Setting Up Scalar Data in Fortran ........... 134
  Setting Up Arrays in Fortran ................. 134
  Creating Multiple Threads and Calling ESSL
  from Your Fortran Program ................. 139
  Handling Errors in Your Fortran Program .... 140
  Example of Handling Errors in a Multithreaded
  Application Program ....................... 149
C Programs ...................................... 151
  Calling ESSL Subroutines and Functions in C .... 151
  Passing Arguments in C ....................... 152
  Setting Up a User-Supplied Subroutine for ESSL
  in C ....................................... 153
  Setting Up Scalar Data in C .................. 153
  Setting Up Complex Data Types in C ........ 154
  Using Logical Data in C ..................... 155
  Setting Up Arrays in C ....................... 155
  Creating Multiple Threads and Calling ESSL
  from Your C Program ...................... 156
  Handling Errors in Your C Program .......... 158
C++ Programs ................................... 166
  Calling ESSL Subroutines and Functions in C++ . 166
  Passing Arguments in C++ .................... 167
  Setting Up a User-Supplied Subroutine for ESSL
  in C++ .................................... 168
  Setting Up Scalar Data in C++ ............... 169
  Using Complex Data in C++ .................. 170
  Using Logical Data in C++ ................... 172
  Setting Up Arrays in C++ ..................... 172
  Creating Multiple Threads and Calling ESSL
  from Your C++ Program .................... 173
  Handling Errors in Your C++ Program ....... 175
Chapter 5. Processing Your Program 185
Processing Your Program on AIX 185
Fortran Program Procedures on AIX 185
C Program Procedures on AIX 187
C++ Program Procedures on AIX 188
Processing Your Program on Linux (little endian mode) 191
Fortran Program Procedures on Linux (little endian mode) 192
C Program Procedures on Linux (little endian mode) 193
C++ Program Procedures on Linux (little endian mode) 196

Chapter 6. Migrating Your Programs 201
Migrating Programs from ESSL for Linux on Power Version 5 Release 4 to Version 5 Release 5 201
Migrating Programs from ESSL for Linux on Power Version 5 Release 3.2 to Version 5 Release 4 201
Migrating Programs from ESSL for Linux on Power Version 5 Release 3.1 to Version 5 Release 3.2 201
Migrating Programs from ESSL for Linux on Power Version 5 Release 2 or ESSL Version 5 Release 3 to Version 5 Release 3.1 201
Migrating Programs from ESSL for Linux on Power Version 5 Release 2 to Version 5 Release 3 202
Migrating Programs from ESSL for AIX 5.1 and ESSL for Linux on Power Version 5 Release 1.1 to Version 5 Release 2 202
Migrating Programs from ESSL for Linux on Power Version 5 Release 1 to Version 5 Release 1.1 203
Migrating Programs from ESSL Version 4 Release 4 to Version 5 Release 1 203
Migrating Programs from ESSL Version 4 Release 3 to Version 4 Release 4 203
Migrating Programs from ESSL Version 4 Release 2.2 or Later to ESSL Version 4 Release 3 204
Migrating Programs from ESSL Version 4 Release 2.1 to Version 4 Release 2.2 204
Migrating Programs from ESSL Version 4 Release 2 to Version 4 Release 2.1 204
Migrating Programs from ESSL Version 4 Release 1 to Version 4 Release 2 204
Planning for Future Migration 205
Migrating From One Hardware Platform to Another 205
  Auxiliary Storage 205
  Bitwise-Identical Results 205
Migrating from Other Libraries to ESSL 206
  Migrating from ESSL/370 206
  Migrating from Another IBM Subroutine Library 206
  Migrating from LAPACK 206
  Migrating from FFTW Version 3.1.2 206
  Migrating from a Non-IBM Subroutine Library 206

Chapter 7. Handling Problems 207
Where to Find More Information About Errors 207
Getting Help from IBM Support 207
National Language Support 208
Dealing with Errors 209
Program Exceptions 209
ESSL Input-Argument Error Messages 209
ESSL Computational Error Messages 210
ESSL Resource Error Messages 210
ESSL Informational and Attention Messages 211
Miscellaneous Error Messages 211
Messages 211
  Attention Messages(2200-2299) 222
  Informational and Attention Error Messages(2600-2699) 222
  Miscellaneous Error Messages(2700-2799) 222

Part 2. Reference Information 223

Chapter 8. Linear Algebra Subprograms 225
Overview of the Linear Algebra Subprograms 225
  Vector-Scalar Linear Algebra Subprograms 225
  Sparse Vector-Scalar Linear Algebra Subprograms 227
  Matrix-Vector Linear Algebra Subprograms 228
  Sparse Matrix-Vector Linear Algebra Subprograms 229
  Use Considerations 230
  Performance and Accuracy Considerations 230
  Vector-Scalar Subprograms 232
  ISAMAX, IDAMAX, ICAMAX, and IZAMAX (Position of the First or Last Occurrence of the Vector Element Having the Largest Magnitude) 233
  ISAMIN and IDAMIN (Position of the First or Last Occurrence of the Vector Element Having Minimum Absolute Value) 236
  ISMAX and IDMAX (Position of the First or Last Occurrence of the Vector Element Having Maximum Value) 239
  ISMIN and IDMIN (Position of the First or Last Occurrence of the Vector Element Having Minimum Value) 242
  SASUM, DASUM, SCASUM, and DZASUM (Sum of the Magnitudes of the Elements in a Vector) 245
  SAXPY, DAXPY, CAXPY, and ZAXPY (Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in the Vector Y) 248
  SCOPY, DCOPY, CCOPY, and ZCOPY (Copy a Vector) 251
  SDOT, DDOT, CDOTU, ZDOTU, CDOTC, and ZDTC (Dot Product of Two Vectors) 254
  SNAXPY and DNAXPY (Compute SAXPY or DAXPY N Times) 258
  SDDOT and DDNODT (Compute Special Dot Products N Times) 263
  SNRM2, DNRM2, SCNRM2, and DZNRM2 (Euclidean Length of a Vector with Scaling of Input to Avoid Destructive Underflow and Overflow) 268

Contents V
SNORM2, DNorm2, CNORM2, and ZNORM2 (Euclidean Length of a Vector with No Scaling of
Input) .......................................................... 271
SROTG, DROTG, CROTG, and ZROTG (Construct
a Givens Plane Rotation) .................................. 274
SROT, DROT, CROT, ZROT, CSROT, and ZDROT
(Apply a Plane Rotation) ................................. 280
SROTMG and DROTMG (Construct a modified
Givens Transformation) ...................................... 284
SROT and DROT (Apply a modified Givens
Transformation) .............................................. 287
SSCAL, DSCAL, CSCAL, ZSCAL, CSSCAL, and
ZDSCAL (Multiply a Vector X by a Scalar
and Store in the Vector X) ......................... 290
SSWAP, DSWAP, CSWAP, and ZSWAP (Interchange
the Elements of Two Vectors) .................. 293
SVEA, DVEA, CVEA, and ZVEA (Add a Vector X
to a Vector Y and Store in a Vector Z) ........... 296
SVES, DVES, CVES, and ZVES (Subtract a Vector Y
from a Vector X and Store in a Vector Z) ........... 300
SVEM, DVEM, CVEM, and ZVEM (Multiply a
Vector X by a Vector Y and Store in a Vector Z) .... 304
SYAX, DYAX, CYAX, ZYAX, CSYAX, and ZDYAX
(Multiply a Vector X by a Scalar and Store in a
Vector Y) ....................................................... 308
SZAXPY, DZAXPY, CZAXPY, and ZZAXPY
(Multiply a Vector X by a Scalar, Add to a Vector Y,
and Store in a Vector Z) .................................. 312
Sparse Vector-Scalar Subprograms .................. 316
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) .............................................. 317
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) ............. 320
SGTHRZ, DGTHRZ, CGTHRZ, and ZGTHRZ
(Gather Specified Elements of a Sparse Vector Y in
Full-Vector Storage Mode into a Sparse Vector X in
Compressed-Vector Storage Mode) ............. 323
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) .............................................................. 326
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) ............. 329
Matrix-Vector Subprograms ......................... 333
SGEMV, DGEMV, CGEMV, ZGEMV, SGEMX,
DGEMX, SGEMTX, and DGEMTX (Matrix-Vector
Product for a General Matrix, Its Transpose, or Its
Conjugate Transpose) ................................... 334
SGER, DGER, CGERU, ZGERU, CGERC, and
ZGERC (Rank-One Update of a General Matrix) .... 345
SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV,
DSYMV, CHEMV, ZHEMV, SSLMX, and DSLMX
(Matrix-Vector Product for a Real Symmetric or
Complex Hermitian Matrix) ......................... 353
SSPR, DSPR, CHPR, ZHPR, SSSYR, DSYR, CHER,
ZHER, SSLR1, and DSLR1 (Rank-One Update of a
Real Symmetric or Complex Hermitian Matrix) .... 362
SSPR2, DSPR2, CHPR2, ZHPR2, SSSYR2, DSYR2,
CHER2, ZHER2, SSLR2, and DSLR2 (Rank-Two
Update of a Real Symmetric or Complex Hermitian
Matrix) .......................................................... 370
SGBMV, DGBMV, CGBMV, and ZGBMV
(Matrix-Vector Product for a General Band Matrix,
Its Transpose, or Its Conjugate Transpose) ........ 379
SSBMV, DSBMV, CHBMV, and ZHBMV
(Matrix-Vector Product for a Real Symmetric or
Complex Hermitian Band Matrix) ................. 386
STRMV, DTRMV, CRTMV, ZTRMV, STPMV,
DTPMV, CTPMV, and ZTPMV (Matrix-Vector
Product for a Triangular Matrix, Its Transpose, or
Its Conjugate Transpose) ................................. 391
STRSV, DTRSV, CRTSV, ZTRSV, SPTSV, DTPSV,
CTPSV, and ZTPSV (Solution of a Triangular
System of Equations with a Single Right-Hand
Side) .......................................................... 398
STBMV, DTBMV, CTBMV, and ZTBMV
(Matrix-Vector Product for a Triangular Band
Matrix, Its Transpose, or Its Conjugate Transpose) . 405
STBSV, DTBSV, CTBSV, and ZTBSV (Triangular
Band Equation Solve) ....................................... 411
Sparse Matrix-Vector Subprograms .............. 417
DSMMX (Matrix-Vector Product for a Sparse
Matrix in Compressed-Matrix Storage Mode) .... 418
DSMTM (Transpose a Sparse Matrix in
Compressed-Matrix Storage Mode) ............... 421
DSDMX (Matrix-Vector Product for a Sparse Matrix
or Its Transpose in Compressed-Diagonal Storage
Mode) ......................................................... 425

Chapter 9. Matrix Operations ............... 429
Overview of the Matrix Operation Subroutines .... 429
Use Considerations ............................................ 430
Specifying Normal, Transposed, or Conjugate
Transposed Input Matrices ............................... 430
Transposing or Conjugate Transposing .......... 431
Performance and Accuracy Considerations ......... 431
In General ...................................................... 431
For Large Matrices ........................................... 431
For Combined Operations ................................. 432
Matrix Operation Subroutines ..................... 433
SGEADD, DGEADD, CGEADD, and ZGEADD
(Matrix Addition for General Matrices or Their
Transposes) ............................................... 434
SGESUB, DGESUB, CGESUB, and ZGESUB (Matrix
Subtraction for General Matrices or Their
Transposes) ............................................... 440
SGEMUL, DGEMUL, CGEMUL, and ZGEMUL
(Matrix Multiplication for General Matrices, Their
Transposes, or Conjugate Transposes) ............... 446
SGEMMS, DGEMMS, CGEMMS, and ZGEMMS
(Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes Using Winograd's Variation of Strassen's Algorithm) .................. 455
SGEMM, DGEMM, CGEMM, and ZGEMM
(Combined Matrix Multiplication and Addition for General Matrices, Their Transposes, or Conjugate Transposes) .................. 461
SSYMM, DSYMM, CSYMM, ZSYMM, CHEMM, and ZHEMM (Matrix-Matrix Product Where One Matrix is Real or Complex Symmetric or Complex Hermitian) .................. 470
STRMM, DTRMM, CTRMM, and ZTRMM
(Triangular Matrix-Matrix Product) ................................ 478
STRSM, DTRSM, CTRSM, and ZTRSM (Solution of Triangular Systems of Equations with Multiple Right-Hand Sides) .................. 486
SSYRK, DSYRK, CSYRK, ZSYRK, CHERK, and ZHERK (Rank-K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix) .................. 494
SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, and ZHER2K (Rank-2K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix) .................. 501
SGETMI, DGETMI, CGETMI, ZGETMI, CGERCI, and ZGERCI (General Matrix Transpose or Conjugate Transpose [In-Place]) .................. 509
SGETMO, DGETMO, CGETMO, ZGETMO, CGECMO, and ZGECMO (General Matrix Transpose or Conjugate Transpose [Out-Of-Place]) .................. 512

Chapter 10. Linear Algebraic Equations ................................ 517
Overview of the Linear Algebraic Equation
Subroutines
Dense Linear Algebraic Equation Subroutines 517
Banded Linear Algebraic Equation Subroutines 520
Sparse Linear Algebraic Equation Subroutines 522
Linear Least Squares Subroutines 523
Dense and Banded Linear Algebraic Equation Considerations .................. 524
Use Considerations .................. 524
Performance and Accuracy Considerations .................. 524
Sparse Matrix Direct Solver Considerations .................. 524
Use Considerations .................. 524
Performance and Accuracy Considerations .................. 525
Sparse Matrix Skyline Solver Considerations .................. 525
Use Considerations .................. 525
Performance and Accuracy Considerations .................. 526
Sparse Matrix Iterative Solver Considerations .................. 526
Use Considerations .................. 526
Performance and Accuracy Considerations .................. 527
Linear Least Squares Considerations .................. 527
Use Considerations .................. 527
Performance and Accuracy Considerations .................. 527
Dense Linear Algebraic Equation Subroutines 529
SGETRF, DGETRF, CGETRF and ZGETRF (General Matrix Factorization and Multiple Right-Hand Side Solve) .................. 530
SGETRS, DGETRS, CGETRS, and ZGETRS (General Matrix Multiple Right-Hand Side Solve) .................. 539
SGEF, DGEF, CGEF, and ZGEF (General Matrix Factorization) .................. 544
SGES, DGES, CGES, and ZGES (General Matrix, Its Transpose, or Its Conjugate Transpose Solve) .................. 547
SGESM, DGESM, CGESM, and ZGESM (General Matrix, Its Transpose, or Its Conjugate Transpose Multiple Right-Hand Side Solve) .................. 551
SPECON, DPECON, CPECON, and ZPECON
(Estimate the Reciprocal of the Condition Number of a General Matrix) .................. 556
SGEFCD and DGEFCD (General Matrix Factorization, Condition Number Reciprocal, and Determinant) .................. 561
SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD (General Matrix Inverse, Condition Number Reciprocal, and Determinant) .................. 572
SLANGE, DLANGE, CLANGE, and ZLANGE
(General Matrix Norm) .................. 575
SPOSV, DPOSV, CPOSV, and ZPOSV (Positive Definite Real Symmetric and Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve) .................. 581
SPOF, DPOF, CPOF, ZPOF, SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPPE, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization) .................. 587
SSPS, DPPS, CPPS, and ZPPS (Positive Definite Real Symmetric and Complex Hermitian Matrix Multiple Right-Hand Side Solve) .................. 598
SPPS and DPPS (Positive Definite Real Symmetric Matrix Solve) .................. 608
SPOCON, DPOCON, CPOCON, ZPOCON, SPPCON, DPPCON, CPPCON, and ZPPCON
(Estimate the Reciprocal of the Condition Number of a Positive Definite Real Symmetric or Complex Hermitian Matrix) .................. 611
SPPFCD, DPPFCD, SPOFCD, and DPOFCD
(Positive Definite Real Symmetric Matrix Factorization, Condition Number Reciprocal, and Determinant) .................. 620
SPOF, DPOF, CPOF, ZPOF, SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPPE, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Inverse, Condition Number Reciprocal, and Determinant) .................. 626
SLANSY, DLANSY, CLANHE, ZLANHE, SLANSP, DLANS, CLANHP, and ZLANHP (Real Symmetric or Complex Hermitian Matrix Norm) .................. 637
SGSV, DGSV, CGSV, and ZGSV (General Band Matrix Factorization and Multiple Right-Hand Side Solve) ........................................ 698

SGBTF, DGBTF, CGBTF, and ZGBTF (General Band Matrix Factorization) .......................................................... 702

SGBTRS, DGBTRS, CGBTRS, and ZGBTRS (General Band Matrix Multiple Right-Hand Side Solve) .............. 706

SGBS and DGBS (General Band Matrix Solve) ........................................ 712

SPBSV, DPBSV, CPBSV, and ZPBSV (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization and Multiple Right-Hand Side Solve) .................................................. 715

SPBTRF, DPBTRF, CBPTRF, and ZPBTRF (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization) .......................................................... 721

SPBTRS, DPBTRS, CPBTRS, and ZPBTRS (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Multiple Right-Hand Side Solve) ........................................ 726

SGTSV, DGTSV, CGTSV, and ZGTSV (General Tridiagonal Matrix Factorization and Multiple Right-Hand Side Solve) .................................................................................. 731

SGTTRF, DGTTRF, CGTTRF, and ZGTTRF (General Tridiagonal Matrix Factorization) ........................................ 735

SGTTRS, DGTTRS, CGTTRS, and ZGTTRS (General Tridiagonal Matrix Multiple Right-Hand Side Solve) ........ 740

SPTSV, DPTSV, CPTSV, and ZPTSV (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization and Multiple Right-Hand Side Solve) .................. 746

SPTTRE, DPTTRE, CPTTRE, and ZPTTRE (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization) .......................................................... 751

SPTTRS, DPTTRS, CPTTRS, and ZPTTRS (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Multiple Right-Hand Solve) ........................................ 755

SGBF and DGBF (General Band Matrix Factorization) .............. 761

SGBS and DGBS (General Band Matrix Solve) ........................................ 765

SPBF, DPBF, SPBCHF, and DPBCHF (Positive Definite Symmetric Band Matrix Factorization) .......... 768

SPBS, DPBS, SPBCHS, and DPBCHS (Positive Definite Symmetric Band Matrix Solve) ...................................... 772

SGTF and DGTF (General Tridiagonal Matrix Factorization) .................................................................................. 775

SGTS and DGTS (General Tridiagonal Matrix Solve) ........... 778

SGTNP, DGTNP, CGTNP, and ZGTNP (General Tridiagonal Matrix Combined Factorization and Solve with No Pivoting) .............. 780

SGTNPF, DGTNPF, CGTNPF, and ZGTNPF (General Tridiagonal Matrix Factorization with No Pivoting) .................................................. 783

SGTNPS, DGTNPS, CGTNPS, and ZGTNPS (General Tridiagonal Matrix Solve with No Pivoting) .............. 786

SGTF and DPTF (Positive Definite Symmetric Tridiagonal Matrix Factorization) ...................................... 789

SPS and DPTS (Positive Definite Symmetric Tridiagonal Matrix Solve) .................................................. 791

**Sparse Linear Algebraic Equation Subroutines** ........................................ 793

DGGF (General Sparse Matrix Factorization Using Storage by Indices, Rows, or Columns) ............................ 794

DGSS (General Sparse Matrix or Its Transpose Solve Using Storage by Indices, Rows, or Columns) ............... 800

DGKFS (General Sparse Matrix or Its Transpose Factorization, Determinant, and Solve Using Skyline Storage Mode) .................................................. 804

DSKFS (Symmetric Sparse Matrix Factorization, Determinant, and Solve Using Skyline Storage Mode) .... 821

DSRIS (Iterative Linear System Solver for a General or Symmetric Sparse Matrix Stored by Rows) ............... 839

DSMCG (Sparse Positive Definite or Negative Definite Symmetric Matrix Iterative Solve Using Compressed-Matrix Storage Mode) .................................................. 850

DSDCG (Sparse Positive Definite or Negative Definite Symmetric Matrix Iterative Solve Using Compressed-Diagonal Storage Mode) .................................................. 858

DSMCCG (General Sparse Matrix Iterative Solve Using Compressed-Matrix Storage Mode) .................. 866

DSDCCG (General Sparse Matrix Iterative Solve Using Compressed-Diagonal Storage Mode) .................. 873

**Linear Least Squares Subroutines** ........................................ 880

SGESV, DGESV, CGESV, ZGESV, SGEESV, DGEESV, CGEESV, and ZGEESV (General Matrix QR Factorization) .................................................. 881

SGEQRF, DGEQRF, CGEQRF, and ZGEQRF (General Matrix QR Factorization) .................................................. 895

SGELS, DGELS, CGELS, and ZGELS (Linear Least Squares Solution for a General Matrix) .................................................. 901

SGELSD, DGELSD, CGELSD, and ZGELSD (Linear Least Squares Solution for a General Matrix Using the Singular Value Decomposition) .................................................. 911
SGESVF and DGESVF (Singular Value Decomposition for a General Matrix) 919
SGESVS and DGESVS (Linear Least Squares Solution for a General Matrix Using the Singular Value Decomposition) 927
SGELLS and DGELLS (Linear Least Squares Solution for a General Matrix with Column Pivoting) 932

Chapter 11. Eigensystem Analysis 939
Overview of the Eigensystem Analysis Subroutines 939
Performance and Accuracy Considerations 940
Eigensystem Analysis Subroutines 941
SGEEV, DGEEV, CGEEV, ZGEEV, SGEEVX, DGEEVX, CGEEVX, and ZGEEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix) 942
SSYEV, DSYEV, CHEEV, ZHEEV, SSPEVX, DSDERV, CHPEVX, ZHPEVX, SSYEVX, DSYEVX, CHEEVX, and ZHEEVX (Eigenvalues and, Optionally, the Eigenvectors of a Real Symmetric or Complex Hermitian Matrix Using a Divide-and-Conquer Algorithm) 978
SGGEV, DGGEV, CGGEV, ZGGEV, SGGEVX, DGGEVX, CGGEVX, and ZGGEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix) 991
SSPGVX, DSPGVX, CHPGVX, ZHPGVX, SSYGVX, DSYGVX, CHEGCVX, and ZHEGCVX (Eigenvalues and, Optionally, the Eigenvectors of a Positive Definite Real Symmetric or Complex Hermitian Generalized Eigenproblem) 1008

Chapter 12. Fourier Transforms, Convolution and Correlation, and Related Computations 1025
Overview of the Signal Processing Subroutines 1025
Fourier Transforms Subroutines 1025
Convolution and Correlation Subroutines 1026
Related-Computation Subroutines 1026
Fourier Transforms, Convolutions, and Correlations Considerations 1027
Use Considerations 1027
Initializing Auxiliary Working Storage 1030
Determining the Amount of Auxiliary Working Storage That You Need 1030
Performance and Accuracy Considerations 1030
When Running on the Workstation Processors 1031
Defining Arrays 1031
Fourier Transform Considerations 1031

How the Fourier Transform Subroutines
Achieve High Performance 1032
Convolution and Correlation Considerations 1032
Related Computation Considerations 1034
Accuracy Considerations 1034

Fourier Transform Subroutines 1035
SCFTD and DCFTD (Multidimensional Complex Fourier Transform) 1036
SRCFTD and DRCFTD (Multidimensional Real-to-Complex Fourier Transform) 1044
SCFTD and DRCFTD (Multidimensional Complex-to-Real Fourier Transform) 1052
SCFT and DCFT (Complex Fourier Transform) 1060
SRCFT and DRCFT (Real-to-Complex Fourier Transform) 1069
SCFT and DRCFT (Complex-to-Real Fourier Transform) 1077
SCOSF and DCOSF (Cosine Transform) 1085
SSINF and DSINF (Sine Transform) 1093
SCFT2 and DCFT2 (Complex Fourier Transform in Two Dimensions) 1101
SRCFT2 and DRCFT2 (Real-to-Complex Fourier Transform in Two Dimensions) 1108
SCFT2 and DRCFT2 (Complex-to-Real Fourier Transform in Two Dimensions) 1115
SCFT3 and DCFT3 (Complex Fourier Transform in Three Dimensions) 1123
SRCFT3 and DRCFT3 (Real-to-Complex Fourier Transform in Three Dimensions) 1130
SCFT3 and DRCFT3 (Complex-to-Real Fourier Transform in Three Dimensions) 1137

Convolution and Correlation Subroutines 1144
SCON and SCOR (Convolution or Correlation of One Sequence with One or More Sequences) 1145
SCOND and SCORD (Convolution or Correlation of One Sequence with Another Sequence Using a Direct Method) 1151
SCONF and SCORF (Convolution or Correlation of One Sequence with One or More Sequences Using the Mixed-Radix Fourier Method) 1157
SDCON, DDCON, SDCOR, and DDCOR (Convolution or Correlation with Decimated Output Using a Direct Method) 1167
SACORF (Autocorrelation of One or More Sequences) 1172

Chapter 13. Sorting and Searching 1201
Overview of the Sorting and Searching Subroutines 1201
Use Considerations 1201

Contents ix
Tables

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Abbreviated names</td>
<td>xx</td>
</tr>
<tr>
<td>2. Summary of typographic conventions</td>
<td>xxi</td>
</tr>
<tr>
<td>3. Syntax for all subroutines</td>
<td>xxvii</td>
</tr>
<tr>
<td>4. Syntax for CBLAS subroutines</td>
<td>xxvii</td>
</tr>
<tr>
<td>5. Syntax for LAPACKE subroutines</td>
<td>xxvii</td>
</tr>
<tr>
<td>6. Translating character argument values to CBLAS enumerated types</td>
<td>xxviii</td>
</tr>
<tr>
<td>7. Summary of ESSL Subroutines</td>
<td>xxi</td>
</tr>
<tr>
<td>8. Operating systems supported by ESSL</td>
<td>v</td>
</tr>
<tr>
<td>9. Required Software Products for ESSL on AIX</td>
<td>98</td>
</tr>
<tr>
<td>10. Required Software Products for ESSL</td>
<td>100</td>
</tr>
<tr>
<td>11. Software needed to display various formats of ESSL online information</td>
<td>11</td>
</tr>
<tr>
<td>12. List of Vector-Scalar Linear Algebra Subprograms</td>
<td>12</td>
</tr>
<tr>
<td>13. List of Sparse Vector-Scalar Linear Algebra Subprograms</td>
<td>14</td>
</tr>
<tr>
<td>14. List of Matrix-Vector Linear Algebra Subprograms</td>
<td>14</td>
</tr>
<tr>
<td>15. List of Sparse Matrix-Vector Linear Algebra Subprograms</td>
<td>16</td>
</tr>
<tr>
<td>16. List of Matrix Operation Subroutines</td>
<td>16</td>
</tr>
<tr>
<td>17. List of LAPACK Dense Linear Algebraic Equation Subroutines</td>
<td>18</td>
</tr>
<tr>
<td>18. List of Dense Linear Algebraic Equation Subroutines</td>
<td>20</td>
</tr>
<tr>
<td>19. List of LAPACK Banded Linear Algebraic Equation Subroutines</td>
<td>21</td>
</tr>
<tr>
<td>20. List of non-LAPACK Banded Linear Algebraic Equation Subroutines</td>
<td>22</td>
</tr>
<tr>
<td>21. List of Sparse Linear Algebraic Equation Subroutines</td>
<td>23</td>
</tr>
<tr>
<td>22. List of LAPACK Linear Least Squares Subroutines</td>
<td>24</td>
</tr>
<tr>
<td>23. List of Non-LAPACK Linear Least Squares Subroutines</td>
<td>24</td>
</tr>
<tr>
<td>24. List of LAPACK Eigensystem Analysis Subroutines</td>
<td>25</td>
</tr>
<tr>
<td>25. List of Fourier Transform Subroutines</td>
<td>26</td>
</tr>
<tr>
<td>26. List of Convolution and Correlation Subroutines</td>
<td>27</td>
</tr>
<tr>
<td>27. List of Related-Computation Subroutines</td>
<td>27</td>
</tr>
<tr>
<td>28. List of Sorting and Searching Subroutines</td>
<td>28</td>
</tr>
<tr>
<td>29. List of Interpolation Subroutines</td>
<td>28</td>
</tr>
<tr>
<td>30. List of Numerical Quadrature Subroutines</td>
<td>29</td>
</tr>
<tr>
<td>31. List of Random Number Generation Initialization Subroutines</td>
<td>29</td>
</tr>
<tr>
<td>32. List of Random Number Generation Subroutines</td>
<td>29</td>
</tr>
<tr>
<td>33. List of Utility Subroutines</td>
<td>30</td>
</tr>
<tr>
<td>34. VSX Alignment Requirements for SIMD Algorithms in Linear Algebra Subroutines</td>
<td>33</td>
</tr>
<tr>
<td>35. VSX Alignment Requirements for SIMD Algorithms in Fourier Transform Subroutines and Convolution and Correlation Subroutines</td>
<td>33</td>
</tr>
<tr>
<td>36. ESSL Subroutines that Automatically Use SIMD Algorithms When Alignment Restrictions are Met on VSX-enabled Processors</td>
<td>34</td>
</tr>
<tr>
<td>37. Altivec-Enabled Processor Alignment Restrictions for SIMD Algorithms in Linear Algebra Subroutines</td>
<td>36</td>
</tr>
<tr>
<td>38. Altivec-Enabled Processor Alignment Restrictions for SIMD Algorithms in Fourier Transform and Fourier Method Convolution and Correlation Subroutines</td>
<td>36</td>
</tr>
<tr>
<td>39. ESSL Subroutines that Automatically Use SIMD Algorithms When Alignment Restrictions are Met on POWER 6 Altivec-Enabled Processors</td>
<td>37</td>
</tr>
<tr>
<td>40. Multithreaded Subroutines</td>
<td>39</td>
</tr>
<tr>
<td>41. ESSL Subroutines Requiring Auxiliary Working Storage</td>
<td>51</td>
</tr>
<tr>
<td>42. Example of Input-Argument Error Recovery for Auxiliary Storage Sizes</td>
<td>57</td>
</tr>
<tr>
<td>43. ESSL Subroutines Requiring Transform Lengths</td>
<td>58</td>
</tr>
<tr>
<td>44. Example of Input-Argument Error Recovery for Transform Lengths</td>
<td>62</td>
</tr>
<tr>
<td>45. ESSL Error Option Table Default Values</td>
<td>71</td>
</tr>
<tr>
<td>46. Scalar Data Types in Fortran Programs</td>
<td>134</td>
</tr>
<tr>
<td>47. Scalar Data Types in C Programs</td>
<td>153</td>
</tr>
<tr>
<td>48. Scalar Data Types in C++ Programs</td>
<td>169</td>
</tr>
<tr>
<td>49. Fortran Compile Commands on AIX</td>
<td>186</td>
</tr>
<tr>
<td>50. Fortran Compile Commands on AIX for use with FFTW Wrapper libraries</td>
<td>186</td>
</tr>
<tr>
<td>51. C Compile and Link Commands on AIX</td>
<td>187</td>
</tr>
<tr>
<td>52. C Compile and Link Commands on AIX for use with FFTW Wrapper Libraries</td>
<td>187</td>
</tr>
<tr>
<td>53. C++ Compile and Link Commands on AIX</td>
<td>189</td>
</tr>
<tr>
<td>54. C++ Compile and Link Commands on AIX for use with FFTW Wrapper Libraries</td>
<td>190</td>
</tr>
<tr>
<td>55. Fortran Compile Commands on Linux (little endian mode)</td>
<td>192</td>
</tr>
<tr>
<td>56. Fortran Compile Commands on Linux for Use with FFTW Wrapper Libraries</td>
<td>193</td>
</tr>
<tr>
<td>57. C Compile and Link Commands on Linux (little endian mode)</td>
<td>193</td>
</tr>
<tr>
<td>58. C Compile and Link Commands on Linux for Use with FFTW Wrapper Libraries (little endian mode)</td>
<td>194</td>
</tr>
<tr>
<td>59. gcc Compile and Link Commands on Linux (little endian mode)</td>
<td>195</td>
</tr>
<tr>
<td>60. gcc Compile and Link Commands on Linux for Use with FFTW Wrapper Libraries (little endian mode)</td>
<td>195</td>
</tr>
<tr>
<td>61. C++ Compile and Link Commands on Linux (little endian mode)</td>
<td>196</td>
</tr>
<tr>
<td>62. C++ Compile and Link Commands on Linux for Use with FFTW Wrapper Libraries (little endian mode)</td>
<td>197</td>
</tr>
<tr>
<td>Page</td>
<td>Section</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
</tr>
<tr>
<td>63.</td>
<td>g++ Compile and Link Commands on Linux (little endian mode)</td>
</tr>
<tr>
<td>64.</td>
<td>g++ Compile and Link Commands on Linux for Use with FFTW Wrapper Libraries (little endian mode)</td>
</tr>
<tr>
<td>65.</td>
<td>CBLAS enumerated types changing to type definitions for ESSL Version 5 Release 5.</td>
</tr>
<tr>
<td>66.</td>
<td>Replacing Non-LAPACK-Conforming subroutines with LAPACK subroutines</td>
</tr>
<tr>
<td>67.</td>
<td>Product File Set and Package Names</td>
</tr>
<tr>
<td>68.</td>
<td>List of Vector-Scalar Linear Algebra Subprograms.</td>
</tr>
<tr>
<td>69.</td>
<td>List of Sparse Vector-Scalar Linear Algebra Subprograms.</td>
</tr>
<tr>
<td>70.</td>
<td>List of Matrix-Vector Linear Algebra Subprograms.</td>
</tr>
<tr>
<td>71.</td>
<td>List of Sparse Matrix-Vector Linear Algebra Subprograms.</td>
</tr>
<tr>
<td>72.</td>
<td>Data Types</td>
</tr>
<tr>
<td>73.</td>
<td>Data Types</td>
</tr>
<tr>
<td>74.</td>
<td>Data Types</td>
</tr>
<tr>
<td>75.</td>
<td>Data Types</td>
</tr>
<tr>
<td>76.</td>
<td>Data Types</td>
</tr>
<tr>
<td>77.</td>
<td>Data Types</td>
</tr>
<tr>
<td>78.</td>
<td>Data Types</td>
</tr>
<tr>
<td>79.</td>
<td>Data Types</td>
</tr>
<tr>
<td>80.</td>
<td>Data Types</td>
</tr>
<tr>
<td>81.</td>
<td>Data Types</td>
</tr>
<tr>
<td>82.</td>
<td>Data Types</td>
</tr>
<tr>
<td>83.</td>
<td>Data Types</td>
</tr>
<tr>
<td>84.</td>
<td>Data Types</td>
</tr>
<tr>
<td>85.</td>
<td>Data Types</td>
</tr>
<tr>
<td>86.</td>
<td>Data Types</td>
</tr>
<tr>
<td>87.</td>
<td>Data Types</td>
</tr>
<tr>
<td>88.</td>
<td>Data Types</td>
</tr>
<tr>
<td>89.</td>
<td>Data Types</td>
</tr>
<tr>
<td>90.</td>
<td>Data Types</td>
</tr>
<tr>
<td>91.</td>
<td>Data Types</td>
</tr>
<tr>
<td>92.</td>
<td>Data Types</td>
</tr>
<tr>
<td>93.</td>
<td>Data Types</td>
</tr>
<tr>
<td>94.</td>
<td>Data Types</td>
</tr>
<tr>
<td>95.</td>
<td>Data Types</td>
</tr>
<tr>
<td>96.</td>
<td>Data Types</td>
</tr>
<tr>
<td>97.</td>
<td>Data Types</td>
</tr>
<tr>
<td>98.</td>
<td>Data Types</td>
</tr>
<tr>
<td>99.</td>
<td>Data Types</td>
</tr>
<tr>
<td>100.</td>
<td>Data Types</td>
</tr>
<tr>
<td>101.</td>
<td>Data Types</td>
</tr>
<tr>
<td>102.</td>
<td>Data Types</td>
</tr>
<tr>
<td>103.</td>
<td>Data Types</td>
</tr>
<tr>
<td>104.</td>
<td>Data Types</td>
</tr>
<tr>
<td>105.</td>
<td>Data Types</td>
</tr>
<tr>
<td>106.</td>
<td>Data Types</td>
</tr>
<tr>
<td>107.</td>
<td>Data Types</td>
</tr>
<tr>
<td>108.</td>
<td>Data Types</td>
</tr>
<tr>
<td>109.</td>
<td>Data Types</td>
</tr>
<tr>
<td>110.</td>
<td>Data Types</td>
</tr>
<tr>
<td>111.</td>
<td>List of Matrix Operation Subroutines</td>
</tr>
<tr>
<td>112.</td>
<td>Data Types</td>
</tr>
<tr>
<td>113.</td>
<td>Data Types</td>
</tr>
<tr>
<td>114.</td>
<td>Data Types</td>
</tr>
<tr>
<td>115.</td>
<td>Data Types</td>
</tr>
<tr>
<td>116.</td>
<td>Data Types</td>
</tr>
<tr>
<td>117.</td>
<td>Data Types</td>
</tr>
<tr>
<td>118.</td>
<td>Data Types</td>
</tr>
<tr>
<td>119.</td>
<td>Data Types</td>
</tr>
<tr>
<td>120.</td>
<td>Data Types</td>
</tr>
<tr>
<td>121.</td>
<td>Data Types</td>
</tr>
<tr>
<td>122.</td>
<td>Data Types</td>
</tr>
<tr>
<td>123.</td>
<td>Data Types</td>
</tr>
<tr>
<td>124.</td>
<td>List of LAPACK Dense Linear Algebraic Equation Subroutines</td>
</tr>
<tr>
<td>125.</td>
<td>List of Dense Linear Algebraic Equation Subroutines</td>
</tr>
<tr>
<td>126.</td>
<td>List of LAPACK Banded Linear Algebraic Equation Subroutines</td>
</tr>
<tr>
<td>127.</td>
<td>List of non-LAPACK Banded Linear Algebraic Equation Subroutines</td>
</tr>
<tr>
<td>128.</td>
<td>List of Sparse Linear Algebraic Equation Subroutines</td>
</tr>
<tr>
<td>129.</td>
<td>List of LAPACK Linear Least Squares Subroutines</td>
</tr>
<tr>
<td>130.</td>
<td>List of Non-LAPACK Linear Least Squares Subroutines</td>
</tr>
<tr>
<td>131.</td>
<td>Data Types</td>
</tr>
<tr>
<td>132.</td>
<td>Data Types</td>
</tr>
<tr>
<td>133.</td>
<td>Data Types</td>
</tr>
<tr>
<td>134.</td>
<td>Data Types</td>
</tr>
<tr>
<td>135.</td>
<td>Data Types</td>
</tr>
<tr>
<td>136.</td>
<td>Data Types</td>
</tr>
<tr>
<td>137.</td>
<td>Data Types</td>
</tr>
<tr>
<td>138.</td>
<td>Data Types</td>
</tr>
<tr>
<td>139.</td>
<td>Data Types</td>
</tr>
<tr>
<td>140.</td>
<td>Data Types</td>
</tr>
<tr>
<td>141.</td>
<td>Data Types</td>
</tr>
<tr>
<td>142.</td>
<td>Data Types</td>
</tr>
<tr>
<td>143.</td>
<td>Data Types</td>
</tr>
<tr>
<td>144.</td>
<td>Data Types</td>
</tr>
<tr>
<td>145.</td>
<td>Data Types</td>
</tr>
<tr>
<td>146.</td>
<td>Data Types</td>
</tr>
<tr>
<td>147.</td>
<td>Data Types</td>
</tr>
<tr>
<td>148.</td>
<td>Data Types</td>
</tr>
<tr>
<td>149.</td>
<td>Data Types</td>
</tr>
<tr>
<td>150.</td>
<td>Data Types</td>
</tr>
<tr>
<td>151.</td>
<td>Data Types</td>
</tr>
<tr>
<td>152.</td>
<td>Data Types</td>
</tr>
<tr>
<td>153.</td>
<td>Data Types</td>
</tr>
<tr>
<td>154.</td>
<td>Data Types</td>
</tr>
<tr>
<td>155.</td>
<td>Data Types</td>
</tr>
<tr>
<td>156.</td>
<td>Data Types</td>
</tr>
<tr>
<td>157.</td>
<td>Data Types</td>
</tr>
<tr>
<td>158.</td>
<td>Data Types</td>
</tr>
<tr>
<td>159.</td>
<td>Data Types</td>
</tr>
<tr>
<td>160.</td>
<td>Data Types</td>
</tr>
<tr>
<td>161.</td>
<td>Data Types</td>
</tr>
<tr>
<td>162.</td>
<td>Data Types</td>
</tr>
<tr>
<td>163.</td>
<td>Data Types</td>
</tr>
<tr>
<td>164.</td>
<td>Data Types</td>
</tr>
<tr>
<td>165.</td>
<td>Data Types</td>
</tr>
<tr>
<td>166.</td>
<td>Data Types</td>
</tr>
<tr>
<td>167.</td>
<td>Data Types</td>
</tr>
<tr>
<td>168.</td>
<td>Data Types</td>
</tr>
<tr>
<td>Page</td>
<td>Data Types</td>
</tr>
<tr>
<td>------</td>
<td>------------</td>
</tr>
<tr>
<td>169.</td>
<td>Data Types</td>
</tr>
<tr>
<td>170.</td>
<td>Data Types</td>
</tr>
<tr>
<td>171.</td>
<td>Data Types</td>
</tr>
<tr>
<td>172.</td>
<td>Data Types</td>
</tr>
<tr>
<td>173.</td>
<td>Data Types</td>
</tr>
<tr>
<td>174.</td>
<td>Data Types</td>
</tr>
<tr>
<td>175.</td>
<td>Data Types</td>
</tr>
<tr>
<td>176.</td>
<td>Data Types</td>
</tr>
<tr>
<td>177.</td>
<td>Data Types</td>
</tr>
<tr>
<td>178.</td>
<td>Data Types</td>
</tr>
<tr>
<td>179.</td>
<td>Data Types</td>
</tr>
<tr>
<td>180.</td>
<td>Data Types</td>
</tr>
<tr>
<td>181.</td>
<td>Data Types</td>
</tr>
<tr>
<td>182.</td>
<td>Data Types</td>
</tr>
<tr>
<td>183.</td>
<td>Data Types</td>
</tr>
<tr>
<td>184.</td>
<td>Data Types</td>
</tr>
<tr>
<td>185.</td>
<td>Data Types</td>
</tr>
<tr>
<td>186.</td>
<td>Data Types</td>
</tr>
<tr>
<td>187.</td>
<td>Data Types</td>
</tr>
<tr>
<td>188.</td>
<td>Data Types</td>
</tr>
<tr>
<td>189.</td>
<td>List of LAPACK Eigensystem Analysis Subroutines</td>
</tr>
<tr>
<td>190.</td>
<td>Data Types</td>
</tr>
<tr>
<td>191.</td>
<td>Data Types</td>
</tr>
<tr>
<td>192.</td>
<td>Data Types</td>
</tr>
<tr>
<td>193.</td>
<td>Data Types</td>
</tr>
<tr>
<td>194.</td>
<td>Data Types</td>
</tr>
<tr>
<td>195.</td>
<td>List of Fourier Transform Subroutines</td>
</tr>
<tr>
<td>196.</td>
<td>List of Convolution and Correlation Subroutines</td>
</tr>
<tr>
<td>197.</td>
<td>List of Related-Computation Subroutines</td>
</tr>
<tr>
<td>198.</td>
<td>Fourier Transform subroutines allowing all lengths between 0 and 1073479680</td>
</tr>
<tr>
<td>199.</td>
<td>Fourier Transform subroutines whose lengths are limited to those in Figure 15 on page 1029</td>
</tr>
<tr>
<td>200.</td>
<td>Data Types</td>
</tr>
<tr>
<td>201.</td>
<td>Data Types</td>
</tr>
<tr>
<td>202.</td>
<td>Data Types</td>
</tr>
<tr>
<td>203.</td>
<td>Data Types</td>
</tr>
<tr>
<td>204.</td>
<td>Data Types</td>
</tr>
<tr>
<td>205.</td>
<td>Data Types</td>
</tr>
<tr>
<td>206.</td>
<td>Data Types</td>
</tr>
<tr>
<td>207.</td>
<td>Data Types</td>
</tr>
<tr>
<td>208.</td>
<td>Data Types</td>
</tr>
<tr>
<td>209.</td>
<td>Data Types</td>
</tr>
<tr>
<td>210.</td>
<td>Data Types</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
About this information

This provides guide and reference information 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 documentation 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, and C++ programming.

How to Use This Information

Part 1, “Guide Information,” on page 1 provides guidance information for using ESSL. It covers the user-oriented tasks of learning, designing, coding, migrating, processing, and diagnosing. Refer to the following when performing any of these tasks:

- **Chapter 1, “Introduction and Requirements,” on page 3** gives an introduction to ESSL, providing highlights and general information. Read this first to determine the aspects of ESSL you want to use.
- **Chapter 2, “Planning Your Program,” on page 31** provides ESSL-specific information that helps you design your program. Read this before designing your program.
- **Chapter 3, “Setting Up Your Data Structures,” on page 75** 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,” on page 133** tells you how to code your scalar and array data, how to code calls to ESSL in Fortran, C, and C++ programs, and how to do the coding necessary to handle errors. Use this information when coding your program.
- **Chapter 5, “Processing Your Program,” on page 185** 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,” on page 201** explains all aspects of migration to ESSL, to this version of ESSL, to different processors, and to future releases and future processors. Read this before starting to design your program.
- **Chapter 7, “Handling Problems,” on page 207** provides diagnostic procedures for analyzing all ESSL problems. When you encounter a problem, use the symptom indexes at the beginning to guide you to the appropriate diagnostic procedure.

Part 2, “Reference Information,” on page 223 provides reference information you need to code the ESSL calling sequences. It covers each of the mathematical areas of ESSL, and the utility subroutines. The information for each subroutine area begins with an introduction, followed by the subroutine descriptions. Each introduction applies to all the subroutines in that area and is especially important
in planning your use of the subroutines and avoiding problems. Use the appropriate information when coding your program:

- **Chapter 8, “Linear Algebra Subprograms,” on page 225**
- **Chapter 9, “Matrix Operations,” on page 429**
- **Chapter 10, “Linear Algebraic Equations,” on page 517**
- **Chapter 11, “Eigensystem Analysis,” on page 939**
- **Chapter 12, “Fourier Transforms, Convolutions and Correlations, and Related Computations,” on page 1025**
- **Chapter 13, “Sorting and Searching,” on page 1201**
- **Chapter 14, “Interpolation,” on page 1221**
- **Chapter 15, “Numerical Quadrature,” on page 1243**
- **Chapter 16, “Random Number Generation,” on page 1269**
- **Chapter 17, “Utilities,” on page 1293**

**Appendix A, “Basic Linear Algebra Subprograms (BLAS) and Complex BLAS (CBLAS),” on page 1339** provides a list of the Level 1, 2, and 3 Basic Linear Algebra Subprograms (BLAS) included in ESSL.

**Appendix B, “LAPACK and LAPACKE,” on page 1343** provides a list of the LAPACK subroutines included in ESSL.

**Appendix C, “FFTW Version 3.1.2 to ESSL Wrapper Libraries,” on page 1349** provides a list of the FFTW subroutines included in ESSL.

**Bibliography” on page 1363** provides information about publications related to ESSL. Use it when you need more information than this documentation provides.

### Where to Find Related Publications

ESSL documentation, as well as other related information, can be displayed or downloaded from the Internet at the following URL:


### Related Publications

The related Web sites listed below may be useful to you when using ESSL.

<table>
<thead>
<tr>
<th>Product</th>
<th>Web site URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIX®</td>
<td><a href="http://www.ibm.com/servers/aix">http://www.ibm.com/servers/aix</a></td>
</tr>
<tr>
<td>Linux</td>
<td>For general information and documentation on Linux: <a href="http://www.tldp.org/">http://www.tldp.org/</a></td>
</tr>
<tr>
<td></td>
<td>For information about the standard Linux installation procedure using the</td>
</tr>
<tr>
<td></td>
<td>RPM Package Manager (RPM): <a href="http://www.rpm.org/">http://www.rpm.org/</a></td>
</tr>
<tr>
<td></td>
<td>For information about IBM-related offerings for Linux: <a href="http://www.ibm.com/linux/">http://www.ibm.com/linux/</a></td>
</tr>
<tr>
<td>C and C++ XL Fortran</td>
<td><a href="http://www.ibm.com/support/knowledgecenter/">http://www.ibm.com/support/knowledgecenter/</a>, under ‘Rational’ on the</td>
</tr>
<tr>
<td></td>
<td>left hand pane.</td>
</tr>
</tbody>
</table>
Using Bibliography References

Special references are made throughout this documentation 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.

IBM Request for Enhancement (RFE) Community

The IBM Requests for Enhancements (RFEs) Community provides an opportunity to collaborate directly with the IBM product development teams and other product users on RFEs.

You can submit ESSL RFEs at the Servers and Systems Software RFE Community:
https://www.ibm.com/developerworks/rfe/?BRAND_ID=352

How to Find a Subroutine Description

If you want to locate a subroutine description and you know the subroutine name, you can find it listed individually or under the entry “subroutines, ESSL” in the Index.

How to Interpret the Subroutine Names with a Prefix Underscore

A name specified 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.

Special Terms

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

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, “Basic Linear Algebra Subprograms (BLAS) and Complex BLAS (CBLAS),” on page 1339.

Abbreviated Names

The abbreviated names used are defined below.

Table 1. Abbreviated names

<table>
<thead>
<tr>
<th>Short Name</th>
<th>Full Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIX</td>
<td>Advanced Interactive Executive</td>
</tr>
<tr>
<td>AltiVec*</td>
<td>A tradename, owned solely by Freescale Semiconductor, Inc., for a floating point and integer SIMD instruction set designed and owned by Apple, IBM, and Freescale (formerly the Semiconductor Products Sector of Motorola).</td>
</tr>
<tr>
<td>BLAS</td>
<td>Basic Linear Algebra Subprograms (see Appendix A, “Basic Linear Algebra Subprograms (BLAS) and Complex BLAS (CBLAS),” on page 1339)</td>
</tr>
<tr>
<td>CBLAS</td>
<td>C interface to the BLAS (see Appendix A, “Basic Linear Algebra Subprograms (BLAS) and Complex BLAS (CBLAS),” on page 1339)</td>
</tr>
<tr>
<td>CUDA</td>
<td>Parallel computing platform and programming model invented by NVIDIA</td>
</tr>
<tr>
<td>ESSL</td>
<td>IBM Engineering and Scientific Subroutine Library</td>
</tr>
<tr>
<td>FFTTW</td>
<td>Fastest Fourier Transform in the West (see Appendix C, “FFTW Version 3.1.2 to ESSL Wrapper Libraries,” on page 1349)</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics processing unit</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext Markup Language</td>
</tr>
<tr>
<td>LAPACK</td>
<td>Linear Algebra Package (see Appendix B, “LAPACK and LAPACKE,” on page 1343)</td>
</tr>
<tr>
<td>LAPACKE</td>
<td>C Interface to LAPACK (see Appendix B, “LAPACK and LAPACKE,” on page 1343)</td>
</tr>
<tr>
<td>OpenMP</td>
<td>Open Multi-Processing</td>
</tr>
<tr>
<td>SL MATH</td>
<td>Subroutine Library—Mathematics</td>
</tr>
<tr>
<td>SMP</td>
<td>Symmetric Multi-Processing</td>
</tr>
<tr>
<td>SSP</td>
<td>Scientific Subroutine Package</td>
</tr>
</tbody>
</table>

*AltiVec is a trademark of Freescale Semiconductor, Inc.*
Conventions and terminology used

Table 2 describes the typographic conventions used.

<table>
<thead>
<tr>
<th>Table 2. Summary of typographic conventions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Typographic</strong></td>
</tr>
<tr>
<td><strong>Bold</strong></td>
</tr>
</tbody>
</table>
| **Italic** | • **Italic** words or characters represent variable values that you must supply.  
| &nbsp; &nbsp; &nbsp; &nbsp; | • **Italics** are also used for book titles and for general emphasis in text. |
| **Constant width** | Examples and information that the system displays appear in constant width typeface. |
| **[]** | Brackets enclose optional items in format and syntax descriptions. |
| **{}** | Braces enclose a list from which you must choose an item in format and syntax descriptions. |
| **|** | A vertical bar separates items in a list of choices. (In other words, it means “or.”) |
| **< >** | Angle brackets (less-than and greater-than) enclose the name of a key on the keyboard. For example, **<Enter>** refers to the key on your terminal or workstation that is labeled with the word Enter. |
| ... | An ellipsis indicates that you can repeat the preceding item one or more times. |
| **<Ctrl-x>** | The notation **<Ctrl-x>** indicates a control character sequence. For example, **<Ctrl-c>** means that you hold down the control key while pressing **<c>**. |
| \ | The continuation character is used in coding examples for formatting purposes. |

Conventions that are consistent with traditional mathematical usage are followed.

**Fonts**

A variety of special fonts are used to distinguish between many mathematical and programming items. These are defined below.

<table>
<thead>
<tr>
<th><strong>Special Font</strong></th>
<th><strong>Example</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Italic with no subscripts</td>
<td><em>m</em>, <em>inc1x</em>, <em>aux</em>, <em>iopt</em></td>
<td>Calling sequence argument or mathematical variable</td>
</tr>
<tr>
<td>Italic with subscripts</td>
<td><em>x</em>, <em>a_{i,j}</em>, <em>x_{i,j}</em>, <em>x_{i,j,k}</em>, <em>x_{i,j,k,l}</em></td>
<td>Element of a vector, matrix, or sequence</td>
</tr>
<tr>
<td>Bold italic lowercase</td>
<td><em>x</em>, <em>y</em>, <em>z</em></td>
<td>Vector or sequence</td>
</tr>
<tr>
<td>Bold italic uppercase</td>
<td><em>A</em>, <em>B</em>, <em>C</em></td>
<td>Matrix</td>
</tr>
<tr>
<td>Gothic uppercase</td>
<td><em>A</em>, <em>B</em>, <em>C</em>, <em>AGB</em>, <em>IM=ISMAX(4, X, 2)</em></td>
<td>Array</td>
</tr>
</tbody>
</table>

**Special Notations and Conventions**

This explains the special notations and conventions used to describe various types of data.
Scalar Data
Following are the special notations used in the examples 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, and C++, see Chapter 4, “Coding Your Program,” on page 133.

<table>
<thead>
<tr>
<th>Data Item</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character item</td>
<td>'T'</td>
<td>Character(s) in single quotation marks</td>
</tr>
<tr>
<td>Hexadecimal string</td>
<td>X'97FA00C1'</td>
<td>String of 4-bit hexadecimal characters</td>
</tr>
<tr>
<td>Logical item</td>
<td>.TRUE., .FALSE.</td>
<td>True or false logical value, as indicated</td>
</tr>
<tr>
<td>Integer data</td>
<td>1</td>
<td>Number with no decimal point</td>
</tr>
<tr>
<td>Real data</td>
<td>1.6</td>
<td>Number with a decimal point</td>
</tr>
<tr>
<td>Complex data</td>
<td>(1.0, -2.9)</td>
<td>Real part followed by the imaginary part</td>
</tr>
<tr>
<td>Continuation</td>
<td>1.6666</td>
<td>Continue the last digit (1.6666666... and so forth)</td>
</tr>
</tbody>
</table>

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:

$$\begin{bmatrix}
  x_1 \\
  x_2 \\
  x_3 \\
  \vdots \\
  x_n
\end{bmatrix}$$

For a definition of vector, see “Vectors” on page 75.

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

$$\begin{bmatrix}
  a_{11} & \cdots & a_{1n} \\
  \vdots & \ddots & \vdots \\
  \vdots & & \vdots \\
  a_{m1} & \cdots & a_{mn}
\end{bmatrix}$$

For a definition of matrix, see “Matrices” on page 81.
Sequences
Sequences are used in the areas of sorting, searching, Fourier transforms, convolutions, and correlations. For a definition of sequences, see "Sequences" on page 128.

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:

\[
\begin{array}{cccc}
  a_{11} & a_{12} & \cdots & a_{1n} \\
  a_{21} & a_{22} & \cdots & a_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{m1} & a_{m2} & \cdots & a_{mn}
\end{array}
\]

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:

\[
\begin{array}{cccc}
  a_{111} & a_{121} & \cdots & a_{1n1} \\
  a_{211} & a_{221} & a_{2n1} & \cdots \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{m11} & a_{m21} & \cdots & a_{mn1}
\end{array} \begin{array}{cccc}
  a_{112} & a_{122} & \cdots & a_{1n2} \\
  a_{212} & a_{222} & a_{2n2} & \cdots \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{m12} & a_{m22} & \cdots & a_{mn2}
\end{array} \begin{array}{cccc}
  a_{11p} & a_{12p} & \cdots & a_{1np} \\
  a_{21p} & a_{22p} & a_{2np} & \cdots \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{m1p} & a_{m2p} & \cdots & a_{mnp}
\end{array}
\]

Arrays
Arrays contain vectors, matrices, or sequences. For a definition of array, see "How Do You Set Up Your Arrays?" on page 48.

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:

\[
\begin{bmatrix}
1.0 & 11.0 & 5.0 & 25.0 \\
2.0 & 12.0 & 6.0 & 26.0 \\
3.0 & 13.0 & 7.0 & 27.0 \\
4.0 & 14.0 & 8.0 & 28.0
\end{bmatrix}
\]

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:

\[
\begin{bmatrix}
\ldots & \ldots & \ldots & \ldots \\
\ldots & \ldots & \ldots & \ldots \\
1.0 & 2.0 & 5.0 & 4.0 \\
2.0 & 3.0 & 6.0 & 3.0 \\
3.0 & 4.0 & 7.0 & 2.0 \\
4.0 & 5.0 & 8.0 & 1.0 \\
\ldots & \ldots & \ldots & \ldots
\end{bmatrix}
\]

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:

\[
\begin{bmatrix}
1.0 & 11.0 & 5.0 & 25.0 \\
2.0 & 12.0 & 6.0 & 26.0 \\
3.0 & 13.0 & 7.0 & 27.0 \\
4.0 & 14.0 & 8.0 & 28.0
\end{bmatrix}
\begin{bmatrix}
10.0 & 111.0 & 15.0 & 125.0 \\
20.0 & 112.0 & 16.0 & 126.0 \\
30.0 & 113.0 & 17.0 & 127.0 \\
40.0 & 114.0 & 18.0 & 128.0
\end{bmatrix}
\begin{bmatrix}
100.0 & 11.0 & 15.0 & 25.0 \\
200.0 & 12.0 & 16.0 & 26.0 \\
300.0 & 13.0 & 17.0 & 27.0 \\
400.0 & 14.0 & 18.0 & 28.0
\end{bmatrix}
\]

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 are consistent with traditional mathematical and programming usage. These conventions are explained below, along with special abbreviations that are associated with specific values.

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greek letters: $a$, $\sigma$, $\omega$, $\Omega$</td>
<td>Symbolic scalar values</td>
</tr>
<tr>
<td>$</td>
<td>a</td>
</tr>
<tr>
<td>$a \cdot b$</td>
<td>The dot product of $a$ and $b$</td>
</tr>
<tr>
<td>$x_i$</td>
<td>The $i$-th element of vector $x$</td>
</tr>
<tr>
<td>$c_{ij}$</td>
<td>The element in matrix $C$ at row $i$ and column $j$</td>
</tr>
<tr>
<td>$x_1 \ldots x_n$</td>
<td>Elements from $x_1$ to $x_n$</td>
</tr>
<tr>
<td>$i = 1, n$</td>
<td>$i$ is assigned the values 1 to $n$</td>
</tr>
<tr>
<td>$y \times x$</td>
<td>Vector $y$ is replaced by vector $x$</td>
</tr>
<tr>
<td>$xy$</td>
<td>Vector $x$ times vector $y$</td>
</tr>
<tr>
<td>Item</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>$AX \cong B$</td>
<td>$AX$ is congruent to $B$</td>
</tr>
<tr>
<td>$a^k$</td>
<td>$a$ raised to the $k$ power</td>
</tr>
<tr>
<td>$e^x$</td>
<td>Exponential function of $x$</td>
</tr>
<tr>
<td>$A^T; x^T$</td>
<td>The transpose of matrix $A$; the transpose of vector $x$</td>
</tr>
<tr>
<td>$\bar{x}; \overline{A}$</td>
<td>The complex conjugate of vector $x$; the complex conjugate of matrix $A$</td>
</tr>
<tr>
<td>$\bar{x}<em>i; \overline{c}</em>{jk}$</td>
<td>The complex conjugate of the complex vector element $x_i$, where:</td>
</tr>
<tr>
<td></td>
<td>if $x_i = (a_i, b_i)$,</td>
</tr>
<tr>
<td></td>
<td>then $\bar{x}_i = (a_i, -b_i)$</td>
</tr>
<tr>
<td>$x^{H}; A^{H}$</td>
<td>The complex conjugate transpose of vector $x$; the complex conjugate transpose of matrix $A$</td>
</tr>
<tr>
<td>$\sum_{i=1}^{n} x_i$</td>
<td>The sum of elements $x_1$ to $x_n$</td>
</tr>
<tr>
<td>$\sqrt{a + b}$</td>
<td>The square root of $a + b$</td>
</tr>
<tr>
<td>$\int_{a}^{b} f(x) , dx$</td>
<td>The integral from $a$ to $b$ of $f(x) , dx$</td>
</tr>
<tr>
<td>$| x |_2$</td>
<td>The Euclidean norm of vector $x$, defined as:</td>
</tr>
<tr>
<td></td>
<td>$\sqrt{\sum_{j=1}^{n}</td>
</tr>
<tr>
<td>$| A |_1$</td>
<td>The one norm of matrix $A$, defined as:</td>
</tr>
<tr>
<td></td>
<td>$\max \left{ \sum_{i=1}^{m}</td>
</tr>
<tr>
<td>$| A |_2$</td>
<td>The spectral norm of matrix $A$, defined as:</td>
</tr>
<tr>
<td></td>
<td>$\max { | Ax |_2 : | x |_2 = 1 }$</td>
</tr>
<tr>
<td>$| A |_F$</td>
<td>The Frobenius or Euclidean norm of matrix $A$, defined as:</td>
</tr>
<tr>
<td></td>
<td>$\sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n}</td>
</tr>
<tr>
<td>$| A |_\infty$</td>
<td>The infinity norm of matrix $A$, defined as:</td>
</tr>
<tr>
<td></td>
<td>$\max \left{ \sum_{j=1}^{n}</td>
</tr>
<tr>
<td>$A^{-1}$</td>
<td>The inverse of matrix $A$</td>
</tr>
<tr>
<td>Item</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>$A^T$</td>
<td>The transpose of $A$ inverse</td>
</tr>
<tr>
<td>$</td>
<td>A</td>
</tr>
<tr>
<td>$m$ by $n$ matrix $A$</td>
<td>Matrix $A$ has $m$ rows and $n$ columns</td>
</tr>
<tr>
<td>$\sin a$</td>
<td>The sine of $a$</td>
</tr>
<tr>
<td>$\cos b$</td>
<td>The cosine of $b$</td>
</tr>
<tr>
<td>SIGN $(a)$</td>
<td>The sign of $a$; the result is either + or -</td>
</tr>
<tr>
<td>address $[a]$</td>
<td>The storage address of $a$</td>
</tr>
<tr>
<td>$\max(x)$</td>
<td>The maximum element in vector $x$</td>
</tr>
<tr>
<td>$\min(x)$</td>
<td>The minimum element in vector $x$</td>
</tr>
<tr>
<td>$\text{ceiling}(x)$</td>
<td>The smallest integer that is greater than or equal to $x$</td>
</tr>
<tr>
<td>$\text{floor}(x)$</td>
<td>The largest integer that is not greater than $x$</td>
</tr>
<tr>
<td>$\text{int}(x)$</td>
<td>The largest integer that is less than or equal to $x$</td>
</tr>
<tr>
<td>$x \mod(m)$</td>
<td>$x$ modulo $m$; the remainder when $x$ is divided by $m$</td>
</tr>
<tr>
<td>$\infty$</td>
<td>Infinity</td>
</tr>
<tr>
<td>$\pi$</td>
<td>Pi, 3.14159265...</td>
</tr>
</tbody>
</table>

How to Interpret the Subroutine Descriptions

This explains how to interpret the information in the subroutine descriptions.

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.

For subroutines with CBLAS and LAPACKE calling sequences, the Data Types table lists only the Fortran name. The data types used for the CBLAS and LAPACKE are the same as that used for Fortran.

Syntax

This section shows the syntax for subroutines as follows:

- [Table 3 on page xxvii](#) shows the syntax for the Fortran, C and C++ calling sequences. All subroutines will include this information.
- [Table 4 on page xxvii](#) shows the syntax for the CBLAS calling sequences, which will appear for BLAS subroutines in addition to the Fortran, C, and C++ calling sequences.
- [Table 5 on page xxvii](#) shows the syntax for the LAPACKE calling sequences, which will appear for LAPACK subroutines in addition to the Fortran, C, and C++ calling sequences.

Note:

- For information about the CBLAS calling sequence, see [10 on page 1364](#).
- For a list of BLAS, see Appendix A, “Basic Linear Algebra Subprograms (BLAS) and Complex BLAS (CBLAS),” on page 1339.
For information about the LAPACKE calling sequence see the following URL:
http://www.netlib.org/lapack/LAPACKE.html

Table 3. Syntax for all subroutines

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL NAME-1</th>
<th>NAME-2</th>
<th>...</th>
<th>NAME-n (arg-1, arg-2, ..., arg-m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>name-1</td>
<td>name-2</td>
<td>...</td>
<td>name-n (arg-1, arg-2, ..., arg-m);</td>
</tr>
</tbody>
</table>

Table 4. Syntax for CBLAS subroutines

| CBLAS        | cblas_name-1 | cblas_name-2 | ... | cblas_name-n (arg-1, arg-2, ..., arg-m); |

Table 5. Syntax for LAPACKE subroutines

| LAPACKE      | LAPACKE_name-1 | LAPACKE_name-2 | ... | LAPACKE_name-n (arg-1, arg-2, ..., arg-m); |

The syntax indicates:
- Each possible subroutine or subprogram 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 “On Entry” and “On Return”, respectively. An argument used for both input and output is described in both “On Entry” and “On Return”. 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.
- Names beginning with cblas_ indicate that you specify enumerated types. These are used only for CBLAS.

Note:
- If you code a CBLAS calling sequence, there are times when an argument description references a character argument. In that situation, you should translate the character argument to its equivalent CBLAS enumerated type, as shown in Table 6 on page xxviii.
- The CBLAS_ORDER argument has been renamed to CBLAS_LAYOUT. See “Migrating Programs from ESSL for Linux on Power Version 5 Release 4 to Version 5 Release 5” on page 201 for the required changes to your existing programs.
- Although argument CBLAS_ORDER is still supported, IBM recommends using equivalent argument CBLAS_LAYOUT instead.
Table 6. Translating character argument values to CBLAS enumerated types

<table>
<thead>
<tr>
<th>Character argument</th>
<th>Character Argument Value</th>
<th>CBLAS Argument</th>
<th>Enumerated Type Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>trans, transa, transb</td>
<td>'N'</td>
<td>cblas_trans</td>
<td>CblasNoTrans</td>
</tr>
<tr>
<td></td>
<td>'T'</td>
<td>cblas_transa</td>
<td>CblasTrans</td>
</tr>
<tr>
<td></td>
<td>'C'</td>
<td>cblas_transb</td>
<td>CblasConjTrans</td>
</tr>
<tr>
<td>side</td>
<td>'L'</td>
<td>cblas_side</td>
<td>CblasLeft</td>
</tr>
<tr>
<td></td>
<td>'R'</td>
<td></td>
<td>CblasRight</td>
</tr>
<tr>
<td>uplo</td>
<td>'U'</td>
<td>cblas_uplo</td>
<td>CblasUpper</td>
</tr>
<tr>
<td></td>
<td>'L'</td>
<td></td>
<td>CblasLower</td>
</tr>
<tr>
<td>diag</td>
<td>'U'</td>
<td>cblas_diag</td>
<td>CblasUnit</td>
</tr>
<tr>
<td></td>
<td>'N'</td>
<td></td>
<td>CblasNonUnit</td>
</tr>
</tbody>
</table>

**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.

The calling sequences for the Level 2 CBLAS and the Level 3 CBLAS include input arguments that are enumerated types defined in essl.h. Argument cblas_layout indicates whether the input and output matrices are stored in column-major order or row-major order. All other enumerated type arguments replace the character arguments found in the Fortran, C and C++ calling sequences (see Table 6). Unlike the C and C++ interfaces to ESSL, complex scalar arguments are passed by reference instead of being passed by value.

**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.

**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 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.
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 using 32-bit integers. If you are using 64-bit integers, you may need to use a larger workspace and therefore you may need to increase the size of naux and lwork. (See “Setting Up Auxiliary Storage When Dynamic Allocation Is Not Used” on page 53.)

The examples provided for each subroutine 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.

How to Send Your Comments

Your feedback is important in helping us to produce accurate, high-quality information. If you have any comments about this information or any other ESSL documentation, send your comments to the following e-mail address:

mhrvrcfs@us.ibm.com

Include the publication title and order number, and, if applicable, the specific location of the information about which you have comments (for example, a page number or a table number).
Summary of Changes

The following sections summarize changes to ESSL and the ESSL documentation for each new release or major service update for a given product version. Within each book in the library, a vertical line to the left of text and illustrations indicates technical changes or additions made to the previous edition of the book.

Summary of changes
for ESSL for AIX, Version 5 Release 3
and ESSL for Linux on POWER®, Version 5 Release 5
as updated, December 2016

ESSL 5.5 now supports the following:

• IBM Power System S822LC (8335-GTB) servers with NVIDIA P100 GPUs and IBM Power System S822LC (8335-GTA) servers with NVIDIA K80 GPUs running Red Hat Enterprise Linux 7.3 (RHEL7.3) (little endian mode).

Note: The ESSL SMP CUDA Library is only supported on these models.

• IBM Power8 Servers running RHEL 7.2 or RHEL 7.3 (little endian mode).

• New BLAS subroutines:
  – “SROTMG and DROTMG (Construct a modified Givens Transformation)” on page 284
  – “SROTM and DROTM (Apply a modified Givens Transformation)” on page 287

• Multithreaded Versions of Sparse Matrix Vector Product subroutines. see:
  – “DSMMX (Matrix-Vector Product for a Sparse Matrix in Compressed-Matrix Storage Mode)” on page 418
  – “DSDMX (Matrix-Vector Product for a Sparse Matrix or Its Transpose in Compressed-Diagonal Storage Mode)” on page 425

• New LAPACK subroutines:
  – SGESDD, DGESDD, CGESDD, and ZGESDD; See “SGESVD, DGESVD, CGESVD, ZGESVD, SGESDD, DGESDD, CGESDD, and ZGESDD (Singular Value Decomposition for a General Matrix)” on page 881
  – SGEEV, DGEEV, CGEEV and ZGEEV; See “SGEEV, DGEEV, CGEEV, ZGEEV, SGEEVX, DGEEVX, CGEEVX, and ZGEEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix)” on page 942
  – SSYEV, DSYEV, CHEEV, and ZHEEV; See “SSYEV, DSYEV, CHEEV, ZHEEV, SSPEVX, DSPEVX, CHPEVX, ZHPEVX, SSYEVX, DSYEVX, CHEEVX, and ZHEEVX (Eigenvalues and, Optionally, the Eigenvectors of a Real Symmetric or Complex Hermitian Matrix)” on page 959
  – SGEVX, DGGEVX, CGGEVX, and ZGGEVX; See “SGEVX, DGGEVX, CGGEVX, ZGEVX, SGEVX, DGGEVX, CGGEVX, and ZGGEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix Generalized Eigenproblem)” on page 991
• Additional CBLAS (C Interface to the BLAS) Subprograms. See Appendix A, “Basic Linear Algebra Subprograms (BLAS) and Complex BLAS (CBLAS),” on page 1339.

• LAPACKE (C Interface to LAPACK) Subroutines corresponding to the ESSL LAPACK subroutines. See Appendix B, “LAPACK and LAPACKE,” on page 1343.

• Netlib JAVA and python. See Appendix D, “Using ESSL with netlib-java and Python,” on page 1353.

Summary of changes
for ESSL for AIX, Version 5 Release 3
and ESSL for Linux on POWER, Version 5 Release 4
as updated, December 2015

ESSL 5.4 now supports the following:
• IBM Power System S822LC (8335-GT A) servers with NVIDIA K80 GPUs running Red Hat Enterprise Linux 7.2 (RHEL7.2) or later (little endian mode).

  Note: The ESSL SMP CUDA library is only supported on this model.

• Power8 Servers running RHEL 7.2 or later (little endian mode).

• CBLAS, a C Interface to the Basic Linear Algebra Subprogrms (BLAS).

• Compiling ESSL C++ applications using the g++ compiler.

ESSL 5.4 does not support the following:
• Ubuntu (little endian mode)
• SUSE Linux Enterprise Server 12 (SLES12) (little endian mode)
• RHEL7 (big endian mode)
• IBM Power System S824L server Model 42L with NVIDIA K40 GPUs
• IBM Power7+ and Power7 servers and blades.

If you require any of the above support, order ESSL for Linux V5.3.2 instead.

Summary of changes
for ESSL for AIX, Version 5 Release 3
and ESSL for Linux on POWER, Version 5 Release 3.2
as updated, July 2015

ESSL 5.3.2 provides new support for the ESSL SMP CUDA 32-bit integer/64-bit pointer environment library. The ESSL SMP CUDA Library is supported only on IBM Power® System S824L server (8247-42L) with one or two NVIDIA Tesla K40 GPUs running Ubuntu 14.04.2 or Ubuntu 14.10.

You can use the ESSL SMP CUDA Library in two ways for the subset of ESSL Subroutines that are GPU-enabled:
• Using NVIDIA GPUs for the bulk of the computation.
• Using a hybrid combination of POWERs® CPUs and NVIDIA GPUs.

The ESSL SMP CUDA library leverages ESSL BLAS and NVIDIA cuBLAS and blocking techniques to handle problem sizes larger than the GPU memory size. The algorithms support multiple GPUs and are designed for use in both SMP and MPI applications.

For information, see “Using the ESSL SMP CUDA Library” on page 43.
Subroutines

The following new SETGPUS utility subroutine is now included; See “SETGPUS (Set the Number of GPUs and Identify Which GPUs ESSL Should Use)” on page 1305.

Summary of changes
for ESSL for AIX, Version 5 Release 3
and ESSL for Linux on POWER, Version 5 Release 3.1
as updated, December 2014

ESSL 5.3.1 (little endian mode) provides the following new support:
• 64-bit applications running on Power8 servers in little endian mode
• C99 complex floating point types for complex arithmetic when the ESSL header file is used to call ESSL from C and C++ applications

Operating systems

Support has been added for the following operating systems:
• SUSE Linux Enterprise Server 12 (SLES12)
• Ubuntu Server 14.04.01 for IBM Power
• Ubuntu Server 14.10 for IBM Power

For a complete list of operating system versions and distributions on which this release of ESSL is supported, see "Operating Systems Supported by ESSL” on page 8.

Summary of changes
for ESSL for AIX, Version 5 Release 3
and ESSL for Linux on POWER, Version 5 Release 3
as updated, August 2014

This release of ESSL provides the changes described below.

Operating systems

Support has been added for the following operating systems:
• Red Hat Linux Enterprise Server 7 (RHEL7)

Support is no longer provided for the following operating systems:
• Red Hat Linux Enterprise Server 6 (RHEL6)
• SUSE Linux Enterprise Server 11 SP1 (SLES11 SP1)

For a complete list of operating system versions and distributions on which this release of ESSL is supported, see "Operating Systems Supported by ESSL” on page 8.

Servers and processors

This document has been updated to include support for the IBM Power8 processors.

For a complete list of servers and processors on which this release of ESSL is supported, see “Hardware Products Supported by ESSL” on page 8.

Subroutines

The following new subroutines are now included:

Dense Linear Algebraic Equation Subroutines:
• SSYSV, DSYSV, CSYSV, ZSYSV, CHESV, ZHESV, SSPSV, DSPSV, CPSV, ZSPSV, CHPSV, and ZHPSV; See "SSYSV, DSYSV, CSYSV, ZSYSV, ..."
CHESV, ZHESV, SSPSV, DSPSV, CSPSV, ZSPSV, CHPSV, and ZHPSV (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)” on page 642.

- SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, ZHETRF, SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF, and ZHPTRF; See “SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, ZHETRF, SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF, and ZHPTRF (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Factorization)” on page 651.

- SSYTRS, DSYTRS, CSYTRS, ZSYTRS, CHETRS, ZHETRS, SSPTRS, DSPTRS, CSPTRS, ZSPTRS, CHPTRS, and ZHPTRS; See “SSYTRS, DSYTRS, CSYTRS, ZSYTRS, CHETRS, ZHETRS, SSPTRS, DSPTRS, CSPTRS, ZSPTRS, CHPTRS, and ZHPTRS (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)” on page 660.

- SLANTR, DLANTR, CLANTR, ZLANTR, SLANTP, DLANTP, CLANTP, and ZLANTP; See “SLANTR, DLANTR, CLANTR, ZLANTR, SLANTP, DLANTP, CLANTP, and ZLANTP (Trapezoidal or Triangular Matrix Norm)” on page 690.

Banded Linear Algebraic Equation Subroutines:

- SGBSV, DGBSV, CGBSV, and ZGBSV; See “SGBSV, DGBSV, CGBSV, and ZGBSV (General Band Matrix Factorization and Multiple Right-Hand Side Solve)” on page 698.

- SGBTTRF, DGBTTRF, CGBTTRF, and ZGBTTRF; See “SGBTTRF, DGBTTRF, CGBTTRF and ZGBTTRF (General Band Matrix Factorization)” on page 702.

- SGBTTRS, DGBTTRS, CGBTTRS, and ZGBTTRS; See “SGBTTRS, DGBTTRS, CGBTTRS, and ZGBTTRS (General Band Matrix Multiple Right-Hand Side Solve)” on page 706.

- SGTSV, DGTSV, CGTSV, and ZGTSV; See “SGTSV, DGTSV, CGTSV, and ZGTSV (General Tridiagonal Matrix Factorization and Multiple Right-Hand Side Solve)” on page 731.

- SGTTTRF, DGTTTRF, CGTTTRF, and ZGTTTRF; See “SGTTTRF, DGTTTRF, CGTTTRF, and ZGTTTRF (General Tridiagonal Matrix Factorization)” on page 735.

- SGTTTRS, DGTTTRS, CGTTTRS, and ZGTTTRS; See “SGTTTRS, DGTTTRS, CGTTTRS, and ZGTTTRS (General Tridiagonal Matrix Multiple Right-Hand Side Solve)” on page 740.

Linear Least Squares Subroutines:

- SGESVD, DGESVD, CGESVD, and ZGESVD; See “SGESVD, DGESVD, CGESVD, ZGESVD, DGESDD, CGESDD, ZGESDD, and ZGESDD (Singular Value Decomposition for a General Matrix)” on page 881.

- SGELSD, DGELSD, CGELSD, and ZGELSD; See “SGELSD, DGELSD, CGELSD, and ZGELSD (Linear Least Squares Solution for a General Matrix Using the Singular Value Decomposition)” on page 911.

Random Number Generation Subroutines:

- INITRNG; See “INITRNG (Initialize Random Number Generators)” on page 1271.

- SURNG and DURNG; See “SURNG and DURNG (Generate a Vector of Uniformly Distributed Pseudo-Random Numbers)” on page 1276.

- SNRNG and DNRNG; See “SNRNG and DNRNG (Generate a Vector of Normally Distributed Pseudo-Random numbers)” on page 1279.
Summary of changes
for ESSL for AIX, Version 5 Release 2
and ESSL for Linux on POWER, Version 5 Release 2
as updated, February 2013

This release of ESSL provides the changes described below.

Operating systems
Support is no longer provided for the following operating systems:
- AIX 5.3

For a complete list of operating system versions and distributions on
which this release of ESSL is supported, see “Operating Systems Supported
by ESSL” on page 8.

Servers and processors
This document has been updated to include support for the IBM
POWER7® processors. This support was added to ESSL after the July 2012
publication of this document.

Support is no longer provided for the following servers and processors:
- IBM BlueGene/Q

For a complete list of servers and processors on which this release of ESSL
is supported, see “Hardware Products Supported by ESSL” on page 8.

Subroutines
The following new subroutines are now included:

Matrix Operations:
- CGECMI and ZGECMI; See “SGETMI, DGETMI, CGETMI, ZGETMI,
  CGECMI and ZGECMI (General Matrix Transpose or Conjugate
  Transpose [In-Place])” on page 509.
- CGECMO and ZGECMO; See “SGETMO, DGETMO, CGETMO,
  ZGETMO, CGECMO, and ZGECMO (General Matrix Transpose or
  Conjugate Transpose [Out-of-Place])” on page 512.

Banded Linear Algebraic Equation Subroutines:
- SPBSV, DPBSV, CPBSV, and ZPBSV; See “SPBSV, DPBSV, CPBSV,
  and ZPBSV (Positive Definite Real Symmetric or Complex Hermitian Band
  Matrix Factorization and Multiple Right-Hand Side Solve)” on page 715.
- SPBTRF, DPBTRF, CPBTRF, and ZPBTRF; See “SPBTRF, DPBTRF,
  CPBTRF, and ZPBTRF (Positive Definite Real Symmetric or Complex
  Hermitian Band Matrix Factorization)” on page 721.
- SPBTRS, DPBTRS, CPBTRS, and ZPBTRS; See “SPBTRS, DPBTRS,
  CPBTRS, and ZPBTRS (Positive Definite Real Symmetric or Complex
  Hermitian Band Matrix Multiple Right-Hand Side Solve)” on page 726.
- SPTSV, DPTSV, CPTSV, and ZPTSV; See “SPTSV, DPTSV, CPTSV,
  and ZPTSV (Positive Definite Real Symmetric or Complex Hermitian
  Tridiagonal Matrix Factorization and Multiple Right-Hand Side Solve)” on page 746.
- SPTTRF, DPTTRF, CPTTRF, and ZPTTRF; See “SPTTRF, DPTTRF,
  CPTTRF, and ZPTTRF (Positive Definite Real Symmetric or Complex
  Hermitian Tridiagonal Matrix Factorization)” on page 751.
- SPTTRS, DPTTRS, CPTTRS, and ZPTTRS; See “SPTTRS, DPTTRS,
  CPTTRS, and ZPTTRS (Positive Definite Real Symmetric or Complex
  Hermitian Tridiagonal Matrix Multiple Right-Hand Solve)” on page 755.
Eigensystem Analysis Subroutines:

- SSPEVD, DSPEVD, CHPEVD, ZHPEVD, SSYEVD, DSYEVD, CHEEVD, and ZHEEVD; See “SSPEVD, DSPEVD, CHPEVD, ZHPEVD, SSYEVD, DSYEVD, CHEEVD, and ZHEEVD (Eigenvalues and, Optionally the Eigenvectors, of a Real Symmetric or Complex Hermitian Matrix Using a Divide-and-Conquer Algorithm)” on page 978.

- SGGEV, DGGEV, CGGEV, and ZGGEV; See “SGGEV, DGGEV, CGGEV, ZGGEV, SGGEVX, DGGEVX, CGGEVX, and ZGGEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix Generalized Eigenproblem)” on page 991.

- SSPGVX, DSPGVX, CHPGVX, ZHPGVX, SSYGVX, CHEGVX, and ZHEGVX; See “SSPGVX, DSPGVX, CHPGVX, ZHPGVX, SSYGVX, DSYGVX, CHEGVX, and ZHEGVX (Eigenvalues and, Optionally, the Eigenvectors of a Positive Definite Real Symmetric or Complex Hermitian Generalized Eigenproblem)” on page 1008.

The ESSL 5.1 non-LAPACK-conforming subroutines, that is, those subroutines whose name is the same as an existing LAPACK subroutine, but whose calling-sequence arguments and functionality are different from that LAPACK subroutine have been removed from ESSL 5.2. For details, see “Migrating Programs from ESSL for AIX 5.1 and ESSL for Linux on Power Version 5 Release 1.1 to Version 5 Release 2” on page 202.

Summary of changes for ESSL for AIX, Version 5 Release 1 and ESSL for Linux on POWER, Version 5 Release 1.1 as updated, July 2012

This release of ESSL for Linux on POWER provides the following new libraries:
- ESSL Blue Gene® Serial Library and ESSL Blue Gene SMP Library, which provide versions of the ESSL subroutines for use on Blue Gene®/Q and run in a 32-bit integer, 64-bit pointer environment on RHEL6.2.
  These libraries can also be used with the FFTW Wrappers Support.

Support has been added for the following compiler levels:
- IBM XL Fortran for AIX 14.1 and IBM XL C/C++ for AIX 12.1
- IBM XL Fortran for Linux 14.1 and IBM XL C/C++ for Linux 12.1

This document has also been updated to include support for RHEL6 for Power platforms. This support was added to ESSL 5.1 after the October 2010 publication of this document.

Summary of changes for ESSL for AIX, Version 5 Release 1 and ESSL for Linux on POWER, Version 5 Release 1 as updated, October 2010

The ESSL 5.1 Serial Library and the ESSL SMP Library contain:
- A VSX (SIMD) version of selected subroutines for use on POWER7 processor-based servers
- An Altivec (SIMD) version of selected subroutines for use on POWER6® processor-based servers

This release of ESSL provides the changes described below.
Operating systems

Support has been added for the following operating system version:
- AIX 7.1

Support is no longer provided for the following operating systems:
- SUSE Linux Enterprise Server 10 for POWER (SLES10)
- Red Hat Enterprise Linux 5 (RHEL5)

For a complete list of operating system versions and distributions on which this release of ESSL is supported, see “Operating Systems Supported by ESSL” on page 8.

Servers and processors

Support has been added for the POWER7 processor.

Support is no longer provided for the following servers and processors:
- IBM BladeCenter JS21, IBM POWERPC 450, IBM POWERPC 450D, IBM POWER5, IBM POWER5+, IBM POWERPC970 processors, IBM Blue Gene®/P.

For a complete list of servers and processors on which this release of ESSL is supported, see “Hardware Products Supported by ESSL” on page 8.

Subroutines

ESSL 5.1 is the last release to support non-LAPACK-conforming-subroutines; that is, those ESSL subroutines whose name is the same as an existing LAPACK subroutine, but whose calling-sequence arguments and functionality are different from that LAPACK subroutine.

This new LAPACK subroutine is now included:
- DSYGVX. See “SSPGVX, DSPGVX, CHPGVX, ZHPGVX, SSYGVX, DSYGVX, CHEGVX, and ZHEGVX (Eigenvalues and, Optionally, the Eigenvectors of a Positive Definite Real Symmetric or Complex Hermitian Generalized Eigenproblem)” on page 1008

These new Fourier Transform subroutines are now included:
- SRCFTD and DRCFTD. See “SRCFTD and DRCFTD (Multidimensional Real-to-Complex Fourier Transform)” on page 1044
- SCRFTD and DCRFTD. See “SRCFTD and DCRFTD (Multidimensional Complex-to-Real Fourier Transform)” on page 1052

FFTW Wrappers

Support has been added to the ESSL FFTW Wrapper Libraries corresponding to the new ESSL Fourier Transform subroutines. See Appendix C, “FFTW Version 3.1.2 to ESSL Wrapper Libraries,” on page 1349 for the list of FFTW subroutines supported, restrictions on their use, and instructions on how to build, install, and use the ESSL FFTW Wrappers Library.

Documentation for FFTW Version 3.1.2 can be found at:
http://www.fftw.org

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 205, which explains what you can do now to prevent future migration problems.
Part 1. Guide Information

The following types of guidance information about how to use ESSL are available:

• Learning how to use ESSL documentation
• Learning what is new in ESSL
• Learning about the ESSL product
• Designing your program
• Setting up your data structures
• Coding your program
• Processing your program
• Migrating your programs
• Handling problems
Chapter 1. Introduction and Requirements

This introduces you to the Engineering and Scientific Subroutine Library (ESSL) product.

Overview of ESSL

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

ESSL is provided as run-time libraries that run on the servers and processors listed in “Hardware Products Supported by ESSL” on page 8.

ESSL can be used with Fortran, C, and C++ programs operating under the AIX and Linux operating systems.

To order ESSL, specify one of the program numbers below:

ESSL for AIX
5765-H25

ESSL for Linux
5765-L51

Performance and Functional Capability

The mathematical subroutines, in nine computational areas, are tuned for performance. 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 runs under the AIX and Linux operating systems.

ESSL provides the following run-time libraries (described in detail in “What ESSL Library Do You Want to Use?” on page 31):

- ESSL Serial Libraries and ESSL SMP Libraries, which run in the following environments:
  - 32-bit integer, 32-bit pointer environment (AIX only)
  - 32-bit integer, 64-bit pointer environment
  - 64-bit integer, 64-bit pointer environment
- ESSL SMP CUDA Library which runs in the following environment
  - 32-bit integer, 64-bit pointer environment (little endian only)

Notes:
For the 32-bit integer, 64-bit pointer environment, in accordance with the LP64 data model, all ESSL integer arguments remain 32 bits except for the iusadr argument for ERRSET.

To avoid 32-bit integer overflow problems (for example, matrices of order \( n \) where \( N > 46340 \), use the ESSL 64-bit integer, 64-bit pointer environment libraries.

These libraries contain:
- a VSX (SIMD) version of selected subroutines for use on VSX enabled processor-based servers.
- an AltiVec version of selected subroutines for use on POWER6 processors (AIX only).

These ESSL libraries are described in detail in “What ESSL Library Do You Want to Use?” on page 31.

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

The ESSL subroutines can be called from application programs written in Fortran, C, and C++.

Usability

ESSL is designed for usability:
- It has an easy-to-use call interface.
- If your existing application programs use the Serial Libraries, you only need to re-link your program to take advantage of the increased performance of the SMP Libraries.
- It has informative error-handling capabilities, enabling you to calculate auxiliary storage sizes and transform lengths.
- Online documentation that can be displayed using a Hypertext Markup Language (HTML) document browser is available for use with ESSL.

The Variety of Mathematical Functions

ESSL includes several different types of mathematical functions.

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
ESSL provides run-time libraries that are designed to provide high levels of performance for numerically intensive computing jobs.

The subroutines provided in ESSL, summarized in Table 7, fall into the following groups:
• Nine major areas of mathematical computation, providing the computations commonly used by the industry applications listed
• Utilities, performing general-purpose functions

Most of the subroutine calls are compatible with those in the ESSL/370 product.

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 31.

Table 7. Summary of ESSL Subroutines

<table>
<thead>
<tr>
<th>ESSL Area of Computation</th>
<th>Integer Subroutines</th>
<th>Short-Precision Subroutines</th>
<th>Long-Precision Subroutines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Algebra Subprograms:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector-scalar</td>
<td>0</td>
<td>43</td>
<td>43</td>
</tr>
<tr>
<td>Sparse vector-scalar</td>
<td>0</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>Matrix-vector</td>
<td>0</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td>Sparse matrix-vector</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Matrix Operations:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addition, subtraction, multiplications, triangular solves, rank-k updates, rank-2k updates, and matrix transposes</td>
<td>0</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>Linear Algebraic Equations:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dense linear algebraic equations</td>
<td>0</td>
<td>82</td>
<td>87</td>
</tr>
<tr>
<td>Banded linear algebraic equations</td>
<td>0</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>Sparse linear algebraic equations</td>
<td>0</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>Linear least squares</td>
<td>0</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>Eigensystem Analysis:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Solutions to the algebraic eigensystem analysis problem and the generalized eigensystem analysis problem</td>
<td>0</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Signal Processing Computations:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fourier transforms</td>
<td>0</td>
<td>18</td>
<td>14</td>
</tr>
<tr>
<td>Convolutions and correlations</td>
<td>0</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>Related computations</td>
<td>0</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Sorting and Searching:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorting, sorting with index, and binary and sequential searching</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Interpolation:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Polynomial and cubic spline interpolation</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Numerical Quadrature:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Numerical quadrature on a set of points or on a function</td>
<td>0</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>
Table 7. Summary of ESSL Subroutines (continued)

<table>
<thead>
<tr>
<th>ESSL Area of Computation</th>
<th>Integer Subroutines</th>
<th>Short-Precision Subroutines</th>
<th>Long-Precision Subroutines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Number Generation: Generating vectors of uniformly distributed and normally distributed random numbers</td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Utilities: General service operations</td>
<td>9</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Total ESSL Subroutines</td>
<td>15</td>
<td>330</td>
<td>341</td>
</tr>
</tbody>
</table>

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 and long-precision versions of the subroutines are ANSI/IEEE 32-bit and 64-bit binary floating-point format. 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 64.)

For more information on accuracy, see “Getting the Best Accuracy” on page 63.

High Performance of ESSL
The ESSL subroutines have been designed to provide high performance. (See references [38 on page 1365], [49 on page 1366], and [50 on page 1366].)

Algorithms
To achieve high 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.
• Structuring the ESSL subroutines so, where applicable, the compiled code fully utilizes the dual 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 takes full advantage of the hardware data prefetching.

Obtaining High Performance
Obtaining high performance depends on the type of processor you are using.
Obtaining High Performance on SMP processors with NVIDIA GPUs:  The ESSL SMP CUDA Library is designed to exploit the processing power of the NVIDIA GPUs and of the Power8 CPUs for a subset of the ESSL subroutines. For a list of these subroutines, see "Using the ESSL SMP CUDA Library" on page 43.

Obtaining High Performance on SMP Processors:  The ESSL SMP Libraries and the ESSL SMP CUDA Library are 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 Libraries, see Table 40 on page 39.

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 either the XL Fortran XLSMPOPTS or the OMP_NUM_THREADS environment variable to specify the number of threads you want to create.

Obtaining High Performance on VSX-Enabled Processors:  The ESSL Serial Libraries, the ESSL SMP Libraries, and the ESSL SMP CUDA Library are designed to exploit the processing power of VSX-enabled processors. For details about how to use it to achieve optimal performance, see "SIMD Algorithms on VSX-Enabled Processors" on page 32.

Obtaining High Performance on AltiVec-Enabled Processors:  The ESSL Serial Libraries and the ESSL SMP Libraries are designed to exploit the processing power of the AltiVec unit on certain PowerPC® processors. For details about how to use it to achieve optimal performance, see "SIMD Algorithms on POWER 6 AltiVec-Enabled Processors" on page 33.

SMT Mode

SMT is a processor technology that allows multiple instruction streams (threads) to run concurrently on the same physical processor, improving overall throughput. To the operating system, each hardware thread is treated as an independent logical processor.

Not all applications benefit from SMT. Having multiple threads executing on the same processor will not increase the performance of applications with execution-unit–limited performance or applications that consume all the chip’s memory bandwidth. For this reason, these processors support single-threaded (ST) execution mode. In this mode, these processors give all the physical resources to the active thread.

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 or C++, 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, “Coding Your Program,” on page 133.

Software and Hardware Products That Can Be Used with ESSL

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

- “Hardware Products Supported by ESSL”
- “Operating Systems Supported by ESSL”
- “Software Products Required by ESSL” on page 9
- “Software Products for Installing and Customizing ESSL” on page 10
- “Software Products for Displaying ESSL Documentation” on page 10

Hardware Products Supported by ESSL

ESSL for AIX runs on the following hardware platforms:
- IBM POWER8 servers
- IBM POWER7+™ and POWER7 servers and blades
- IBM POWER6+™ and POWER6 servers and blades

ESSL for Linux on POWER is supported on the following hardware platforms running in little endian mode:
- IBM POWER8 servers

Note: The ESSL SMP CUDA Library is supported only on IBM Power System S822LC (8335-GTB) servers with NVIDIA P100 GPUs and IBM Power System S822LC (8335-GTA) servers with NVIDIA K80 GPUs running Red Hat Enterprise Linux 7.3 (RHEL7.3) (little endian mode).

Operating Systems Supported by ESSL

ESSL is supported in the following operating system environments:

<table>
<thead>
<tr>
<th>Product</th>
<th>Supported Environment big endian mode</th>
<th>Supported Environment little endian mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESSL for AIX</td>
<td>• AIX 7.1 with the latest available Technology Level</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>• AIX 6.1 with the latest available Technology Level</td>
<td></td>
</tr>
</tbody>
</table>
Table 8. Operating systems supported by ESSL (continued)

<table>
<thead>
<tr>
<th>Product</th>
<th>Supported Environment</th>
<th>Supported Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>big endian mode</td>
<td>little endian mode</td>
</tr>
<tr>
<td>ESSL for Linux on POWER</td>
<td>N/A</td>
<td>Red Hat Enterprise Linux 7.3 (RHEL7.3) (little endian mode) for IBM Power System S822LC (8335-GTB) servers with NVIDIA P100 GPUs and IBM Power System S822LC (8335-GTA) servers with NVIDIA K80 GPUs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Red Hat Enterprise Linux 7.2 (RHEL7.2) or RHEL7.3 (little endian mode) for other Power8 Servers.</td>
</tr>
</tbody>
</table>

Software Products Required by ESSL

This describes the software products that are required by ESSL.

- “Software Products Required by ESSL for AIX”
- “Software Products Required by ESSL for Linux”

Software Products Required by ESSL for AIX

ESSL for AIX requires the software products shown in “Required Software Products on AIX” 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 151 and “C++ Programs” on page 166.

Required Software Products on AIX:
The following table lists the required software products for ESSL for AIX:

Table 9. Required Software Products for ESSL for AIX

<table>
<thead>
<tr>
<th>Required Software Products</th>
<th>Supported Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>For Compiling</td>
<td></td>
</tr>
<tr>
<td>IBM XL Fortran for AIX</td>
<td>15.1 or later with the latest service</td>
</tr>
<tr>
<td>IBM XL C/C++ for AIX</td>
<td>13.1 or later with the latest service</td>
</tr>
<tr>
<td>For Linking, Loading, or Running (See Note 1)</td>
<td></td>
</tr>
<tr>
<td>IBM XL Fortran Runtime Environment for AIX</td>
<td>15.1 or later with the latest service</td>
</tr>
<tr>
<td>(See Note 2)</td>
<td></td>
</tr>
<tr>
<td>IBM XL C libraries</td>
<td>(See Note 3)</td>
</tr>
</tbody>
</table>

Notes:
1. Optional filesets are required for building applications. For details, consult the AIX and compiler documentation.
2. The correct version of IBM XL Fortran Runtime Environment for AIX is automatically shipped with the compiler. It is also available for downloading from the following website:
   http://www.ibm.com/support/docview.wss?rs=43&uid=swg21156900
3. The AIX product includes the C and math libraries in the Application Development Toolkit.

Software Products Required by ESSL for Linux

ESSL for Linux requires the software products listed in “Required Software Products on Linux” on page 10 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 151 and “C++ Programs” on page 166.
## Required Software Products on Linux:
The following table lists the required software products for ESSL for Linux on POWER:

<table>
<thead>
<tr>
<th>Table 10. Required Software Products for ESSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required software products</td>
</tr>
<tr>
<td>For Compiling</td>
</tr>
<tr>
<td>IBM XL Fortran for Linux</td>
</tr>
<tr>
<td>IBM XL C/C++ for Linux</td>
</tr>
<tr>
<td>gcc and g++</td>
</tr>
<tr>
<td>For Linking, Loading, or Running</td>
</tr>
<tr>
<td>IBM XL Fortran Runtime Environment for Linux</td>
</tr>
<tr>
<td>(See Note 1)</td>
</tr>
<tr>
<td>gcc and g++ 64-bit libraries</td>
</tr>
<tr>
<td>CUDA Toolkit</td>
</tr>
<tr>
<td>(See Note 4)</td>
</tr>
</tbody>
</table>

**Notes:**
1. Additional software packages may be required for building applications. For details, consult the Linux and compiler documentation.
2. The correct version of IBM XL Fortran Runtime Environment and Addons Library for Linux is automatically shipped with the compiler. It is also available for downloading from the following website: http://www.ibm.com/support/docview.wss?rs=43&uid=swg21156900
3. Use the compilers and libraries provided with your Linux distribution. The ESSL SMP libraries require the XL OpenMP runtime. The gcc OpenMP runtime is not compatible with the XL OpenMP runtime. Therefore, the ESSL SMP libraries can only be used with other compilers if the program calling ESSL is a serial program (does not use OpenMP) because in this case only the XL OpenMP runtime is used.
4. This product is only required in order to use the ESSL SMP CUDA library. The ESSL SMP CUDA Library is only supported on IBM Power System S822LC (8335-GTB) servers with NVIDIA P100 GPUs and IBM Power System S822LC (8335-GTA) servers with NVIDIA K80 GPUs running Red Hat Enterprise Linux 7.3 (RHEL7.3) (little endian mode).

---

### Software Products for Installing and Customizing ESSL

The ESSL licensed program is distributed on a CD. Different software products are required for installing and customizing ESSL on AIX or on Linux.

- "Software Products for Installing and Customizing ESSL for AIX"
- "Software Products for Installing and Customizing ESSL for Linux"

### Software Products for Installing and Customizing ESSL for AIX

The ESSL for AIX Installation Guide provides the detailed information you need to install ESSL for AIX.

### Software Products for Installing and Customizing ESSL for Linux

The ESSL for Linux Installation Guide provides the detailed information you need to install ESSL for Linux.

### Software Products for Displaying ESSL Documentation

The software products needed to display ESSL online information are listed in Table 11 on page 11.
Table 1. Software needed to display various formats of ESSL online information

<table>
<thead>
<tr>
<th>Format of online information</th>
<th>Software needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML</td>
<td>HTML document browser (such as Microsoft Internet Explorer)</td>
</tr>
<tr>
<td>PDF</td>
<td>Adobe Acrobat Reader, which is freely available for downloading from the Adobe Web site at: <a href="http://www.adobe.com">http://www.adobe.com</a></td>
</tr>
</tbody>
</table>
| Manpages                     | No additional software needed.  
Note: In order for manpages to be displayed properly on Linux, the LANG environment variable must be set to either of the following values: C or en_US.iso885915.  
To display a specific manpage, use the man command as follows:  
```
man subroutine-name
```

Note: These manpages will be installed in the following directory: `/usr/share/man/man3`

The manpages provided by LAPACK are installed in the `/usr/share/man/manl` directory. By default, ESSL manpages will be displayed rather than BLAS or LAPACK manpages with the same names. If you want to access the BLAS or LAPACK manpages, you must set the MANPATH environment variable. See the documentation for the man command.

List of ESSL Subroutines

ESSL provides several different types of subroutines.

Appendix A, “Basic Linear Algebra Subprograms (BLAS) and Complex BLAS (CBLAS),” on page 1339 contains a list of Level 1, 2, and 3 Basic Linear Algebra Subprograms (BLAS) included in ESSL.

Appendix B, “LAPACK and LAPACKE,” on page 1343 contains a list of Linear Algebra Package (LAPACK) subroutines included in ESSL.

Linear Algebra Subprograms

There are several types of linear algebra subprograms.

- Vector-scalar linear algebra subprograms (“Vector-Scalar Linear Algebra Subprograms” on page 12)
- Sparse vector-scalar linear algebra subprograms (“Sparse Vector-Scalar Linear Algebra Subprograms” on page 14)
- Matrix-vector linear algebra subprograms (“Matrix-Vector Linear Algebra Subprograms” on page 14)
- Sparse matrix-vector linear algebra subprograms (“Sparse Matrix-Vector Linear Algebra Subprograms” on page 16)

Note:

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 [93 on page 1368]. 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 12. List of Vector-Scalar Linear Algebra Subprograms

<table>
<thead>
<tr>
<th>Short-Precision Subprogram</th>
<th>Long-Precision Subprogram</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISAMAX&quot;</td>
<td>IDAMAX&quot;</td>
<td>&quot;ISAMAX, IDAMAX, ICAMAX, and IZAMAX (Position of the First or Last Occurrence of the Vector Element Having the Largest Magnitude)” on page 233</td>
</tr>
<tr>
<td>ICAMAX&quot;</td>
<td>IDZAMAX&quot;</td>
<td>&quot;ISAMAX, IDAMAX, ICAMAX, and IZAMAX (Position of the First or Last Occurrence of the Vector Element Having the Largest Magnitude)” on page 233</td>
</tr>
<tr>
<td>cblas_isamax</td>
<td>cblas_idamax</td>
<td>&quot;ISAMAX, IDAMAX, ICAMAX, and IZAMAX (Position of the First or Last Occurrence of the Vector Element Having the Largest Magnitude)” on page 233</td>
</tr>
<tr>
<td>ICAMAX&quot;</td>
<td>IDZAMAX&quot;</td>
<td>&quot;ISAMAX, IDAMAX, ICAMAX, and IZAMAX (Position of the First or Last Occurrence of the Vector Element Having the Largest Magnitude)” on page 233</td>
</tr>
<tr>
<td>cblas_isamax</td>
<td>cblas_idamax</td>
<td>&quot;ISAMAX, IDAMAX, ICAMAX, and IZAMAX (Position of the First or Last Occurrence of the Vector Element Having the Largest Magnitude)” on page 233</td>
</tr>
<tr>
<td>ISAMIN&quot;</td>
<td>IDAMIN&quot;</td>
<td>&quot;ISAMIN and IDAMIN (Position of the First or Last Occurrence of the Vector Element Having Minimum Absolute Value)” on page 236</td>
</tr>
<tr>
<td>ICAMIN&quot;</td>
<td>IDZAMIN&quot;</td>
<td>&quot;ISAMIN and IDAMIN (Position of the First or Last Occurrence of the Vector Element Having Minimum Absolute Value)” on page 236</td>
</tr>
<tr>
<td>cblas_isamin</td>
<td>cblas_idamin</td>
<td>&quot;ISAMIN and IDAMIN (Position of the First or Last Occurrence of the Vector Element Having Minimum Absolute Value)” on page 236</td>
</tr>
<tr>
<td>ISMAX&quot;</td>
<td>IDMAX&quot;</td>
<td>&quot;ISMAX and IDMAX (Position of the First or Last Occurrence of the Vector Element Having the Maximum Value)” on page 239</td>
</tr>
<tr>
<td>ICAMAX&quot;</td>
<td>IDZAMAX&quot;</td>
<td>&quot;ISMAX and IDMAX (Position of the First or Last Occurrence of the Vector Element Having the Maximum Value)” on page 239</td>
</tr>
<tr>
<td>cblas_isamax</td>
<td>cblas_idamax</td>
<td>&quot;ISMAX and IDMAX (Position of the First or Last Occurrence of the Vector Element Having the Maximum Value)” on page 239</td>
</tr>
<tr>
<td>ISMIN&quot;</td>
<td>IDMIN&quot;</td>
<td>&quot;ISMIN and IDMIN (Position of the First or Last Occurrence of the Vector Element Having Minimum Value)” on page 242</td>
</tr>
<tr>
<td>ICAMIN&quot;</td>
<td>IDZAMIN&quot;</td>
<td>&quot;ISMIN and IDMIN (Position of the First or Last Occurrence of the Vector Element Having Minimum Value)” on page 242</td>
</tr>
<tr>
<td>cblas_isamin</td>
<td>cblas_idamin</td>
<td>&quot;ISMIN and IDMIN (Position of the First or Last Occurrence of the Vector Element Having Minimum Value)” on page 242</td>
</tr>
<tr>
<td>SASUM&quot;</td>
<td>DASUM&quot;</td>
<td>&quot;SASUM, DASUM, SCASUM, and DZASUM (Sum of the Magnitudes of the Elements in a Vector)” on page 245</td>
</tr>
<tr>
<td>SCASUM&quot;</td>
<td>DCASUM&quot;</td>
<td>&quot;SASUM, DASUM, SCASUM, and DZASUM (Sum of the Magnitudes of the Elements in a Vector)” on page 245</td>
</tr>
<tr>
<td>cblas_sasum</td>
<td>cblas_scasum</td>
<td>&quot;SASUM, DASUM, SCASUM, and DZASUM (Sum of the Magnitudes of the Elements in a Vector)” on page 245</td>
</tr>
<tr>
<td>DZASUM&quot;</td>
<td>DZCUM&quot;</td>
<td>&quot;SASUM, DASUM, SCASUM, and DZASUM (Sum of the Magnitudes of the Elements in a Vector)” on page 245</td>
</tr>
<tr>
<td>cblas_dzasum</td>
<td>cblas_dzum</td>
<td>&quot;SASUM, DASUM, SCASUM, and DZASUM (Sum of the Magnitudes of the Elements in a Vector)” on page 245</td>
</tr>
<tr>
<td>SCOPY&quot;</td>
<td>DCOPY&quot;</td>
<td>&quot;SCOPY, DCOPY, CCOPY, and ZCOPY (Copy a Vector)” on page 251</td>
</tr>
<tr>
<td>CCOPY&quot;</td>
<td>CCOPY&quot;</td>
<td>&quot;SCOPY, DCOPY, CCOPY, and ZCOPY (Copy a Vector)” on page 251</td>
</tr>
<tr>
<td>cblas_scopy</td>
<td>cblas_dcopy</td>
<td>&quot;SCOPY, DCOPY, CCOPY, and ZCOPY (Copy a Vector)” on page 251</td>
</tr>
<tr>
<td>ZCOPY&quot;</td>
<td>ZCOPY&quot;</td>
<td>&quot;SCOPY, DCOPY, CCOPY, and ZCOPY (Copy a Vector)” on page 251</td>
</tr>
<tr>
<td>cblas_zcopy</td>
<td>cblas_zcopy</td>
<td>&quot;SCOPY, DCOPY, CCOPY, and ZCOPY (Copy a Vector)” on page 251</td>
</tr>
<tr>
<td>SDOT&quot;</td>
<td>DDOT&quot;</td>
<td>&quot;SDOT, DDOT, CDOTU, CDOTC, and ZDOTC (Dot Product of Two Vectors)” on page 254</td>
</tr>
<tr>
<td>CDOTU&quot;</td>
<td>ZDOTU&quot;</td>
<td>&quot;SDOT, DDOT, CDOTU, CDOTC, and ZDOTC (Dot Product of Two Vectors)” on page 254</td>
</tr>
<tr>
<td>CDOTC&quot;</td>
<td>ZDOTC&quot;</td>
<td>&quot;SDOT, DDOT, CDOTU, CDOTC, and ZDOTC (Dot Product of Two Vectors)” on page 254</td>
</tr>
<tr>
<td>cblas_sdot</td>
<td>cblas_ddot</td>
<td>&quot;SDOT, DDOT, CDOTU, CDOTC, and ZDOTC (Dot Product of Two Vectors)” on page 254</td>
</tr>
<tr>
<td>cblas_zdotc</td>
<td>cblas_zdotc</td>
<td>&quot;SDOT, DDOT, CDOTU, CDOTC, and ZDOTC (Dot Product of Two Vectors)” on page 254</td>
</tr>
<tr>
<td>SNAXPY</td>
<td>DNAXPY</td>
<td>&quot;SNAXPY and DNAXPY (Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in the Vector Y)” on page 248</td>
</tr>
<tr>
<td>SNDOT</td>
<td>DNDOT</td>
<td>&quot;SNAXPY and DNAXPY (Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in the Vector Y)” on page 248</td>
</tr>
<tr>
<td>SNRM2&quot;</td>
<td>DNRM2&quot;</td>
<td>&quot;SNRM2, DNRM2, SCNRM2, and DZNRM2 (Euclidean Length of a Vector with Scaling of Input to Avoid Destructive Underflow and Overflow)” on page 268</td>
</tr>
<tr>
<td>SCNRM2&quot;</td>
<td>DZNRM2&quot;</td>
<td>&quot;SNRM2, DNRM2, SCNRM2, and DZNRM2 (Euclidean Length of a Vector with Scaling of Input to Avoid Destructive Underflow and Overflow)” on page 268</td>
</tr>
<tr>
<td>cblas_snrm2</td>
<td>cblas_dnrm2</td>
<td>&quot;SNRM2, DNRM2, SCNRM2, and DZNRM2 (Euclidean Length of a Vector with Scaling of Input to Avoid Destructive Underflow and Overflow)” on page 268</td>
</tr>
<tr>
<td>ZNRM2&quot;</td>
<td>ZCNRM2&quot;</td>
<td>&quot;SNRM2, DNRM2, SCNRM2, and DZNRM2 (Euclidean Length of a Vector with No Scaling of Input)” on page 271</td>
</tr>
</tbody>
</table>
| cblas_znorm2 | cblas_zcnrm2 | "SNRM2, DNRM2, SCNRM2, and DZNRM2 (Euclidean Length of a Vector with No Scaling of Input)” on page 271
<table>
<thead>
<tr>
<th>Short-Precision Subprogram</th>
<th>Long-Precision Subprogram</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SROTG</td>
<td>DROTG</td>
<td>“SROTG, DROTG, CROTG, and ZROTG (Construct a Givens Plane Rotation)” on page 274</td>
</tr>
<tr>
<td>CROTG</td>
<td>ZROTG</td>
<td></td>
</tr>
<tr>
<td>cblas_srotg</td>
<td>cblas_drotg</td>
<td></td>
</tr>
<tr>
<td>cblas_crotg</td>
<td>cblas_zrotg</td>
<td></td>
</tr>
<tr>
<td>SROT</td>
<td>DROT</td>
<td>“SROT, DROT, CROT, ZROT, CSROT, and ZDROT (Apply a Plane Rotation)” on page 280</td>
</tr>
<tr>
<td>CROT</td>
<td>ZROT</td>
<td></td>
</tr>
<tr>
<td>CSROT</td>
<td>ZDROT</td>
<td></td>
</tr>
<tr>
<td>cblas_srot</td>
<td>cblas_drot</td>
<td></td>
</tr>
<tr>
<td>cblas_crot</td>
<td>cblas_zrot</td>
<td></td>
</tr>
<tr>
<td>SROTMG</td>
<td>DROTMG</td>
<td>“SROTMG and DROTMG (Construct a modified Givens Transformation)” on page 284</td>
</tr>
<tr>
<td>cblas_srotmg</td>
<td>cblas_drotmg</td>
<td></td>
</tr>
<tr>
<td>SROTM</td>
<td>DROTM</td>
<td>“SROTM and DROTM (Apply a modified Givens Transformation)” on page 287</td>
</tr>
<tr>
<td>cblas_srotm</td>
<td>cblas_drotm</td>
<td></td>
</tr>
<tr>
<td>SSCAL</td>
<td>DSCAL</td>
<td>“SSCAL, DSCAL, CSCAL, ZSCAL, CCSCAL, and ZDSCAL (Multiply a Vector X by a Scalar and Store in the Vector X)” on page 290</td>
</tr>
<tr>
<td>CSCAL</td>
<td>ZSCAL</td>
<td></td>
</tr>
<tr>
<td>CSSCAL</td>
<td>ZDSCAL</td>
<td></td>
</tr>
<tr>
<td>cblas_sscal</td>
<td>cblas_dscal</td>
<td></td>
</tr>
<tr>
<td>cblas_cscal</td>
<td>cblas_zscal</td>
<td></td>
</tr>
<tr>
<td>SSWAP</td>
<td>DSWAP</td>
<td>“SSWAP, DSWAP, CSWAP, and ZSWAP (Interchange the Elements of Two Vectors)” on page 293</td>
</tr>
<tr>
<td>CSWAP</td>
<td>ZSWAP</td>
<td></td>
</tr>
<tr>
<td>cblas_sswap</td>
<td>cblas_dswap</td>
<td></td>
</tr>
<tr>
<td>cblas_cswap</td>
<td>cblas_zswap</td>
<td></td>
</tr>
<tr>
<td>SVEA</td>
<td>DVEA</td>
<td>“SVEA, DVEA, CVEA, and ZVEA (Add a Vector X to a Vector Y and Store in a Vector Z)” on page 296</td>
</tr>
<tr>
<td>CVEA</td>
<td>ZVEA</td>
<td></td>
</tr>
<tr>
<td>cblas_svea</td>
<td>cblas_dvea</td>
<td></td>
</tr>
<tr>
<td>cblas_cvnea</td>
<td>cblas_zvea</td>
<td></td>
</tr>
<tr>
<td>SVES</td>
<td>DVES</td>
<td>“SVES, DVES, CVES, and ZVES (Subtract a Vector Y from a Vector X and Store in a Vector Z)” on page 300</td>
</tr>
<tr>
<td>CVES</td>
<td>ZVES</td>
<td></td>
</tr>
<tr>
<td>cblas_sves</td>
<td>cblas_dves</td>
<td></td>
</tr>
<tr>
<td>cblas_cvzes</td>
<td>cblas_zves</td>
<td></td>
</tr>
<tr>
<td>SVEM</td>
<td>DVEM</td>
<td>“SVEM, DVEM, CVEM, and ZVEM (Multiply a Vector X by a Vector Y and Store in a Vector Z)” on page 304</td>
</tr>
<tr>
<td>CVEM</td>
<td>ZVEM</td>
<td></td>
</tr>
<tr>
<td>cblas_svem</td>
<td>cblas_dvem</td>
<td></td>
</tr>
<tr>
<td>cblas_cvem</td>
<td>cblas_zvem</td>
<td></td>
</tr>
<tr>
<td>SYAX</td>
<td>DYAX</td>
<td>“SYAX, DYAX, CYAX, ZYAX, CSYAX, and ZDYAX (Multiply a Vector X by a Scalar and Store in a Vector Y)” on page 308</td>
</tr>
<tr>
<td>CYAX</td>
<td>ZYAX</td>
<td></td>
</tr>
<tr>
<td>CSYAX</td>
<td>ZDYAX</td>
<td></td>
</tr>
<tr>
<td>cblas_syax</td>
<td>cblas_dyax</td>
<td></td>
</tr>
<tr>
<td>cblas_cyax</td>
<td>cblas_zyax</td>
<td></td>
</tr>
<tr>
<td>cblas_csyax</td>
<td>cblas_zdyax</td>
<td></td>
</tr>
<tr>
<td>SZAXPY</td>
<td>DZAXPY</td>
<td>“SZAXPY, DZAXPY, CZAXPY, and ZZAXPY (Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in a Vector Z)” on page 312</td>
</tr>
<tr>
<td>CZAXPY</td>
<td>ZZAXPY</td>
<td></td>
</tr>
<tr>
<td>cblas_szaxpy</td>
<td>cblas_dzaxpy</td>
<td></td>
</tr>
<tr>
<td>cblas_czaxpy</td>
<td>cblas_zzaxpy</td>
<td></td>
</tr>
</tbody>
</table>

† 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 [37 on page 1365]. Both real and complex versions of the subprograms are provided.

Table 13. List of Sparse Vector-Scalar Linear Algebra Subprograms

<table>
<thead>
<tr>
<th>Short-Precision Subprogram</th>
<th>Long-Precision Subprogram</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSCTR</td>
<td>DSCTR</td>
<td>“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)” on page 317</td>
</tr>
<tr>
<td>CSCTR</td>
<td>ZSCTR</td>
<td></td>
</tr>
<tr>
<td>SGTHR</td>
<td>DGTHR</td>
<td>“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)” on page 320</td>
</tr>
<tr>
<td>CGTHR</td>
<td>ZGTHR</td>
<td></td>
</tr>
<tr>
<td>SGTHRZ</td>
<td>DGTHRZ</td>
<td>“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)” on page 323</td>
</tr>
<tr>
<td>CGTHRZ</td>
<td>ZGTHRZ</td>
<td></td>
</tr>
<tr>
<td>SAXPYI</td>
<td>DAXPYI</td>
<td>“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)” on page 326</td>
</tr>
<tr>
<td>CAXPYI</td>
<td>ZAXPYI</td>
<td></td>
</tr>
<tr>
<td>SDOTI^1</td>
<td>DDOTI^1</td>
<td>“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)” on page 329</td>
</tr>
<tr>
<td>CDOTCI^1</td>
<td>ZDOTCI^1</td>
<td></td>
</tr>
<tr>
<td>CDOTUI^1</td>
<td>ZDOTUI^1</td>
<td></td>
</tr>
</tbody>
</table>

^1 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 [42 on page 1365] and [43 on page 1365]. Both real and complex versions of the subprograms are provided.

Table 14. List of Matrix-Vector Linear Algebra Subprograms

<table>
<thead>
<tr>
<th>Short-Precision Subprogram</th>
<th>Long-Precision Subprogram</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGEMV^*</td>
<td>DGEMV^*</td>
<td>“SGEMV, DGEMV, CGEMV, ZGEMV, SGEMX, DGEMX, SGEMTX, and DGEMTX (Matrix-Vector Product for a General Matrix, Its Transpose, or Its Conjugate Transpose)” on page 334</td>
</tr>
<tr>
<td>CGEMV^*</td>
<td>ZGEMV^*</td>
<td></td>
</tr>
<tr>
<td>SGEMX^§</td>
<td>DGEMX^§</td>
<td></td>
</tr>
<tr>
<td>SGEMTX^§</td>
<td>cblas_sgemv*</td>
<td></td>
</tr>
<tr>
<td>cblas_cgemv*</td>
<td>cblas_dgemv*</td>
<td></td>
</tr>
<tr>
<td>cblas_zgemv*</td>
<td>cblas_dgemv*</td>
<td></td>
</tr>
<tr>
<td>cblas_zgemv*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SGER^*</td>
<td>DGER^*</td>
<td>“SGER, DGER, CGERU, ZGERU, CGERC, and ZGERC (Rank-One Update of a General Matrix)” on page 345</td>
</tr>
<tr>
<td>CGERU^*</td>
<td>ZGERU^*</td>
<td></td>
</tr>
<tr>
<td>CGERC^*</td>
<td>cblas_sger*</td>
<td></td>
</tr>
<tr>
<td>cblas_cgeru*</td>
<td>cblas_zgeru*</td>
<td></td>
</tr>
<tr>
<td>cblas_cgerc*</td>
<td>cblas_zgerc*</td>
<td></td>
</tr>
</tbody>
</table>
Table 14. List of Matrix-Vector Linear Algebra Subprograms (continued)

<table>
<thead>
<tr>
<th>Short-Precision Subprogram</th>
<th>Long-Precision Subprogram</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSMV*</td>
<td>DSPMV*</td>
<td>“SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV, SSLMX, and DSLMX (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Matrix)” on page 353</td>
</tr>
<tr>
<td>CHPMV*</td>
<td>ZHPMV*</td>
<td></td>
</tr>
<tr>
<td>SSYMV*</td>
<td>DSYMV*</td>
<td></td>
</tr>
<tr>
<td>CHEMV*</td>
<td>ZHEMV*</td>
<td></td>
</tr>
<tr>
<td>SSLMX§</td>
<td>cblas_dspmv*</td>
<td></td>
</tr>
<tr>
<td>cblas_zhpnv*</td>
<td>cblas_dsymv*</td>
<td></td>
</tr>
<tr>
<td>cblas_chemv*</td>
<td>cblas_zhemv*</td>
<td></td>
</tr>
<tr>
<td>SSMV*</td>
<td>DSPMV*</td>
<td>“SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV, SSLMX, and DSLMX (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Matrix)” on page 353</td>
</tr>
<tr>
<td>CHPMV*</td>
<td>ZHPMV*</td>
<td></td>
</tr>
<tr>
<td>SSYMV*</td>
<td>DSYMV*</td>
<td></td>
</tr>
<tr>
<td>CHEMV*</td>
<td>ZHEMV*</td>
<td></td>
</tr>
<tr>
<td>SSLMX§</td>
<td>cblas_dspmv*</td>
<td></td>
</tr>
<tr>
<td>cblas_zhpnv*</td>
<td>cblas_dsymv*</td>
<td></td>
</tr>
<tr>
<td>cblas_chemv*</td>
<td>cblas_zhemv*</td>
<td></td>
</tr>
<tr>
<td>SSMV*</td>
<td>DSPMV*</td>
<td>“SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV, SSLMX, and DSLMX (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Matrix)” on page 353</td>
</tr>
<tr>
<td>CHPMV*</td>
<td>ZHPMV*</td>
<td></td>
</tr>
<tr>
<td>SSYMV*</td>
<td>DSYMV*</td>
<td></td>
</tr>
<tr>
<td>CHEMV*</td>
<td>ZHEMV*</td>
<td></td>
</tr>
<tr>
<td>SSLMX§</td>
<td>cblas_dspmv*</td>
<td></td>
</tr>
<tr>
<td>cblas_zhpnv*</td>
<td>cblas_dsymv*</td>
<td></td>
</tr>
<tr>
<td>cblas_chemv*</td>
<td>cblas_zhemv*</td>
<td></td>
</tr>
<tr>
<td>SSMV*</td>
<td>DSPMV*</td>
<td>“SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV, SSLMX, and DSLMX (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Matrix)” on page 353</td>
</tr>
<tr>
<td>CHPMV*</td>
<td>ZHPMV*</td>
<td></td>
</tr>
<tr>
<td>SSYMV*</td>
<td>DSYMV*</td>
<td></td>
</tr>
<tr>
<td>CHEMV*</td>
<td>ZHEMV*</td>
<td></td>
</tr>
<tr>
<td>SSLMX§</td>
<td>cblas_dspmv*</td>
<td></td>
</tr>
<tr>
<td>cblas_zhpnv*</td>
<td>cblas_dsymv*</td>
<td></td>
</tr>
<tr>
<td>cblas_chemv*</td>
<td>cblas_zhemv*</td>
<td></td>
</tr>
<tr>
<td>SSMV*</td>
<td>DSPMV*</td>
<td>“SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV, SSLMX, and DSLMX (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Matrix)” on page 353</td>
</tr>
<tr>
<td>CHPMV*</td>
<td>ZHPMV*</td>
<td></td>
</tr>
<tr>
<td>SSYMV*</td>
<td>DSYMV*</td>
<td></td>
</tr>
<tr>
<td>CHEMV*</td>
<td>ZHEMV*</td>
<td></td>
</tr>
<tr>
<td>SSLMX§</td>
<td>cblas_dspmv*</td>
<td></td>
</tr>
<tr>
<td>cblas_zhpnv*</td>
<td>cblas_dsymv*</td>
<td></td>
</tr>
<tr>
<td>cblas_chemv*</td>
<td>cblas_zhemv*</td>
<td></td>
</tr>
<tr>
<td>SSMV*</td>
<td>DSPMV*</td>
<td>“SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV, SSLMX, and DSLMX (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Matrix)” on page 353</td>
</tr>
<tr>
<td>CHPMV*</td>
<td>ZHPMV*</td>
<td></td>
</tr>
<tr>
<td>SSYMV*</td>
<td>DSYMV*</td>
<td></td>
</tr>
<tr>
<td>CHEMV*</td>
<td>ZHEMV*</td>
<td></td>
</tr>
<tr>
<td>SSLMX§</td>
<td>cblas_dspmv*</td>
<td></td>
</tr>
<tr>
<td>cblas_zhpnv*</td>
<td>cblas_dsymv*</td>
<td></td>
</tr>
<tr>
<td>cblas_chemv*</td>
<td>cblas_zhemv*</td>
<td></td>
</tr>
<tr>
<td>SSMV*</td>
<td>DSPMV*</td>
<td>“SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV, SSLMX, and DSLMX (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Matrix)” on page 353</td>
</tr>
<tr>
<td>CHPMV*</td>
<td>ZHPMV*</td>
<td></td>
</tr>
<tr>
<td>SSYMV*</td>
<td>DSYMV*</td>
<td></td>
</tr>
<tr>
<td>CHEMV*</td>
<td>ZHEMV*</td>
<td></td>
</tr>
<tr>
<td>SSLMX§</td>
<td>cblas_dspmv*</td>
<td></td>
</tr>
<tr>
<td>cblas_zhpnv*</td>
<td>cblas_dsymv*</td>
<td></td>
</tr>
<tr>
<td>cblas_chemv*</td>
<td>cblas_zhemv*</td>
<td></td>
</tr>
<tr>
<td>SSMV*</td>
<td>DSPMV*</td>
<td>“SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV, SSLMX, and DSLMX (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Matrix)” on page 353</td>
</tr>
<tr>
<td>CHPMV*</td>
<td>ZHPMV*</td>
<td></td>
</tr>
<tr>
<td>SSYMV*</td>
<td>DSYMV*</td>
<td></td>
</tr>
<tr>
<td>CHEMV*</td>
<td>ZHEMV*</td>
<td></td>
</tr>
<tr>
<td>SSLMX§</td>
<td>cblas_dspmv*</td>
<td></td>
</tr>
<tr>
<td>cblas_zhpnv*</td>
<td>cblas_dsymv*</td>
<td></td>
</tr>
<tr>
<td>cblas_chemv*</td>
<td>cblas_zhemv*</td>
<td></td>
</tr>
<tr>
<td>SSMV*</td>
<td>DSPMV*</td>
<td>“SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV, SSLMX, and DSLMX (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Matrix)” on page 353</td>
</tr>
<tr>
<td>CHPMV*</td>
<td>ZHPMV*</td>
<td></td>
</tr>
<tr>
<td>SSYMV*</td>
<td>DSYMV*</td>
<td></td>
</tr>
<tr>
<td>CHEMV*</td>
<td>ZHEMV*</td>
<td></td>
</tr>
<tr>
<td>SSLMX§</td>
<td>cblas_dspmv*</td>
<td></td>
</tr>
<tr>
<td>cblas_zhpnv*</td>
<td>cblas_dsymv*</td>
<td></td>
</tr>
<tr>
<td>cblas_chemv*</td>
<td>cblas_zhemv*</td>
<td></td>
</tr>
<tr>
<td>SSMV*</td>
<td>DSPMV*</td>
<td>“SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV, SSLMX, and DSLMX (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Matrix)” on page 353</td>
</tr>
<tr>
<td>CHPMV*</td>
<td>ZHPMV*</td>
<td></td>
</tr>
<tr>
<td>SSYMV*</td>
<td>DSYMV*</td>
<td></td>
</tr>
<tr>
<td>CHEMV*</td>
<td>ZHEMV*</td>
<td></td>
</tr>
<tr>
<td>SSLMX§</td>
<td>cblas_dspmv*</td>
<td></td>
</tr>
<tr>
<td>cblas_zhpnv*</td>
<td>cblas_dsymv*</td>
<td></td>
</tr>
<tr>
<td>cblas_chemv*</td>
<td>cblas_zhemv*</td>
<td></td>
</tr>
<tr>
<td>SSMV*</td>
<td>DSPMV*</td>
<td>“SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV, SSLMX, and DSLMX (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Matrix)” on page 353</td>
</tr>
<tr>
<td>CHPMV*</td>
<td>ZHPMV*</td>
<td></td>
</tr>
<tr>
<td>SSYMV*</td>
<td>DSYMV*</td>
<td></td>
</tr>
<tr>
<td>CHEMV*</td>
<td>ZHEMV*</td>
<td></td>
</tr>
<tr>
<td>SSLMX§</td>
<td>cblas_dspmv*</td>
<td></td>
</tr>
<tr>
<td>cblas_zhpnv*</td>
<td>cblas_dsymv*</td>
<td></td>
</tr>
<tr>
<td>cblas_chemv*</td>
<td>cblas_zhemv*</td>
<td></td>
</tr>
</tbody>
</table>
Table 14. List of Matrix-Vector Linear Algebra Subprograms (continued)

<table>
<thead>
<tr>
<th>Short-Precision Subprogram</th>
<th>Long-Precision Subprogram</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>STBMV*</td>
<td>DTBMV*</td>
<td>“STBMV, DTBMV, CTBMV, and ZTBMV (Matrix-Vector Product for a Triangular Band Matrix, Its Transpose, or Its Conjugate Transpose)” on page 405</td>
</tr>
<tr>
<td>CTBMV*</td>
<td>ZTBMV*</td>
<td></td>
</tr>
<tr>
<td>cblas_stbmv*</td>
<td>cblas_dtbmv*</td>
<td></td>
</tr>
<tr>
<td>cblas_ctbmv*</td>
<td>cblas_ztbmv*</td>
<td></td>
</tr>
<tr>
<td>STBSV*</td>
<td>DTBSV*</td>
<td>“STBSV, DTBSV, CTBSV, and ZTBSV (Triangular Band Equation Solve)” on page 411</td>
</tr>
<tr>
<td>CTBSV*</td>
<td>ZTBSV*</td>
<td></td>
</tr>
<tr>
<td>cblas_stbsv*</td>
<td>cblas_dtbsv*</td>
<td></td>
</tr>
<tr>
<td>cblas_ctbsv*</td>
<td>cblas_ztbsv*</td>
<td></td>
</tr>
</tbody>
</table>

* 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 15. List of Sparse Matrix-Vector Linear Algebra Subprograms

<table>
<thead>
<tr>
<th>Long-Precision Subprogram</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSMMX</td>
<td>“DSMMX (Matrix-Vector Product for a Sparse Matrix in Compressed-Matrix Storage Mode)” on page 418</td>
</tr>
<tr>
<td>DSMTM</td>
<td>“DSMTM (Transpose a Sparse Matrix in Compressed-Matrix Storage Mode)” on page 421</td>
</tr>
<tr>
<td>DSDMX</td>
<td>“DSDMX (Matrix-Vector Product for a Sparse Matrix or Its Transpose in Compressed-Diagonal Storage Mode)” on page 425</td>
</tr>
</tbody>
</table>

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 [40 on page 1365]. The matrix operation subroutines also include the commonly used matrix operations: addition, subtraction, multiplication, and transposition.

Table 16. List of Matrix Operation Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGEADD</td>
<td>DGEADD</td>
<td>“SGEADD, DGEADD, CGEADD, and ZGEMML (Matrix Addition for General Matrices or Their Transposes)” on page 434</td>
</tr>
<tr>
<td>CGEADD</td>
<td>ZGEMML</td>
<td></td>
</tr>
<tr>
<td>SGESUB</td>
<td>DGESUB</td>
<td>“SGESUB, DGESUB, CGESUB, and ZGESUB (Matrix Subtraction for General Matrices or Their Transposes)” on page 440</td>
</tr>
<tr>
<td>CGESUB</td>
<td>ZGESUB</td>
<td></td>
</tr>
<tr>
<td>SGEMUL</td>
<td>DGEMUL</td>
<td>“SGEMUL, DGEMUL, CGEMUL, and ZGEMUL (Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes)” on page 446</td>
</tr>
<tr>
<td>CGEMUL</td>
<td>ZGEMUL</td>
<td></td>
</tr>
</tbody>
</table>

16  ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
Table 16. List of Matrix Operation Subroutines (continued)

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGERVMS</td>
<td>DGERVMS</td>
<td>“SGEMMS, DGERVMS, CGERVMS, and ZGERVMS (Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes Using Winograd’s Variation of Strassen’s Algorithm)” on page 455</td>
</tr>
<tr>
<td>CGERMMS</td>
<td>ZGERMMS</td>
<td>“SGEMM, DGERVMS, CGERMMS, and ZGERMMS (Combined Matrix Multiplication and Addition for General Matrices, Their Transposes, or Conjugate Transposes)” on page 461</td>
</tr>
<tr>
<td>DSYMMS</td>
<td>ZSYMMS</td>
<td>“SSYMM, DSYMMS, CSYMM, ZSYMMS, CHEMM, and ZHEMM (Matrix-Matrix Product Where One Matrix is Real or Complex Symmetric or Complex Hermitian)” on page 470</td>
</tr>
<tr>
<td>STRMM*</td>
<td>DTRMM*</td>
<td>“STRMM, DTRMM, CTRMM, and ZTRMM (Triangular Matrix-Matrix Product)” on page 478</td>
</tr>
<tr>
<td>STRSM*</td>
<td>DTRSM*</td>
<td>“STRSM, DTRSM, CTRSM, and ZTRSM (Solution of Triangular Systems of Equations with Multiple Right-Hand Sides)” on page 486</td>
</tr>
<tr>
<td>DSYRK*</td>
<td>ZSYRK*</td>
<td>“SSYRK, DSYRK, CSYRK, ZSYRK, CHERK, and ZHERK (Rank-K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix)” on page 494</td>
</tr>
<tr>
<td>STRTV2K*</td>
<td>DSTRK2K*</td>
<td>“SSYRK2K, DSYRK2K, CSYRK2K, ZSYRK2K, CHERK2K, and ZHERK2K (Rank-2K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix)” on page 501</td>
</tr>
<tr>
<td>SGETMI</td>
<td>DGETMI</td>
<td>“SGETMI, DGETMI, CGETMI, ZGETMI, CGECMI and ZGECMI (General Matrix Transpose or Conjugate Transpose [In-Place])” on page 509</td>
</tr>
<tr>
<td>SGETMX</td>
<td>DGETMX</td>
<td>“SGETMO, DGETMO, CGETMO, ZGETMO, CGECMO, and ZGECMO (General Matrix Transpose or Conjugate Transpose [Out-of-Place])” on page 512</td>
</tr>
</tbody>
</table>

* 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” on page 18
- “Banded Linear Algebraic Equations” on page 21
- “Sparse Linear Algebraic Equations” on page 23
- “Linear Least Squares” on page 24
Note: Some of the linear algebraic equations were designed in accordance with the 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 LAPACK, see [8 on page 1363].

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, indefinite real or complex symmetric or complex Hermitian matrices, and triangular matrices. Some of these subroutines correspond to the LAPACK routines described in reference [8 on page 1363].

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGETRF&lt;sup&gt;A&lt;/sup&gt;</td>
<td>DGETRF&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;SGETRF, DGETRF, CGETRF and ZGETRF (General Matrix Factorization)&quot; on page 534</td>
</tr>
<tr>
<td>CGETRF&lt;sup&gt;A&lt;/sup&gt;</td>
<td>ZGETRF&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;CGETRF, ZGETRF, CGETRF and ZGETRF (General Matrix Factorization)&quot; on page 534</td>
</tr>
<tr>
<td>LAPACKE_sgetrf&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACKE_dgetrf&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;SGETRF, DGETRF, CGETRF and ZGETRF (General Matrix Factorization)&quot; on page 534</td>
</tr>
<tr>
<td>LAPACKE_cgetrf&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACKE_zgetrf&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;CGETRF, ZGETRF, CGETRF and ZGETRF (General Matrix Factorization)&quot; on page 534</td>
</tr>
<tr>
<td>SGETRS&lt;sup&gt;A&lt;/sup&gt;</td>
<td>DGETRS&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;SGETRS, DGETRS, CGETRS, and ZGETRS (General Matrix Multiple Right-Hand Side Solve)&quot; on page 539</td>
</tr>
<tr>
<td>CGETRS&lt;sup&gt;A&lt;/sup&gt;</td>
<td>ZGETRS&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;CGETRS, ZGETRS, CGETRS, and ZGETRS (General Matrix Multiple Right-Hand Side Solve)&quot; on page 539</td>
</tr>
<tr>
<td>LAPACKE_sgetrs&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACKE_dgetrs&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;SGETRS, DGETRS, CGETRS, and ZGETRS (General Matrix Multiple Right-Hand Side Solve)&quot; on page 539</td>
</tr>
<tr>
<td>LAPACKE_cgetrs&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACKE_zgetrs&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;CGETRS, ZGETRS, CGETRS, and ZGETRS (General Matrix Multiple Right-Hand Side Solve)&quot; on page 539</td>
</tr>
<tr>
<td>SGECON&lt;sup&gt;A&lt;/sup&gt;</td>
<td>DGECON&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;SGECON, DGECON, CGECON, and ZGECON (Estimate the Reciprocal of the Condition Number of a General Matrix)&quot; on page 556</td>
</tr>
<tr>
<td>CGECON&lt;sup&gt;A&lt;/sup&gt;</td>
<td>ZGECON&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;CGECON, ZGECON, CGECON, and ZGECON (Estimate the Reciprocal of the Condition Number of a General Matrix)&quot; on page 556</td>
</tr>
<tr>
<td>LAPACKE_sgecon&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACKE_dgecon&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;SGECON, DGECON, CGECON, and ZGECON (Estimate the Reciprocal of the Condition Number of a General Matrix)&quot; on page 556</td>
</tr>
<tr>
<td>LAPACKE_cgecon&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACKE_zgecon&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;CGECON, ZGECON, CGECON, and ZGECON (Estimate the Reciprocal of the Condition Number of a General Matrix)&quot; on page 556</td>
</tr>
<tr>
<td>SGETRI&lt;sup&gt;A&lt;/sup&gt;</td>
<td>DGETRI&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD (General Matrix Inverse, Condition Number Reciprocal, and Determinant)&quot; on page 565</td>
</tr>
<tr>
<td>CGETRI&lt;sup&gt;A&lt;/sup&gt;</td>
<td>ZGETRI&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD (General Matrix Inverse, Condition Number Reciprocal, and Determinant)&quot; on page 565</td>
</tr>
<tr>
<td>LAPACKE_sgetri&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACKE_dgetri&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD (General Matrix Inverse, Condition Number Reciprocal, and Determinant)&quot; on page 565</td>
</tr>
<tr>
<td>LAPACKE_cgetri&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACKE_zgetri&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD (General Matrix Inverse, Condition Number Reciprocal, and Determinant)&quot; on page 565</td>
</tr>
<tr>
<td>SLANGE&lt;sup&gt;A&lt;/sup&gt;</td>
<td>DLANGE&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;SLANGE, DLANGE, CLANGE, and ZLANGE (General Matrix Norm)&quot; on page 572</td>
</tr>
<tr>
<td>CLANGE&lt;sup&gt;A&lt;/sup&gt;</td>
<td>ZLANGE&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;SLANGE, DLANGE, CLANGE, and ZLANGE (General Matrix Norm)&quot; on page 572</td>
</tr>
<tr>
<td>LAPACKE_slange&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACKE_dlange&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;SLANGE, DLANGE, CLANGE, and ZLANGE (General Matrix Norm)&quot; on page 572</td>
</tr>
<tr>
<td>LAPACKE_clange&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACKE_zlange&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;SLANGE, DLANGE, CLANGE, and ZLANGE (General Matrix Norm)&quot; on page 572</td>
</tr>
<tr>
<td>SPPSV&lt;sup&gt;A&lt;/sup&gt;</td>
<td>DPPSV&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;SPPSV, DPPSV, CPPSV, and ZPPSV (Positive Definite Real Symmetric and Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)&quot; on page 573</td>
</tr>
<tr>
<td>CPPSV&lt;sup&gt;A&lt;/sup&gt;</td>
<td>ZPPSV&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;SPPSV, DPPSV, CPPSV, and ZPPSV (Positive Definite Real Symmetric and Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)&quot; on page 573</td>
</tr>
<tr>
<td>LAPACKE_sppsv&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACKE_dppsv&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;SPPSV, DPPSV, CPPSV, and ZPPSV (Positive Definite Real Symmetric and Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)&quot; on page 573</td>
</tr>
<tr>
<td>LAPACKE_cppsv&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACKE_zppsv&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;SPPSV, DPPSV, CPPSV, and ZPPSV (Positive Definite Real Symmetric and Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)&quot; on page 573</td>
</tr>
<tr>
<td>SPOSV&lt;sup&gt;A&lt;/sup&gt;</td>
<td>DPOSV&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;SPOSV, DPOSV, CPOSV, and ZPOSV (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)&quot; on page 581</td>
</tr>
<tr>
<td>CPOSV&lt;sup&gt;A&lt;/sup&gt;</td>
<td>ZPOSV&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;SPOSV, DPOSV, CPOSV, and ZPOSV (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)&quot; on page 581</td>
</tr>
<tr>
<td>LAPACKE_sposv&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACKE_dposv&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;SPOSV, DPOSV, CPOSV, and ZPOSV (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)&quot; on page 581</td>
</tr>
<tr>
<td>LAPACKE_cposv&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACKE_zposv&lt;sup&gt;A&lt;/sup&gt;</td>
<td>&quot;SPOSV, DPOSV, CPOSV, and ZPOSV (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)&quot; on page 581</td>
</tr>
</tbody>
</table>
### Table 17. List of LAPACK Dense Linear Algebraic Equation Subroutines (continued)

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPOTRF&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DPOTRF&lt;sup&gt;a&lt;/sup&gt;</td>
<td>SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF; SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPPE, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)” on page 587</td>
</tr>
<tr>
<td>CPOTRF&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZPOTRF&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SPPTRF&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DPPTRF&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CPPTRF&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZPPTRF&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;spotrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;dpotrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;cpotrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;zpotrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;spptrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;dpptrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;cpptrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;zpptrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SPOTRS&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DPOTRS&lt;sup&gt;a&lt;/sup&gt;</td>
<td>SPOTRS, DPOTRS, CPOTRS, ZPOTRS, SPOSM, DPOSM, CPOSM, ZPOSM, SPPTRS, DPPTRS, CPPTRS, and ZPPTRS (Positive Definite Real Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)” on page 600</td>
</tr>
<tr>
<td>CPOTRS&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZPOTRS&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SPPTRS&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DPPTRS&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CPPTRS&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZPPTRS&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;spotrs&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;dpotrs&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;cpotrs&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;zpotrs&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;spptrs&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;dpptrs&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;cpptrs&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;zpptrs&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SPOCON&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DPOCON&lt;sup&gt;a&lt;/sup&gt;</td>
<td>SPOCON, DPOCON, CPOCON, ZPOCON, SPOIC, DPOIC, CPOIC, ZPOIC, CPPCON, and ZPPCON (Estimate the Reciprocal of the Condition Number of a Positive Definite Real Symmetric or Complex Hermitian Matrix)” on page 611</td>
</tr>
<tr>
<td>CPOCON&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZPOCON&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SSPCON&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DPPCON&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CPPCON&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZPPCON&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;spon&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;dpcon&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;cpon&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;zpon&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;sppcon&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;dpcon&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;cppcon&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;zppcon&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SPOTRI&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DPOTRI&lt;sup&gt;a&lt;/sup&gt;</td>
<td>SPOTRI, DPOTRI, CPOTRI, ZPOTRI, SPOICD, DPOICD, SPPI, DPPTRI, CPPTRI, ZPPTRI, SPPE, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Inverse, Condition Number Reciprocal, and Determinant)” on page 626</td>
</tr>
<tr>
<td>CPOTRI&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZPOTRI&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SPPTRI&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DPPTRI&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CPPTRI&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZPPTRI&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;sportri&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;dpotri&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;cppotri&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;zpotri&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;spptr&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;dpptr&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;cpptr&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;zpptr&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SLANSY&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DLANSY&lt;sup&gt;a&lt;/sup&gt;</td>
<td>SLANSY, DLANSY, CLANHE, ZLANHE, SLANSP, DLANSP, CLANHP, and ZLANHP (Real Symmetric or Complex Hermitian Matrix Norm)” on page 637</td>
</tr>
<tr>
<td>CLANHE&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DLANHE&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SLANSP&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DLANSP&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CLANHP&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZLANHP&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;lan&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;dlan&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;clan&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;zlhan&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;lan&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;dlan&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;clanhp&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;zlhanhp&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SSYSV&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DSYSV&lt;sup&gt;a&lt;/sup&gt;</td>
<td>SSYSV, DSYSV, CSYSV, ZSYSV, CHESV, ZHESV, SSPSV, DSPSV, CPSV, ZCPSV, ZHPSV, and ZHPF (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)” on page 642</td>
</tr>
<tr>
<td>CSYSV&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZSYSV&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CHESV&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZHESV&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SSPSV&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DSSPSV&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CPSV&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZCPSV&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CHPSV&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZCHPSV&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;ssysv&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;dsysv&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;csysv&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;zsysv&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;chesv&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;zhesv&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;sspsv&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;dspsv&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;cppsv&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;zcppsv&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;chpsv&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK&lt;sub&gt;s&lt;/sub&gt;zchpsv&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
</tbody>
</table>
Table 17. List of LAPACK Dense Linear Algebraic Equation Subroutines (continued)

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSYTRF³</td>
<td>DSYTRF³</td>
<td>&quot;SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, ZHETRF, SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF, and ZHPTRF (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Factorization)&quot; on page 651</td>
</tr>
<tr>
<td>CSYTRF³</td>
<td>ZSYTRF³</td>
<td></td>
</tr>
<tr>
<td>CHETRF³</td>
<td>ZHETRF³</td>
<td></td>
</tr>
<tr>
<td>SSPTRF³</td>
<td>DSPTRF³</td>
<td></td>
</tr>
<tr>
<td>CSPTRF³</td>
<td>ZSPTRF³</td>
<td></td>
</tr>
<tr>
<td>CHPTRF³</td>
<td>ZHPTRF³</td>
<td></td>
</tr>
<tr>
<td>LAPACKESsytrf³</td>
<td>LAPACKEDsytrf³</td>
<td></td>
</tr>
<tr>
<td>LAPACKECsytrf³</td>
<td>LAPACKECsytrf³</td>
<td></td>
</tr>
<tr>
<td>LAPACKECHetr³</td>
<td>LAPACKECHetr³</td>
<td></td>
</tr>
<tr>
<td>LAPACKESSptrf³</td>
<td>LAPACKEDsptrf³</td>
<td></td>
</tr>
<tr>
<td>LAPACKECsptrf³</td>
<td>LAPACKECsptrf³</td>
<td></td>
</tr>
<tr>
<td>LAPACKECHptrf³</td>
<td>LAPACKECzhptrf³</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSSYTRF³</td>
<td>DSSYTRF³</td>
<td>&quot;SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, ZHETRF, SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF, and ZHPTRF (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)&quot; on page 660</td>
</tr>
<tr>
<td>CCSYTRF³</td>
<td>ZCSYTRF³</td>
<td></td>
</tr>
<tr>
<td>CHCTRF³</td>
<td>ZCHCTRF³</td>
<td></td>
</tr>
<tr>
<td>SSPSTRF³</td>
<td>DSSPTRF³</td>
<td></td>
</tr>
<tr>
<td>CSPSTRF³</td>
<td>ZCSPSTRF³</td>
<td></td>
</tr>
<tr>
<td>CCPTRF³</td>
<td>ZCCPTRF³</td>
<td></td>
</tr>
<tr>
<td>LAPACKESsytrs³</td>
<td>LAPACKEDsytrs³</td>
<td></td>
</tr>
<tr>
<td>LAPACKECsytrs³</td>
<td>LAPACKECsytrs³</td>
<td></td>
</tr>
<tr>
<td>LAPACKECHetr³</td>
<td>LAPACKECHetr³</td>
<td></td>
</tr>
<tr>
<td>LAPACKESSptstr³</td>
<td>LAPACKEDsptr³</td>
<td></td>
</tr>
<tr>
<td>LAPACKECsptr³</td>
<td>LAPACKECsptr³</td>
<td></td>
</tr>
<tr>
<td>LAPACKECHptr³</td>
<td>LAPACKECHptr³</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLANTR³</td>
<td>DLANTR³</td>
<td>&quot;SLANTR, DLANTR, CLANTR, ZLANTR, SLANTP, DLANTP, CLANTP, and ZLANTP (Trapezoidal or Triangular Matrix Norm)&quot; on page 690</td>
</tr>
<tr>
<td>CLANTR³</td>
<td>ZLANTR³</td>
<td></td>
</tr>
<tr>
<td>SLANTP³</td>
<td>DLANTP³</td>
<td></td>
</tr>
<tr>
<td>CLANTP³</td>
<td>ZLANTP³</td>
<td></td>
</tr>
<tr>
<td>LAPACKESlantr³</td>
<td>LAPACKEDlantr³</td>
<td></td>
</tr>
<tr>
<td>LAPACKEClantr³</td>
<td>LAPACKEClantr³</td>
<td></td>
</tr>
<tr>
<td>LAPACKESlantp³</td>
<td>LAPACKEDlantp³</td>
<td></td>
</tr>
<tr>
<td>LAPACKECclantr³</td>
<td>LAPACKEClantr³</td>
<td></td>
</tr>
<tr>
<td>LAPACKECclantp³</td>
<td>LAPACKEClantp³</td>
<td></td>
</tr>
<tr>
<td>LAPACK</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 18. List of Dense Linear Algebraic Equation Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGEF</td>
<td>DGEF</td>
<td>“SGEF, DGEF, CGEF, and ZGEF (General Matrix Factorization)&quot; on page 544</td>
</tr>
<tr>
<td>CGEF</td>
<td>ZGEF</td>
<td></td>
</tr>
<tr>
<td>DGEFP⁵</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SGESM</td>
<td>DGESM</td>
<td>“SGESM, DGESM, CGESM, and ZGESM (General Matrix, Its Transpose, or Its Conjugate Transpose Multiple Right-Hand Side Solve)&quot; on page 551</td>
</tr>
<tr>
<td>CGESM</td>
<td>ZGESM</td>
<td></td>
</tr>
<tr>
<td>SGES</td>
<td>DGES</td>
<td>“SGES, DGES, CGES, and ZGES (General Matrix, Its Transpose, or Its Conjugate Transpose Solve)&quot; on page 547</td>
</tr>
<tr>
<td>CGES</td>
<td>ZGES</td>
<td></td>
</tr>
</tbody>
</table>

ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
Table 18. List of Dense Linear Algebraic Equation Subroutines (continued)

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGEFCD</td>
<td>DGEFCD</td>
<td>“SGEFCD and DGEFCD (General Matrix Factorization, Condition Number, Reciprocal, and Determinant)” on page 561</td>
</tr>
<tr>
<td>SGEICD</td>
<td>DGEICD</td>
<td>“SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD (General Matrix Inverse, Condition Number Reciprocal, and Determinant)” on page 565</td>
</tr>
<tr>
<td>SPOF CPOF SPPF</td>
<td>DPOF ZPOF DPPF DPPFP</td>
<td>“SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPPF, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)” on page 587</td>
</tr>
<tr>
<td>SPOSM CPOSM</td>
<td>DPOSM ZPOSM</td>
<td>“SPOTRS, DPOTRS, CPOTRS, ZPOTRS, SPOSM, DPOSM, CPOSM, ZPOSM, SPPTRS, DPPTRS, CPPTRS, and ZPPTRS (Positive Definite Real Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)” on page 600</td>
</tr>
<tr>
<td>SPPS</td>
<td>DPSS</td>
<td>“SPPS and DPSS (Positive Definite Real Symmetric Matrix Solve)” on page 608</td>
</tr>
<tr>
<td>SPPFCD SPOFCD</td>
<td>DPPFCD DPOFCD</td>
<td>“SPPFCD, DPPFCD, SPOFCD, and DPOFCD (Positive Definite Real Symmetric Matrix Factorization, Condition Number Reciprocal, and Determinant)” on page 620</td>
</tr>
<tr>
<td>SPPICD SPOICD</td>
<td>DPPICD DPOICD</td>
<td>“SPOTRI, DPOTRI, CPOTRI, ZPOTRI, SPOICD, DPOICD, SPPTRI, DPPTRI, CPPTRI, ZPPTRI, SPPICD, and DPPICD (Positive Definite Real Symmetric or Complex Hermitian Matrix Inverse, Condition Number Reciprocal, and Determinant)” on page 626</td>
</tr>
<tr>
<td>DBSSV</td>
<td></td>
<td>“DBSSV (Symmetric Indefinite Matrix Factorization and Multiple Right-Hand Side Solve)” on page 667</td>
</tr>
<tr>
<td>DBSTRF</td>
<td></td>
<td>“DBSTRF (Symmetric Indefinite Matrix Factorization)” on page 673</td>
</tr>
<tr>
<td>DBTRSF</td>
<td></td>
<td>“DBTRSF (Symmetric Indefinite Matrix Multiple Right-Hand Side Solve)” on page 678</td>
</tr>
<tr>
<td>STRI STPI</td>
<td>DTRI DTPI</td>
<td>“STRTRI, DTRTRI, CTRTRI, ZTRTRI, STPTRI, DTPTRI, CTPTRI, and ZTPTRI (Triangular Matrix Inverse)” on page 682</td>
</tr>
</tbody>
</table>

† 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.

Banded Linear Algebraic Equations
The banded linear algebraic equation subroutines provide solutions to linear systems of equations for:
• Real or complex general band matrices
• Positive definite real symmetric or complex Hermitian band matrices
• Real or complex general tridiagonal matrices
• Positive definite real symmetric or complex Hermitian tridiagonal matrices

Table 19. List of LAPACK Banded Linear Algebraic Equation Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGBSV A</td>
<td>DGBSV A</td>
<td>“SGBSV, DGBSV, CGBSV, and ZGBSV (General Band Matrix Factorization and Multiple Right-Hand Side Solve)” on page 698</td>
</tr>
<tr>
<td>CGBSV A</td>
<td>ZGBSV A</td>
<td></td>
</tr>
<tr>
<td>LAPACKES_gbsv A</td>
<td>LAPACKE_dgbsv A</td>
<td></td>
</tr>
<tr>
<td>LAPACKEC_gbsv A</td>
<td>LAPACKE_zgbsv A</td>
<td></td>
</tr>
</tbody>
</table>
Table 19. List of LAPACK Banded Linear Algebraic Equation Subroutines (continued)

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGBTRF&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DGBTRF&lt;sup&gt;a&lt;/sup&gt;</td>
<td>“SGBTRF, DGBTRF, CGBTRF and ZGBTRF (General Band Matrix Factorization)” on page 702</td>
</tr>
<tr>
<td>CGBTRF&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZGBTRF&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_sgbtrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK_dgbtrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_cgbtrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK_zgbtrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SGBTRS&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DGBTRS&lt;sup&gt;a&lt;/sup&gt;</td>
<td>“SGBTRS, DGBTRS, CGBTRS, and ZGBTRS (General Band Matrix Multiple Right-Hand Side Solve)” on page 706</td>
</tr>
<tr>
<td>CGBTRS&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZGBTRS&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_sgbtrs&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK_dgbtrs&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_cgbtrs&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK_zgbtrs&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SPBSV&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DPBSV&lt;sup&gt;a&lt;/sup&gt;</td>
<td>“SPBSV, DPBSV, CPBSV, and ZPBSV (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization and Multiple Right-Hand Side Solve)” on page 715</td>
</tr>
<tr>
<td>CPBSV&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZPBSV&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_spbsv&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK_dpbsv&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_cpbsv&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK_zpbsv&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SPBTRF&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DPBTRF&lt;sup&gt;a&lt;/sup&gt;</td>
<td>“SPBTRF, DPBTRF, CPBTRF, and ZPBTRF (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization)” on page 721</td>
</tr>
<tr>
<td>CPBTRF&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZPBTRF&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_spbtrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK_dpbrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_cpbtrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK_zpbtrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SPBTRS&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DPBTRS&lt;sup&gt;a&lt;/sup&gt;</td>
<td>“SPBTRS, DPBTRS, CPBTRS, and ZPBTRS (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Multiple Right-Hand Side Solve)” on page 726</td>
</tr>
<tr>
<td>CPBTRS&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZPBTRS&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_spbtrs&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK_dpbrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_cpbtrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK_zpbtrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SGTYS&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DGTYS&lt;sup&gt;a&lt;/sup&gt;</td>
<td>“SGTYS, DGTYS, CGTYS, and ZGTYS (General Tridiagonal Matrix Factorization and Multiple Right-Hand Side Solve)” on page 731</td>
</tr>
<tr>
<td>CGTYS&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZGTYS&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_sgtys&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK_dgtys&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_cgtys&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK_zgtys&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SGTTRF&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DGTTRF&lt;sup&gt;a&lt;/sup&gt;</td>
<td>“SGTTRF, DGTTRF, CGTTRF, and ZGTTRF (General Tridiagonal Matrix Factorization)” on page 735</td>
</tr>
<tr>
<td>CGTTRF&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZGTTRF&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_sgttrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK_dgttrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_cgttrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK_zgttrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SGTTRS&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DGTTRS&lt;sup&gt;a&lt;/sup&gt;</td>
<td>“SGTTRS, DGTTRS, CGTTRS, and ZGTTRS (General Tridiagonal Matrix Multiple Right-Hand Side Solve)” on page 740</td>
</tr>
<tr>
<td>CGTTRS&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZGTTRS&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_sgttrs&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK_dgttrs&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_cgttrs&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK_zgttrs&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SPTYS&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DPTYS&lt;sup&gt;a&lt;/sup&gt;</td>
<td>“SPTYS, DPTYS, CPTYS, and ZPTYS (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization and Multiple Right-Hand Side Solve)” on page 746</td>
</tr>
<tr>
<td>CPTYS&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZPTYS&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_sptys&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK_dptys&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_cptys&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK_zptys&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SPTTRF&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DPTTRF&lt;sup&gt;a&lt;/sup&gt;</td>
<td>“SPTTRF, DPTTRF, CPTTRF, and ZPTTRF (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization)” on page 751</td>
</tr>
<tr>
<td>CPTTRF&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZPTTRF&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_spttrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK_dpttrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_cpttrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK_zpttrf&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SPTTRS&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DPTTRS&lt;sup&gt;a&lt;/sup&gt;</td>
<td>“SPTTRS, DPTTRS, CPPTRS, and ZPTTRS (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Multiple Right-Hand Solve)” on page 755</td>
</tr>
<tr>
<td>CPTTRS&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZPTTRS&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_spttrs&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK_dpttrs&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_cpttrs&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACK_zpttrs&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
</tbody>
</table>

<sup>a</sup> LAPACK

Table 20. List of non-LAPACK Banded Linear Algebraic Equation Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGBF&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DGBF&lt;sup&gt;a&lt;/sup&gt;</td>
<td>“SGBF and DGBF (General Band Matrix Factorization)” on page 761</td>
</tr>
</tbody>
</table>

22 ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
Table 20. List of non-LAPACK Banded Linear Algebraic Equation Subroutines (continued)

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGBS§</td>
<td>DGBS§</td>
<td>“SGBS and DGBS (General Band Matrix Solve)” on page 712</td>
</tr>
<tr>
<td>SGBS§</td>
<td>DGBS§</td>
<td>“SGBS and DGBS (General Band Matrix Solve)” on page 712</td>
</tr>
<tr>
<td>SPBF§</td>
<td>DPBF§</td>
<td>“SPBF, DPBF, SPBCHF, and DPBCHF (Positive Definite Symmetric Band Matrix Factorization)” on page 768</td>
</tr>
<tr>
<td>SPBF§</td>
<td>DPBCHF§</td>
<td>“SPBF, DPBF, SPBCHF, and DPBCHF (Positive Definite Symmetric Band Matrix Factorization)” on page 768</td>
</tr>
<tr>
<td>SPBS§</td>
<td>DPBS§</td>
<td>“SPBS, DPBS, SPBCHS, and DPBCHS (Positive Definite Symmetric Band Matrix Solve)” on page 772</td>
</tr>
<tr>
<td>SPBS§</td>
<td>DPBCHS§</td>
<td>“SPBS, DPBS, SPBCHS, and DPBCHS (Positive Definite Symmetric Band Matrix Solve)” on page 772</td>
</tr>
<tr>
<td>SGTF§</td>
<td>DGTF§</td>
<td>“SGTF and DGTF (General Triangular Matrix Factorization)” on page 775</td>
</tr>
<tr>
<td>SGTF§</td>
<td>DGTS§</td>
<td>“SGTS and DGTS (General Triangular Matrix Solve)” on page 778</td>
</tr>
<tr>
<td>SGTPF</td>
<td>DGTPF</td>
<td>“SGTF and DGTF (General Triangular Matrix Factorization)” on page 775</td>
</tr>
<tr>
<td>SGNPF</td>
<td>DGNPF</td>
<td>“SGNPF, DGTNP, CGTNP, and ZGTNP (General Tridiagonal Matrix Combined Factorization and Solve with No Pivoting)” on page 775</td>
</tr>
<tr>
<td>SPSF</td>
<td>DPSF</td>
<td>“SGTPF, DGTNP, CGTNP, and ZGTNP (General Tridiagonal Matrix Combined Factorization and Solve with No Pivoting)” on page 775</td>
</tr>
<tr>
<td>SPBSF</td>
<td>DPSF</td>
<td>“SGNPF, DGTNP, CGTNP, and ZGTNP (General Tridiagonal Matrix Combined Factorization and Solve with No Pivoting)” on page 775</td>
</tr>
<tr>
<td>SPSBF</td>
<td>DPSBF</td>
<td>“SGNPF, DGTNP, CGTNP, and ZGTNP (General Tridiagonal Matrix Combined Factorization and Solve with No Pivoting)” on page 775</td>
</tr>
<tr>
<td>SPTF§</td>
<td>DPTF§</td>
<td>“SPTF and DPTF (Positive Definite Symmetric Triangular Matrix Factorization)” on page 789</td>
</tr>
<tr>
<td>SPTF§</td>
<td>DPTS§</td>
<td>“SPTF and DPTF (Positive Definite Symmetric Triangular Matrix Factorization)” on page 789</td>
</tr>
<tr>
<td>SPTF§</td>
<td>DPTS§</td>
<td>“SPTF and DPTF (Positive Definite Symmetric Triangular Matrix Factorization)” on page 789</td>
</tr>
</tbody>
</table>

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

**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 21. List of Sparse Linear Algebraic Equation Subroutines

<table>
<thead>
<tr>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGSF</td>
<td>“DGSF (General Sparse Matrix Factorization Using Storage by Indices, Rows, or Columns)” on page 794</td>
</tr>
<tr>
<td>DGSS</td>
<td>“DGSS (General Sparse Matrix or Its Transpose Solve Using Storage by Indices, Rows, or Columns)” on page 800</td>
</tr>
<tr>
<td>DGKFS</td>
<td>“DGKFS (General Sparse Matrix Factorization, Determinant, and Solve Using Skyline Storage Mode)” on page 804</td>
</tr>
<tr>
<td>DSKFS</td>
<td>“DSKFS (Symmetric Sparse Matrix Factorization, Determinant, and Solve Using Skyline Storage Mode)” on page 821</td>
</tr>
<tr>
<td>DSRIS</td>
<td>“DSRIS (Iterative Linear System Solver for a General or Symmetric Sparse Matrix Stored by Rows)” on page 839</td>
</tr>
<tr>
<td>DSMCG§</td>
<td>“DSMCG (Sparse Positive Definite or Negative Definite Symmetric Matrix Iterative Solve Using Compressed-Matrix Storage Mode)” on page 850</td>
</tr>
<tr>
<td>DSDCG</td>
<td>“DSDCG (Sparse Positive Definite or Negative Definite Symmetric Matrix Iterative Solve Using Compressed-Diagonal Storage Mode)” on page 858</td>
</tr>
<tr>
<td>DSMGGCG§</td>
<td>“DSMGGCG (General Sparse Matrix Iterative Solve Using Compressed-Matrix Storage Mode)” on page 866</td>
</tr>
</tbody>
</table>
Table 21. List of Sparse Linear Algebraic Equation Subroutines (continued)

<table>
<thead>
<tr>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSDGCG</td>
<td>“DSDGCG (General Sparse Matrix Iterative Solve Using Compressed-Diagonal Storage Mode)” on page 873</td>
</tr>
</tbody>
</table>

³ 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.

³ 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 general matrices using a QR factorization or a singular value decomposition. Some of these subroutines correspond to the LAPACK routines described in reference [8 on page 1363].

Table 22. List of LAPACK Linear Least Squares Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGEQRF³</td>
<td>DGEQRF³</td>
<td>“SGEQRF, DGEQRF, CGEQRF, and ZGEQRF (General Matrix QR Factorization)” on page 895</td>
</tr>
<tr>
<td>CGEQR³</td>
<td>ZGEQRF³</td>
<td>LAPACK_dgeqrf³</td>
</tr>
<tr>
<td>LAPACK_sgeqrf³</td>
<td>LAPACK_zgeqrf³</td>
<td></td>
</tr>
<tr>
<td>SGEVS³</td>
<td>DGEVS³</td>
<td>“SGEVS, DGEVS, CGEVS, and ZGELLS (Linear Least Squares Solution for a General Matrix)” on page 901</td>
</tr>
<tr>
<td>CGEVS³</td>
<td>ZGEVS³</td>
<td>LAPACK_dgels³</td>
</tr>
<tr>
<td>LAPACK_sgevs³</td>
<td>LAPACK_zgels³</td>
<td></td>
</tr>
<tr>
<td>SGELSD³</td>
<td>DGELSD³</td>
<td>“SGELSD, DGELSD, CGELSD, and ZGELSD (Linear Least Squares Solution for a General Matrix Using the Singular Value Decomposition)” on page 911</td>
</tr>
<tr>
<td>CGELSD³</td>
<td>ZGELSD³</td>
<td>LAPACK_dgelsd³</td>
</tr>
<tr>
<td>LAPACK_sgelSD³</td>
<td>LAPACK_zgelsd³</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LAPACK_cgelSD³</td>
<td></td>
</tr>
</tbody>
</table>

³ LAPACK

Table 23. List of Non–LAPACK Linear Least Squares Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGESVF⁸</td>
<td>DGESVF⁸</td>
<td>“SGESVF and DGESVF (Singular Value Decomposition for a General Matrix)” on page 919</td>
</tr>
<tr>
<td>SGESVS⁸</td>
<td>DGESVS⁸</td>
<td>“SGESVS and DGESVS (Linear Least Squares Solution for a General Matrix Using the Singular Value Decomposition)” on page 927</td>
</tr>
<tr>
<td>SGELLS³</td>
<td>DGELLS³</td>
<td>“SGELLS and DGELLS (Linear Least Squares Solution for a General Matrix with Column Pivoting)” on page 932</td>
</tr>
</tbody>
</table>
### Eigensystem Analysis

The eigensystem analysis subroutines provide solutions to the algebraic eigensystem analysis problem and the generalized eigensystem analysis problem. These subroutines correspond to the LAPACK routines described in reference [8 on page 1363](#).

### Table 24. List of LAPACK Eigensystem Analysis Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGEEV&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DGEEV&lt;sup&gt;a&lt;/sup&gt;</td>
<td>SGEEV, DGEEV, CGEEV, ZGEEV, SGEEVX, DGEEVX, CGEEVX, and ZGEEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix)” on page 942</td>
</tr>
<tr>
<td>CGEEV&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZGEEV&lt;sup&gt;a&lt;/sup&gt;</td>
<td>SGEEV, DGEEV, CGEEV, ZGEEV, SGEEVX, DGEEVX, CGEEVX, and ZGEEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix)” on page 942</td>
</tr>
<tr>
<td>SGEEVX&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DGEEVX&lt;sup&gt;a&lt;/sup&gt;</td>
<td>SGEEV, DGEEV, CGEEV, ZGEEV, SGEEVX, DGEEVX, CGEEVX, and ZGEEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix)” on page 942</td>
</tr>
<tr>
<td>CGEEVX&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZGEEVX&lt;sup&gt;a&lt;/sup&gt;</td>
<td>SGEEV, DGEEV, CGEEV, ZGEEV, SGEEVX, DGEEVX, CGEEVX, and ZGEEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix)” on page 942</td>
</tr>
<tr>
<td>LAPACKE_sgeev&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACKE_dgeev&lt;sup&gt;a&lt;/sup&gt;</td>
<td>SGEEV, DGEEV, CGEEV, ZGEEV, SGEEVX, DGEEVX, CGEEVX, and ZGEEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix)” on page 942</td>
</tr>
<tr>
<td>LAPACKE_cggeev&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACKE_zggeev&lt;sup&gt;a&lt;/sup&gt;</td>
<td>SGEEV, DGEEV, CGEEV, ZGEEV, SGEEVX, DGEEVX, CGEEVX, and ZGEEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix)” on page 942</td>
</tr>
<tr>
<td>LAPACKE_sggeev&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACKE_dggeev&lt;sup&gt;a&lt;/sup&gt;</td>
<td>SGEEV, DGEEV, CGEEV, ZGEEV, SGEEVX, DGEEVX, CGEEVX, and ZGEEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix)” on page 942</td>
</tr>
<tr>
<td>LAPACKE_atlas&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACKE_zatlas&lt;sup&gt;a&lt;/sup&gt;</td>
<td>SGEEV, DGEEV, CGEEV, ZGEEV, SGEEVX, DGEEVX, CGEEVX, and ZGEEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix)” on page 942</td>
</tr>
</tbody>
</table>

<sup>a</sup> This subroutine is provided only for migration from earlier releases of ESSL and is not intended for use in new programs.
### Fourier Transforms, Convolutions and Correlations, and Related Computations

This signal processing area provides:
- Fourier transform subroutines
- Convolution and correlation subroutines
- Related-computation subroutines

#### Fourier Transforms

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

### Table 24. List of LAPACK Eigensystem Analysis Subroutines (continued)

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSPGVX^A</td>
<td>DSGPVX^A</td>
<td>“SSPGVX, DSGPVX, CHPGVX, ZHPGVX, SSYGVX, DSYGVX, CHEGVX, and ZHEGVX (Eigenvalues and, Optionally, the Eigenvectors of a Positive Definite Real Symmetric or Complex Hermitian Generalized Eigenproblem)” on page 1008</td>
</tr>
<tr>
<td>CHPGVX^A</td>
<td>ZHPGVX^A</td>
<td></td>
</tr>
<tr>
<td>SSYGVX^A</td>
<td>DSYGVX^A</td>
<td></td>
</tr>
<tr>
<td>CHEGVX^A</td>
<td>ZHEGVX^A</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sspgvx^A</td>
<td>LAPACKE_dspgvlx^A</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chpgvx^A</td>
<td>LAPACKE_zhpghx^A</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssygvx^A</td>
<td>LAPACKE_dsyglx^A</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chegvx^A</td>
<td>LAPACKE_zhegvlx^A</td>
<td></td>
</tr>
<tr>
<td>LAPACKE dspgvx^A</td>
<td>LAPACKE chpgvx^A</td>
<td></td>
</tr>
<tr>
<td>LAPACKE ssygvx^A</td>
<td>LAPACKE dsgvx^A</td>
<td></td>
</tr>
<tr>
<td>LAPACKE chegvx^A</td>
<td>LAPACKE zhegvx^A</td>
<td></td>
</tr>
<tr>
<td>^A LAPACK</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 25. List of Fourier Transform Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCFTD</td>
<td>DCFTD</td>
<td>“SCFTD and DCFTD (Multidimensional Complex Fourier Transform)” on page 1036</td>
</tr>
<tr>
<td>SRCFTD</td>
<td>DRCFTD</td>
<td>“SRCFTD and DRCFTD (Multidimensional Real-to-Complex Fourier Transform)” on page 1044</td>
</tr>
<tr>
<td>SCRFTD</td>
<td>DCRFTD</td>
<td>“SCRFTD and DCRFTD (Multidimensional Complex-to-Real Fourier Transform)” on page 1052</td>
</tr>
<tr>
<td>SCFT^§ SCFTD^ND</td>
<td>DCFT^§</td>
<td>“SCFT and DCFT (Complex Fourier Transform)” on page 1060</td>
</tr>
<tr>
<td>SRCFT^§</td>
<td>DRCFT^§</td>
<td>“SRCFT and DRCFT (Real-to-Complex Fourier Transform)” on page 1069</td>
</tr>
<tr>
<td>SCRFT^§</td>
<td>DCRFT^§</td>
<td>“SCRFT and DCRFT (Complex-to-Real Fourier Transform)” on page 1077</td>
</tr>
<tr>
<td>SCOSF SCOSFT^ND</td>
<td>DCOSF</td>
<td>“SCOSF and DCOSF (Cosine Transform)” on page 1085</td>
</tr>
<tr>
<td>SSINF</td>
<td>DSINF</td>
<td>“SSINF and DSINF (Sine Transform)” on page 1095</td>
</tr>
<tr>
<td>SCFT2^§ SCFT2P^ND</td>
<td>DCFT2^§</td>
<td>“SCFT2 and DCFT2 (Complex Fourier Transform in Two Dimensions)” on page 1101</td>
</tr>
<tr>
<td>SRCFT2^§</td>
<td>DRCFT2^§</td>
<td>“SRCFT2 and DRCFT2 (Real-to-Complex Fourier Transform in Two Dimensions)” on page 1105</td>
</tr>
<tr>
<td>SCRFT2^§</td>
<td>DCRFT2^§</td>
<td>“SCRFT2 and DCRFT2 (Complex-to-Real Fourier Transform in Two Dimensions)” on page 1113</td>
</tr>
<tr>
<td>SCFT3^§ SCFT3P^ND</td>
<td>DCFT3^§</td>
<td>“SCFT3 and DCFT3 (Complex Fourier Transform in Three Dimensions)” on page 1123</td>
</tr>
<tr>
<td>SRCFT3^§</td>
<td>DRCFT3^§</td>
<td>“SRCFT3 and DRCFT3 (Real-to-Complex Fourier Transform in Three Dimensions)” on page 1130</td>
</tr>
</tbody>
</table>
Table 25. List of Fourier Transform Subroutines (continued)

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCRFT3§</td>
<td>DCRFT3§</td>
<td>“SCRFT3 and DCRFT3 (Complex-to-Real Fourier Transform in Three Dimensions)” on page 1137</td>
</tr>
</tbody>
</table>

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

ND 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 26. List of Convolution and Correlation Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCON§</td>
<td>SCOND</td>
<td>“SCON and SCOR (Convolution or Correlation of One Sequence with One or More Sequences)” on page 1145</td>
</tr>
<tr>
<td>SCOR§</td>
<td>SCORD</td>
<td>“SCOND and SCORD (Convolution or Correlation of One Sequence with Another Sequence Using a Direct Method)” on page 1151</td>
</tr>
<tr>
<td>SCONF</td>
<td>SCORF</td>
<td>“SCONF and SCORF (Convolution or Correlation of One Sequence with One or More Sequences Using the Mixed-Radix Fourier Method)” on page 1157</td>
</tr>
<tr>
<td>SDCON</td>
<td>DDCON, DDCOR</td>
<td>“SDCON, DDCON, SDCOR, and DDCOR (Convolution or Correlation with Decimated Output Using a Direct Method)” on page 1167</td>
</tr>
<tr>
<td>SACOR§</td>
<td>SACORF</td>
<td>“SACOR (Autocorrelation of One or More Sequences)” on page 1172</td>
</tr>
</tbody>
</table>

§ 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 27. List of Related-Computation Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPOLY</td>
<td>DPOLY</td>
<td>“SPOLY and DPOLY (Polynomial Evaluation)” on page 1183</td>
</tr>
<tr>
<td>SIZC</td>
<td>DIZC</td>
<td>“SIZC and DIZC (I-th Zero Crossing)” on page 1186</td>
</tr>
<tr>
<td>STREC</td>
<td>DTREC</td>
<td>“STREC and DTREC (Time-Varying Recursive Filter)” on page 1189</td>
</tr>
</tbody>
</table>
### Sorting and Searching

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

#### Table 28. List of Sorting and Searching Subroutines

<table>
<thead>
<tr>
<th>Integer Subroutine</th>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISORT</td>
<td>SSORT</td>
<td>DSORT</td>
<td>&quot;ISORT, SSORT, and DSORT (Sort the Elements of a Sequence)&quot; on page 1204</td>
</tr>
<tr>
<td>ISORTEX</td>
<td>SSORTEX</td>
<td>DSORTEX</td>
<td>&quot;ISORTEX, SSORTEX, and DSORTEX (Sort the Elements of a Sequence and Note the Original Element Positions)&quot; on page 1206</td>
</tr>
<tr>
<td>ISORTS</td>
<td>SSORTS</td>
<td>DSORTS</td>
<td>&quot;ISORTS, SSORTS, and DSORTS (Sort the Elements of a Sequence Using a Stable Sort and Note the Original Element Positions)&quot; on page 1209</td>
</tr>
<tr>
<td>IBSRCH</td>
<td>SBSRCH</td>
<td>DBSRCH</td>
<td>&quot;IBSRCH, SBSRCH, and DBSRCH (Binary Search for Elements of a Sequence X in a Sorted Sequence Y)&quot; on page 1213</td>
</tr>
<tr>
<td>ISSRCH</td>
<td>SSSRCH</td>
<td>DSSRCH</td>
<td>&quot;ISSRCH, SSSRCH, and DSSRCH (Sequential Search for Elements of a Sequence X in the Sequence Y)&quot; on page 1217</td>
</tr>
</tbody>
</table>

### Interpolation

The interpolation subroutines provide the capabilities of doing polynomial interpolation, local polynomial interpolation, and both one- and two-dimensional cubic spline interpolation (Table 29).

#### Table 29. List of Interpolation Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPINT</td>
<td>DPINT</td>
<td>&quot;SPINT and DPINT (Polynomial Interpolation)&quot; on page 1225</td>
</tr>
<tr>
<td>STPINT</td>
<td>DTPINT</td>
<td>&quot;STPINT and DTPINT (Local Polynomial Interpolation)&quot; on page 1228</td>
</tr>
<tr>
<td>SCSINT</td>
<td>DCSINT</td>
<td>&quot;SCSINT and DCSINT (Cubic Spline Interpolation)&quot; on page 1232</td>
</tr>
<tr>
<td>SCSIN2</td>
<td>DCSIN2</td>
<td>&quot;SCSIN2 and DCSIN2 (Two-Dimensional Cubic Spline Interpolation)&quot; on page 1237</td>
</tr>
</tbody>
</table>
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 30. List of Numerical Quadrature Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPTNQ</td>
<td>DPTNQ</td>
<td>“SPTNQ and DPTNQ (Numerical Quadrature Performed on a Set of Points)” on page 1247</td>
</tr>
<tr>
<td>SGLNQ</td>
<td>DGLNQ</td>
<td>“SGLNQ and DGLNQ (Numerical Quadrature Performed on a Function Using Gauss-Legendre Quadrature)” on page 1250</td>
</tr>
<tr>
<td>SGLNQ2</td>
<td>DGLNQ2</td>
<td>“SGLNQ2 and DGLNQ2 (Numerical Quadrature Performed on a Function Over a Rectangle Using Two-Dimensional Gauss-Legendre Quadrature)” on page 1253</td>
</tr>
<tr>
<td>SGLGQ</td>
<td>DGLGQ</td>
<td>“SGLGQ and DGLGQ (Numerical Quadrature Performed on a Function Using Gauss-Laguerre Quadrature)” on page 1259</td>
</tr>
<tr>
<td>SGRAQ</td>
<td>DGRAQ</td>
<td>“SGRAQ and DGRAQ (Numerical Quadrature Performed on a Function Using Gauss-Rational Quadrature)” on page 1262</td>
</tr>
<tr>
<td>SGHMQ</td>
<td>DGHMQ</td>
<td>“SGHMQ and DGHMQ (Numerical Quadrature Performed on a Function Using Gauss-Hermite Quadrature)” on page 1266</td>
</tr>
</tbody>
</table>

† 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 using one of the following algorithms:
• SIMD-oriented Mersenne Twister algorithm
• Multiplicative congruential methods
• Polar methods
• Tausworthe exclusive-or algorithm

Table 31. List of Random Number Generation Initialization Subroutines

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITRNG</td>
<td>“INITRNG (Initialize Random Number Generators)” on page 1271</td>
</tr>
</tbody>
</table>

Table 32. List of Random Number Generation Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SURNG</td>
<td>DURNG</td>
<td>“SURNG and DURNG (Generate a Vector of Uniformly Distributed Pseudo-Random Numbers)” on page 1276</td>
</tr>
<tr>
<td>SNRNG</td>
<td>DNRNG</td>
<td>“SNRNG and DNRNG (Generate a Vector of Normally Distributed Pseudo-Random numbers)” on page 1279</td>
</tr>
<tr>
<td>SURAND</td>
<td>DURAND</td>
<td>“SURAND and DURAND (Generate a Vector of Uniformly Distributed Random Numbers)” on page 1283</td>
</tr>
<tr>
<td>SNRAND</td>
<td>DNRAND</td>
<td>“SNRAND and DNRAND (Generate a Vector of Normally Distributed Random Numbers)” on page 1286</td>
</tr>
</tbody>
</table>
Table 32. List of Random Number Generation Subroutines (continued)

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SURXOR§</td>
<td>DURXOR§</td>
<td>“SURXOR and DURXOR (Generate a Vector of Long Period Uniformly Distributed Random Numbers)” on page 1289</td>
</tr>
</tbody>
</table>

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

Utilities

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

Table 33. List of Utility Subroutines

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINFO</td>
<td>“EINFO (ESSL Error Information-Handler Subroutine)” on page 1296</td>
</tr>
<tr>
<td>ERRSAV</td>
<td>“ERRSAV (ESSL ERRSAV Subroutine)” on page 1299</td>
</tr>
<tr>
<td>ERRSET</td>
<td>“ERRSET (ESSL ERRSET Subroutine)” on page 1300</td>
</tr>
<tr>
<td>ERRSTR</td>
<td>“ERRSTR (ESSL ERRSTR Subroutine)” on page 1302</td>
</tr>
<tr>
<td>IVSSET§</td>
<td>Set the Vector Section Size (VSS) for the ESSL/370 Scalar Library</td>
</tr>
<tr>
<td>IEVOPS§</td>
<td>Set the Extended Vector Operations Indicator for the ESSL/370 Scalar Library</td>
</tr>
<tr>
<td>IESSL</td>
<td>“IESSL (Determine the Level of ESSL Installed)” on page 1303</td>
</tr>
<tr>
<td>SETGPUS</td>
<td>“SETGPUS (Set the Number of GPUs and Identify Which GPUs ESSL Should Use)” on page 1305</td>
</tr>
<tr>
<td>STRIDE</td>
<td>“STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)” on page 1307</td>
</tr>
<tr>
<td>DSRSM</td>
<td>“DSRSM (Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode)” on page 1323</td>
</tr>
<tr>
<td>DGKTRN</td>
<td>“DGKTRN (For a General Sparse Matrix, Convert Between Diagonal-Out and Profile-In Skyline Storage Mode)” on page 1327</td>
</tr>
<tr>
<td>DSKTRN</td>
<td>“DSKTRN (For a Symmetric Sparse Matrix, Convert Between Diagonal-Out and Profile-In Skyline Storage Mode)” on page 1332</td>
</tr>
</tbody>
</table>

§ 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

Planning your ESSL program involves several tasks.

- “Selecting an ESSL Subroutine”
- “Avoiding Conflicts with Internal ESSL Routine Names That are Exported” on page 48
- “Setting Up Your Data” on page 48
- “Setting Up Your ESSL Calling Sequences” on page 50
- “Using Auxiliary Storage in ESSL” on page 51
- “Providing a Correct Transform Length to ESSL” on page 58
- “Getting the Best Accuracy” on page 63
- “Getting the Best Performance” on page 65
- “Dealing with Errors when Using ESSL” on page 67

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.

What ESSL Library Do You Want to Use?

ESSL provides serial and SMP libraries, as described here. (For additional details about using these libraries, see Chapter 4, “Coding Your Program,” on page 133 and Chapter 5, “Processing Your Program,” on page 185.)

Serial and SMP Libraries Provided by ESSL

ESSL provides the following serial library:

- ESSL Serial Libraries, which support the following environments:
  - 32-bit integer, 32-bit pointer environment (AIX only)
  - 32-bit integer, 64-bit pointer environment
  - 64-bit integer, 64-bit pointer environment

These serial libraries provide thread-safe versions of the ESSL subroutines. You may choose to use these libraries if you decide to develop your own multithreaded programs that call the thread-safe ESSL subroutines.

ESSL also provides the following SMP libraries:

- ESSL SMP Libraries, which support the following environments:
  - 32-bit integer, 32-bit pointer environment (AIX only)
  - 32-bit integer, 64-bit pointer environment
  - 64-bit integer, 64-bit pointer environment
- ESSL SMP CUDA Library, which supports the following environment:
  - IBM Power System S822LC (8335-GTB) servers with NVIDIA P100 GPUs and
  - IBM Power System S822LC (8335-GTA) servers with NVIDIA K80 GPUs
    running Red Hat Enterprise Linux 7.3 (RHEL7.3) (little endian mode).
The ESSL SMP CUDA library provides the following options for a subset of ESSL subroutines:

- Use one or more NVIDIA GPUs
- Use one or more NVIDIA GPUs and POWER8 CPUs

The GPU enabled subroutines that the ESSL SMP CUDA Library contains are listed in "Using the ESSL SMP CUDA Library" on page 43.

These ESSL SMP libraries and ESSL SMP CUDA library provide thread-safe versions of the ESSL subroutines, and in addition, a subset of these subroutines are also multithreaded versions; that is, they support the shared memory parallel processing programming model.

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. The maximum number of threads supported by ESSL is 512.

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.

The multithreaded subroutines in the ESSL SMP Libraries are listed in "Multithreaded Subroutines Provided by ESSL" on page 38.

Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL

Some of the subroutines in the libraries provided by ESSL use SIMD algorithms, as explained in the following sections.

SIMD Algorithms on VSX-Enabled Processors

A subset of ESSL subroutines use SIMD algorithms that use the VSX unit on VSX enabled processors. These subroutines need to use the vector load and store instructions to effectively utilize the VSX unit. Alignment requirements for the SIMD algorithms are described in Table 34 on page 33 and Table 35 on page 33.

See Table 36 on page 34 for a list of the ESSL subroutines that automatically use SIMD algorithms when the appropriate alignment restrictions (as described in Table 34 on page 33 and Table 35 on page 33) are met.

Note: For Fourier Transform and Fourier Method Convolution and Correlation subroutines, if you choose to have ESSL calculate the size of auxiliary storage (see "Who Do You Want to Calculate the Size of Auxiliary Storage? You or ESSL?" on page 53), you must pass all array arguments with the same alignment as those passed during the initialization and computation calls. Because of this, it is recommended that you use the processor-independent formulas.
**Table 34. VSX Alignment Requirements for SIMD Algorithms in Linear Algebra Subroutines**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Vector and Matrix Alignment</th>
<th>Vector Stride</th>
<th>Leading Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long-precision real</td>
<td>Quadword and doubleword</td>
<td>Varies depending on the type of subroutine:</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 For vector-scalar linear algebra subroutines</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any For matrix-vector linear algebra subprograms</td>
<td></td>
</tr>
<tr>
<td>Short-precision real</td>
<td>Doubleword and singleword</td>
<td>Varies depending on the type of subroutine:</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 For vector-scalar linear algebra subroutines</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any For matrix-vector linear algebra subprograms</td>
<td></td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Quadword</td>
<td>Any</td>
<td>Any</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Doubleword</td>
<td>Varies depending on the type of subroutine:</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 For vector-scalar linear algebra subroutines</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any For matrix-vector linear algebra subprograms</td>
<td></td>
</tr>
</tbody>
</table>

**Note:**
1. As long as the alignment requirements described in this table are met, you do not have to change your existing application programs that call ESSL to take advantage of the increased performance produced by the SIMD subroutines. However, you will obtain optimal performance for these subroutines when the following additional conditions are met:
   - Vectors and matrices are quadword aligned.
   - LDAs are multiples of 2 for real long-precision matrices.
   - LDAs are multiples of 4 for real short-precision matrices.
   - LDAs are multiples of 2 for complex short-precision matrices.
   - Stride is 1 for vectors.
2. If the alignment restrictions in the table are not met, in some cases attention message 2610 will be issued. The default behavior for message 2610 is for the message to be suppressed. To change the default behavior, see “ERRSET (ESSL ERRSET Subroutine)” on page 1300.

**Table 35. VSX Alignment Requirements for SIMD Algorithms in Fourier Transform Subroutines and Convolution and Correlation Subroutines**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Vector and Matrix Alignment</th>
<th>Stride Between Elements Within Sequence</th>
<th>Stride Between Sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long-precision real</td>
<td>Quadword (see Notes 1 on page 34 and 2 on page 34)</td>
<td>1 (see Note 3 on page 34)</td>
<td>Multiple of 2 (see Note 3 on page 34)</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>Doubleword</td>
<td>1 (see Note 3 on page 34)</td>
<td>Multiple of 4 (see Note 3 on page 34)</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Quadword</td>
<td>Any</td>
<td>Any</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Doubleword</td>
<td>1</td>
<td>Multiple of 2 (see Note 3 on page 34)</td>
</tr>
</tbody>
</table>
### Table 35. VSX Alignment Requirements for SIMD Algorithms in Fourier Transform Subroutines and Convolution and Correlation Subroutines (continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Vector and Matrix Alignment</th>
<th>Stride Between Elements Within Sequence</th>
<th>Stride Between Sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notes:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.</td>
<td>AUX1 must be aligned on a quadword boundary.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>AUX and AUX2 must either be aligned on a quadword boundary or dynamically allocated.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>For <code>_COSF</code> and <code>_SINF</code>, the stride between elements within a sequence and the stride between sequences can have any value.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>As long as the alignment requirements described in this table are met, you do not have to change your existing application programs that call ESSL to take advantage of the increased performance produced by the SIMD subroutines. However, some subroutines require separate calls for initialization and computation, and it can occur that the alignment of an array meets the requirements during initialization but does not meet the requirements during computation. When this happens, in some cases one of the following happens:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Error 2152 will be issued and your program will terminate. If you want your program to continue processing, use ERRSET with an ESSL error exit routine, ENOTRM, to make error 2152 recoverable</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Error 2211 will be issued and your program will terminate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>If the alignment restrictions in this table are not met, in some cases one or more of the following attention messages will be issued:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 2610</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 2611</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 2612</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The default behavior for these messages is to be suppressed. To change the default behavior, see “ERRSET (ESL ERRSET Subroutine)” on page 1300.

### Table 36. ESSL Subroutines that Automatically Use SIMD Algorithms When Alignment Restrictions are Met on VSX-enabled Processors

<table>
<thead>
<tr>
<th>Subroutine Names</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vector-Scalar Linear Algebra Subprograms (See Note)</strong></td>
</tr>
<tr>
<td>ISAMAX, IDAMAX, ICAMAX, IZAMAX</td>
</tr>
<tr>
<td>ISAMIN, IDAMIN</td>
</tr>
<tr>
<td>ISMAX, IDMAX</td>
</tr>
<tr>
<td>ISMIN, IDMIN</td>
</tr>
<tr>
<td>SASUM, DASUM, SCASUM, DZASUM</td>
</tr>
<tr>
<td>SAXPY, DAXPY, CAXPY, ZAXPY</td>
</tr>
<tr>
<td>SCOPY, DCOPY, CCOPY, ZCOPY</td>
</tr>
<tr>
<td>SDOT, DDOT, CDOTU, ZDOTU, CDOTC, ZDTC</td>
</tr>
<tr>
<td>DNRM2, DZNRM2</td>
</tr>
<tr>
<td>DNORM2, ZNORM2</td>
</tr>
<tr>
<td>SROT, DROT, CROT, ZROT, CSROT, ZDROT</td>
</tr>
<tr>
<td>SSCAL, DSCAL, CSCAL, ZSCAL, CSSCAL, ZDSCAL</td>
</tr>
<tr>
<td>SSWAP, DSWAP, CSWAP, ZSWAP</td>
</tr>
<tr>
<td>SVEA, DVEA, CVEA, ZVEA</td>
</tr>
<tr>
<td>SVES, DVES, CVEES, ZVES</td>
</tr>
<tr>
<td>SVEM, DVEM, CVEEM</td>
</tr>
<tr>
<td>SYAX, DYAX, CYAX, ZYAX, CSYAX, ZDYAX</td>
</tr>
<tr>
<td>SZAXPY, DZAXPY, CZAXPY, ZZAXPY</td>
</tr>
</tbody>
</table>

ESL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
Table 36. ESSL Subroutines that Automatically Use SIMD Algorithms When Alignment Restrictions are Met on VSX-enabled Processors (continued)

<table>
<thead>
<tr>
<th>Subroutine Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix-Vector Linear Algebra Subprograms (See Note):</td>
</tr>
<tr>
<td>SGEV, DGEMV, CGEMV, ZGEMV</td>
</tr>
<tr>
<td>SGER, DGER, CGERU, ZGERU, CGERC, ZGERC</td>
</tr>
<tr>
<td>SSPPM, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV</td>
</tr>
<tr>
<td>SSPR, DSPR, CHPR, ZHPR, SSYR, DSYR, CHER, ZHER</td>
</tr>
<tr>
<td>SSPR2, DSPR2, CHPR2, ZHPR2, SSYR2, DSYR2, CHER2, ZHER2</td>
</tr>
<tr>
<td>STRMV, DTRMV, CTRMV, ZTRMV</td>
</tr>
<tr>
<td>STPVM, DTPMV, CTPMV, ZTPMV</td>
</tr>
<tr>
<td>STRSV, DTRSV, CTRSV, ZTRSV, STPSV, DTPSV, CTPSV, ZTPSV</td>
</tr>
<tr>
<td>Matrix Operations (See Note):</td>
</tr>
<tr>
<td>SGEUL, DGEMUL, CGEMUL, ZGEMUL</td>
</tr>
<tr>
<td>SGEMM, DGEMM, CGEMM, ZGEMM</td>
</tr>
<tr>
<td>SSYM, DSYM, CSYM, ZSYM, CHEM, ZHEM</td>
</tr>
<tr>
<td>STRMM, DTRMM, CTRMM, ZTRMM</td>
</tr>
<tr>
<td>STRSM, DTRSM, CTRSM, ZTRSM</td>
</tr>
<tr>
<td>SSYRK, DSYRK, CSYRK, ZSYRK, CHER, ZHERK</td>
</tr>
<tr>
<td>SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, ZHER2K</td>
</tr>
<tr>
<td>SGETMI, DGETMI, CGETMI, ZGETMI, CGECMI, ZGECMI</td>
</tr>
<tr>
<td>SGETMO, DGETMO, CGETMO, ZGETMO, CGECMO, ZGECMO</td>
</tr>
<tr>
<td>Fourier Transforms:</td>
</tr>
<tr>
<td>SCFTD, DCFTD</td>
</tr>
<tr>
<td>SRCFTD, DRCFTD</td>
</tr>
<tr>
<td>SCFT, DCFT</td>
</tr>
<tr>
<td>SRCFT, DRCFT</td>
</tr>
<tr>
<td>SCOSF, DCOSF</td>
</tr>
<tr>
<td>SINF, DSINF</td>
</tr>
<tr>
<td>SCFT2, DCFT2</td>
</tr>
<tr>
<td>SRCFT2, DRCFT2</td>
</tr>
<tr>
<td>SCFT2, DRCFT2</td>
</tr>
<tr>
<td>SCFT3, DCFT3</td>
</tr>
<tr>
<td>SRCFT3, DRCFT3</td>
</tr>
<tr>
<td>Convolutions and Correlations:</td>
</tr>
<tr>
<td>SCONF, SCORE, SACORF</td>
</tr>
<tr>
<td>Random Number Generation:</td>
</tr>
<tr>
<td>SURNG, DURNG</td>
</tr>
<tr>
<td>SNRNG, DRNG</td>
</tr>
</tbody>
</table>

Note: Many of the dense and banded linear algebraic equations and eigensystem analysis subroutines make one or more calls to the vector-scalar, matrix-vector linear algebra, and matrix operation subroutines listed in this table, and therefore they indirectly use SIMD algorithms.

**SIMD Algorithms on POWER 6 Altivec-Enabled Processors**

A subset ESSL subroutines use SIMD algorithms that use the Altivec unit on certain processors for short-precision real and short-precision complex subroutines. These subroutines need to use the vector load and store instructions to use the
AltiVec unit effectively. Alignment requirements for the SIMD algorithms are described in Table 37 and Table 38.

See Table 39 on page 37 for a list of the ESSL subroutines that automatically use SIMD algorithms when the appropriate alignment restrictions (as described in Table 37 and Table 38) are met.

**Note:** For Fourier Transform and Fourier Method Convolution and Correlation subroutines, if you choose to have ESSL calculate the size of auxiliary storage (see “Who Do You Want to Calculate the Size of Auxiliary Storage? You or ESSL?” on page 53), you must pass all array arguments with the same alignment as those passed during the initialization and computation calls. Because of this, it is recommended that you use the processor-independent formulas.

**Table 37. AltiVec-Enabled Processor Alignment Restrictions for SIMD Algorithms in Linear Algebra Subroutines**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Vector and Matrix Alignment</th>
<th>Vector Stride</th>
<th>Leading Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Singleword</td>
<td>Varies depending on the type of subroutine:</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>For vector-scalar linear algebra subroutines</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>For matrix-vector linear algebra subprograms</td>
<td></td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Doubleword</td>
<td>Varies depending on the type of subroutine:</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>For vector-scalar linear algebra subroutines</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>For matrix-vector linear algebra subprograms</td>
<td></td>
</tr>
</tbody>
</table>

**Note:**

1. As long as the alignment requirements described in this table are met, you do not have to change your existing application programs that call ESSL to take advantage of the increased performance produced by the AltiVec-enabled subroutines. However, you will obtain optimal performance for these subroutines when the following additional conditions are met:
   - Vectors and matrices are quadword aligned.
   - LDAs are multiples of 4 for real matrices.
   - LDAs are multiples of 2 for complex matrices.
   - Stride is 1 for real and complex vectors.

2. If the alignment restrictions in the table are not met, in some cases attention message 2610 will be issued. The default behavior for message 2610 is for the message to be suppressed. To change the default behavior, see “ERRSET (ESSL ERRSET Subroutine)” on page 1300.

**Table 38. AltiVec-Enabled Processor Alignment Restrictions for SIMD Algorithms in Fourier Transform and Fourier Method Convolution and Correlation Subroutines**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Vector and Matrix Alignment</th>
<th>Stride Between Elements Within Sequence</th>
<th>Stride Between Sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Quadword</td>
<td>1 (see Note 3 on page 37)</td>
<td>Multiple of 4 (see Note 5 on page 37)</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Quadword</td>
<td>1</td>
<td>Multiple of 2 (see Note 5 on page 37)</td>
</tr>
</tbody>
</table>
Table 38. AltiVec-Enabled Processor Alignment Restrictions for SIMD Algorithms in Fourier Transform and Fourier Method Convolution and Correlation Subroutines (continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Vector and Matrix Alignment</th>
<th>Stride Between Elements Within Sequence</th>
<th>Stride Between Sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long-precision real</td>
<td>Quadword (see Notes [1] and [2])</td>
<td>1</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

Note:
1. AUX1 must be aligned on a quadword boundary.
2. AUX and AUX2 must either be aligned on a quadword boundary or dynamically allocated.
3. For SCOSF and SSINF, the stride between elements within a sequence and the stride between sequences can have any value.
4. As long as the alignment requirements described in this table are met, you do not have to change your existing application programs that call ESSL to take advantage of the increased performance produced by the AltiVec-enabled subroutines. However, some subroutines require separate calls for initialization and computation, and it can occur that the alignment of an array meets the requirements during initialization but does not meet the requirements during computation. When this happens, in some cases error 2211 will be issued and your program will terminate.
5. If the alignment restrictions in the table are not met, one or more of the following attention messages will be issued:
   • 2610
   • 2611
   • 2612

The default behavior for these messages is to be suppressed. To change the default behavior, see “ERRSET (ESSL ERRSET Subroutine)” on page 1300.

Table 39. ESSL Subroutines that Automatically Use SIMD Algorithms When Alignment Restrictions are Met on POWER 6 AltiVec-Enabled Processors

<table>
<thead>
<tr>
<th>Subroutine Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector-Scalar Linear Algebra Subprograms</td>
</tr>
<tr>
<td>ISAMAX, ICAMAX</td>
</tr>
<tr>
<td>ISAMIN</td>
</tr>
<tr>
<td>ISMAX</td>
</tr>
<tr>
<td>ISMIN</td>
</tr>
<tr>
<td>SASUM, SCASUM</td>
</tr>
<tr>
<td>SAXPY</td>
</tr>
<tr>
<td>SDOT, CDOTU, CDOTC</td>
</tr>
<tr>
<td>SROT, CROT, CSROT</td>
</tr>
<tr>
<td>SSCAL, CSCAL, CSSCAL</td>
</tr>
<tr>
<td>SSWAP, CSWAP</td>
</tr>
<tr>
<td>SVEA, CVEA</td>
</tr>
<tr>
<td>SVES, CVES</td>
</tr>
<tr>
<td>SVEM,</td>
</tr>
<tr>
<td>SYAX, CYAX, CSYAX</td>
</tr>
<tr>
<td>SZAXPY, CZAXPY</td>
</tr>
</tbody>
</table>
Table 39. ESSL Subroutines that Automatically Use SIMD Algorithms When Alignment Restrictions are Met on POWER 6 Altivec-Enabled Processors (continued)

<table>
<thead>
<tr>
<th>Subroutine Names</th>
<th>Matrix-Vector Linear Algebra Subprograms</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGER, CGERU, CGERC</td>
<td>SGEMV, SSYMV</td>
</tr>
<tr>
<td>SSPR, CHPR, SSYR, CHER</td>
<td>SSPR2, CHPR2, SSYR2, CHER2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Matrix Operation</th>
<th>SGEADD, CGEADD</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGESUB, CGESUB</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fourier Transforms:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCFTD</td>
</tr>
<tr>
<td>SRCFTD</td>
</tr>
<tr>
<td>SCRFTD</td>
</tr>
<tr>
<td>SCFT</td>
</tr>
<tr>
<td>SRCFT</td>
</tr>
<tr>
<td>SCOSF</td>
</tr>
<tr>
<td>SSINF</td>
</tr>
<tr>
<td>SCFT2</td>
</tr>
<tr>
<td>SRCFT2</td>
</tr>
<tr>
<td>SCRFT2</td>
</tr>
<tr>
<td>SCFT3</td>
</tr>
<tr>
<td>SRCFT3</td>
</tr>
<tr>
<td>SCRFT3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Convolutions and Correlations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCONF, SCORF</td>
</tr>
<tr>
<td>SACORF</td>
</tr>
</tbody>
</table>

**Note:**
1. Many of the dense and banded linear algebraic equations and eigensystem analysis subroutines make one or more calls to the vector-scalar, matrix-vector linear algebra, and matrix operation subroutines listed in this table, and therefore they indirectly use SIMD algorithms.

**Multithreaded Subroutines Provided by ESSL**

Table 40 on page 39 lists the multithreaded subroutines provided by ESSL and also indicates which of those subroutines use SIMD algorithms.
<table>
<thead>
<tr>
<th>Subroutine Category</th>
<th>Multithreaded Subroutine</th>
<th>Does this subroutine also use SIMD algorithms on VSX-enabled processors?</th>
<th>Does this short-precision subroutine also use SIMD algorithms on AltiVec-enabled processors?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector-Scalar Linear Algebra Subprograms</td>
<td>SASUM, DASUM, SCASUM, DZASUM</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SAXPY, DAXPY, CAXPY, ZAXPY</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SCOPY, DCOPY, CCOPY, ZCOPY</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SDOT, DDOT, CDOTU, ZDOTU, CDOTC, ZDOTC</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SNDOT, DNDOT</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SNORM2, DNORM2, CNORM2, ZNORM2</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SROT, DROT, CROT, ZROT, CSROT, ZDROT</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SSCAL, DSCAL, CSCAL, ZSCAL, CSSCAL, ZDSCAL</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SSWAP, DSWAP, CSWAP, ZSWAP</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SVEA, DVEA, CVEA, ZVEA</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SVES, DVES, CVES, ZVES</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SVEM, DVEM, CVEM, ZVEM</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SYAX, DYAX, CYAX, ZYAX, CSYAX, ZDYAX</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SZAXPY, DZAXPY, CZAXPY, ZZAXPY</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Table 40. Multithreaded Subroutines  (continued)

<table>
<thead>
<tr>
<th>Subroutine Category</th>
<th>Multithreaded Subroutine</th>
<th>Does this subroutine also use SIMD algorithms on VSX-enabled processors?</th>
<th>Does this short-precision subroutine also use SIMD algorithms on AltiVec-enabled processors?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix-Vector Linear Algebra Subprograms</td>
<td>SGEMV, DGEMV, CGEMV, ZGEMV</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SGER, DGER, CGERU, ZGERU, CGERC, ZGERC</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SSPMV, DSPMV, CHPMV, ZHPMV</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SSYMV, DSYMV, CHEMV, ZHEMV</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SSPR, DSPR, CHPR, ZHPR</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SSYR, DSYR, CHER, ZHER</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SSPR2, DSPR2, CHPR2, ZHPR2</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SSYR2, DSYR2, CHER2, ZHER2</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SGBMV, DGBMV</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>CGBMV, ZGBMV</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SSBMV, DSBMV</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>CHBMV, ZHBMV</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>STRMV, DTRMV, CTRMV, ZTRMV</td>
<td>Yes except DTRMV and ZTRMV</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>STPMV, DTPMV, CTPMV, ZTPMV</td>
<td>Yes except DTPMV and ZTPMV</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>STRSV, DTRSV, CTRSV, ZTRSV</td>
<td>Yes except DTRSV and ZTRSV</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>STPSV, DTPSV, CTPSV, ZTPSV</td>
<td>Yes except DTPSV, ZTPSV</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>STBMV, DTBMV</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>CBTMV, ZTBMV</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Sparse Matrix Vector Linear Algebra Subprograms</td>
<td>DSMMX, DSDMX</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Subroutine Category</td>
<td>Multithreaded Subroutine</td>
<td>Does this subroutine also use SIMD algorithms on VSX-enabled processors?</td>
<td>Does this short-precision subroutine also use SIMD algorithms on AltiVec-enabled processors?</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------</td>
<td>--------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Matrix Operations</td>
<td>SGEADD, DGEADD, CGEADD, ZGEADD</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SGESUB, DGESUB, CGESUB, ZGESUB</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SGEMUL, DGEMUL, CGEMUL, ZGEMUL</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SGEMM, DGEMM, CGEMM, ZGEMM</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SSYMM, DSYMM, CSYMM, ZSYMM, CHEMM, ZHEMM</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>STRMM, DTRMM, CTRMM, ZTRMM</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>STRSM, DTRSM, CTRSM, ZTRSM</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SSYRK, DSYRK, CSYRK, ZSYRK, CHERK, ZHERK</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, ZHER2K</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SGETMI, DGETMI, CGETMI, ZGETMI, CGECMI, ZGECMI</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SGETMO, DGETMO, CGETMO, ZGETMO, CGECMO, ZGECMO</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Subroutine Category</td>
<td>Multithreaded Subroutine</td>
<td>Does this subroutine also use SIMD algorithms on VSX-enabled processors?</td>
<td>Does this short-precision subroutine also use SIMD algorithms on AltiVec-enabled processors?</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>--------------------------------</td>
<td>--------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Dense Linear Algebraic Equations</td>
<td>SGESV, DGESV, CGESV, ZGESV</td>
<td>See Note 1 on page 43</td>
<td>See Note 1 on page 43</td>
</tr>
<tr>
<td></td>
<td>SGEF, DGEF, CGEF, ZGEF</td>
<td>See Note 1 on page 43</td>
<td>See Note 1 on page 43</td>
</tr>
<tr>
<td></td>
<td>SGES, DGES, CGES, ZGES</td>
<td>See Note 1 on page 43</td>
<td>See Note 1 on page 43</td>
</tr>
<tr>
<td></td>
<td>SGETRF, DGETRF, CGETRF, ZGETRF</td>
<td>See Note 1 on page 43</td>
<td>See Note 1 on page 43</td>
</tr>
<tr>
<td></td>
<td>SGETRS, DGETRS, CGETRS, ZGETRS</td>
<td>See Note 1 on page 43</td>
<td>See Note 1 on page 43</td>
</tr>
<tr>
<td></td>
<td>SPPSV, DPPSV, CPPSV, ZPPSV</td>
<td>See Note 1 on page 43</td>
<td>See Note 1 on page 43</td>
</tr>
<tr>
<td></td>
<td>SPPE, DPPE, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, DPOE, DPOTRF</td>
<td>See Note 1 on page 43</td>
<td>See Note 1 on page 43</td>
</tr>
<tr>
<td></td>
<td>SPPTRS, DPPTRS, CPPTRS, ZPPTRS</td>
<td>See Note 1 on page 43</td>
<td>See Note 1 on page 43</td>
</tr>
<tr>
<td></td>
<td>SPOSV, DPOSV, CPOSV, ZPOSV</td>
<td>See Note 1 on page 43</td>
<td>See Note 1 on page 43</td>
</tr>
<tr>
<td></td>
<td>SPOSM, DPOSM, CPOSM, ZPOSM</td>
<td>See Note 1 on page 43</td>
<td>See Note 1 on page 43</td>
</tr>
<tr>
<td></td>
<td>SPPFCT, DPPFCT, DPOFCT</td>
<td>See Note 1 on page 43</td>
<td>See Note 1 on page 43</td>
</tr>
<tr>
<td></td>
<td>SPPTRI, DPPTRI, CPPTRI, ZPPTRI, SPPICT, DPPICT, DPOTICT</td>
<td>See Note 1 on page 43</td>
<td>See Note 1 on page 43</td>
</tr>
<tr>
<td></td>
<td>STRI, DTRI, STRTRI, DTRTRI, CRTTRI, ZTRTRI</td>
<td>See Note 1 on page 43</td>
<td>See Note 1 on page 43</td>
</tr>
<tr>
<td>Banded Linear Algebraic Equations</td>
<td>SGBSV, DGBSV, CGBSV, ZGBSV</td>
<td>See Note 1 on page 43</td>
<td>See Note 1 on page 43</td>
</tr>
<tr>
<td></td>
<td>SGBTRS, DGBTRS, CGBTRS, ZGBTRS</td>
<td>See Note 1 on page 43</td>
<td>See Note 1 on page 43</td>
</tr>
<tr>
<td></td>
<td>SPBSV, DPBSV, CPBSV, ZPBSV</td>
<td>See Note 1 on page 43</td>
<td>See Note 1 on page 43</td>
</tr>
<tr>
<td></td>
<td>SPBTRS, DPBTRS, CPBTRS, ZPBTRS</td>
<td>See Note 1 on page 43</td>
<td>See Note 1 on page 43</td>
</tr>
<tr>
<td>Sparse Linear Algebraic Equations</td>
<td>DSRIS</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Linear Least Squares</td>
<td>SGEQRF, DGEQRF, CGEQRF, ZGEQRF</td>
<td>See Note 1 on page 43</td>
<td>See Note 1 on page 43</td>
</tr>
</tbody>
</table>
Table 40. Multithreaded Subroutines  (continued)

<table>
<thead>
<tr>
<th>Subroutine Category</th>
<th>Multithreaded Subroutine</th>
<th>Does this subroutine also use SIMD algorithms on VSX-enabled processors?</th>
<th>Does this short-precision subroutine also use SIMD algorithms on Altivec-enabled processors?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fourier Transforms</td>
<td>SCFTD, SRCFTD, SCRFTD, SCFT, SRCFT, SCRFT, SCFT2, SRCFT2, SCRFT2, SCFT3, SRCFT3, SCRFT3</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>DCFTD, DRCFDTD, DRCFTD, DCFT, DRCFT, DCRFT, DCF2, DCF2, DCRFT2, DCF3, DCRFT3</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Convolution and Correlation</td>
<td>SCOND, SCORD</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SDCON, SDCOR, DDCON, DDCOR</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SCONF, SCRF SACOF</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Note:**

1. Many of the dense and banded linear algebraic equations and eigensystem analysis subroutines make one or more calls to the vector-scalar, matrix-vector linear algebra, and matrix operation subroutines listed in this table, and therefore they indirectly use multiple threads and SIMD algorithms.

2. Your performance may be improved by setting the following environment variables:

   **ESSL for AIX**
   
   export MALLOCMULTIHEAP=true
   
   —and—
   
   export XLSMPOPTS="spins=0:yields=0"

   **ESSL for Linux**
   
   export XLSMPOPTS="spins=0:yields=0"

   For additional information, see the *AIX Performance Management Guide* and the XLF Manuals.

3. The Level 2 Banded BLAS use multiple threads only when the bandwidth is sufficiently large.

4. Multiple threads are used for the factor or inverse computation.

5. DSRIS only uses multiple threads when IPARM(4) = 1 or 2.

---

**Using the ESSL SMP CUDA Library**

The ESSL SMP CUDA 32-bit integer, 64-bit pointer environment library is supported only on IBM Power System S822LC (8335-GTB) servers with NVIDIA P100 GPUs and IBM Power System S822LC (8335-GTA) servers with NVIDIA K80 GPUs running Red Hat Enterprise Linux 7.3 (RHEL7.3) (little endian mode). You can use the ESSL SMP CUDA Library in two ways for the subset of ESSL Subroutines that are GPU-enabled:

- Using NVIDIA GPUs for the bulk of the computation
- Using a hybrid combination of POWER8 CPUs and NVIDIA GPUs
The ESSL SMP CUDA library leverages ESSL BLAS, NVIDIA cuBLAS, and blocking techniques to handle problem sizes larger than the GPU memory size. The algorithms support multiple GPUs and are designed for use in both SMP and MPI applications.

The ESSL SMP CUDA Library contains GPU-enabled versions of the following subroutines:

- SGEMM, DGEMM, CGEMM, and ZGEMM
- SSYMM, DSYMM, CSYMM, ZSYMM, CHEMM, and ZHEMM
- STRMM, DTRMM, CTRMM, and ZTRMM
- STRSM, DTRSM, CTRSM, and ZTRSM
- SSYRK, DSYRK, CSYRK, ZSYRK, CHERK, and ZHERK
- SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, and ZHER2K

**Note:** In the descriptions that follow *host* refers to the Power server and *device* refers to the GPU.

To use the ESSL SMP CUDA library, you must specify only host arrays as arguments and link your applications using -lesslmpcuda (see “Processing Your Program on Linux (little endian mode)” on page 191). If desired, you can change the default behavior of the ESSL SMP CUDA Library using either environment variables or the SETGPUS subroutine, see “ESSL SMP CUDA Library Options” on page 45.

For information on the NVIDIA CUDA support, see the following:

**ESSL Support for NVIDIA GPU Compute Modes**

NVIDIA allows you to use GPU compute modes to control how application threads run on the GPU.

**Restriction:** ESSL requires all visible GPUs to be set to the same compute mode, except for those in PROHIBITED mode, which ESSL ignores.

The NVIDIA compute modes are as follows:

1. **0 DEFAULT**
   - Multiple host threads can use the device at the same time.
   
   ESSL can use one or more visible GPUs on the host. See “ESSL SMP CUDA Library Options” on page 45 for information on the CUDA_VISIBLE_DEVICES environment variable.

2. **2 PROHIBITED**
   - No host thread can use the device.
   
   ESSL does not use any GPUs in PROHIBITED compute mode; it uses only the GPUs in other compute modes. If all GPUs are in PROHIBITED compute mode, ESSL issues attention message 2538-2614 and runs using CPUs only, ignoring the setting of the ESSL_CUDA_HYBRID environment variable. See “ESSL SMP CUDA Library Options” on page 45 for information on the ESSL_CUDA_HYBRID environment variable.

3. **3 EXCLUSIVE_PROCESS**
   - Only one context is allowed per device, usable from multiple threads at a time.
ESSL can use one or more visible GPUs on the host. If the CUDA MPS\(^1\) is being used with more than 1 GPU, you can use the SETGPUS subroutine or the environmental variable CUDA_VISIBLE_DEVICES with the local rank of the MPI tasks to select the different GPUs for MPI tasks that you want ESSL to use. See “ESSL SMP CUDA Library Options” for information on the CUDA_VISIBLE_DEVICES environment variable.

### ESSL SMP CUDA Library Options

The ESSL SMP CUDA Library allows you to control these options:

**Control how many and which GPUs ESSL uses**

By default, ESSL uses all devices. Use the CUDA_VISIBLE_DEVICES environment variable or the SETGPUS subroutine to change this default. The CUDA applications will see only the devices whose index is specified in the CUDA_VISIBLE_DEVICES environmental variable, and the devices are enumerated in the order of the sequence specified. For example, if you have three GPUs defined, 0, 1, 2, you can specify that a CUDA application use only a subset of the GPUs, 1 and 2, using the environmental variable as follows:

```bash
export CUDA_VISIBLE_DEVICES=1,2
```

You can also specify a new order in which your three GPUs are enumerated:

```bash
export CUDA_VISIBLE_DEVICES=2,1,0
```

If you need different MPI tasks to use different GPUs, you can use the SETGPUS subroutine or the environmental variable CUDA_VISIBLE_DEVICES with the local rank of the MPI tasks to ensure each task uses unique GPUs. See “SETGPUS (Set the Number of GPUs and Identify Which GPUs ESSL Should Use)” on page 1305.

In some cases ESSL does not use GPUs:

- The GPU-enabled subroutine is called from within an OpenMP parallel construct (OMP_IN_PARALLEL is true).
- For pre- and post-scaling operations, for example, handling the alpha argument in _TRMM.
- When the problem size is too small to benefit from using GPUs.

### Specifying Whether ESSL Runs in Hybrid Mode

By default, the ESSL SMP CUDA library runs in hybrid mode. Use the ESSL_CUDA_HYBRID environment variable to change this default (valid values are yes or no). The default hybrid mode (ESSL_CUDA_HYBRID=yes) means that the ESSL SMP CUDA Library subroutines can run on both POWER8 CPUs and NVIDIA GPUs.

**Note:** Subroutines SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, and ZHER2K only use the Power8 CPUs for scaling operations.

### Specifying Whether ESSL Pins Host Memory Buffers

By default, ESSL does not pin host memory buffers (ESSL_CUDA_PIN=no). Use the ESSL_CUDA_PIN environment variable to change this default (valid values are yes, no, or pinned).

---

1. NVIDIA CUDA Multi Process Service (MPS) is a feature that allows multiple CUDA processes to share a single GPU context.
If you want ESSL to pin your host memory buffers on entry to
gpu-enabled subroutines and unpin them before returning, specify
ESSL_CUDA_PIN=yes.

Performance might be improved if you pin your host memory buffers used
in the ESSL calling sequences once before any calls to ESSL subroutines. To
pin your host memory buffers use the NVIDIA CUDA subroutine
cudaHostRegister. If you pin your own buffers you should specify
ESSL_CUDA_PIN=pinned.

Note: Host memory buffers that are only partially pinned may lead to
NVIDIA Error 11 from cublasSetMatrixAsync or cublasSetMatrix.

How ESSL Assigns Threads

The ESSL SMP CUDA Library requires at least one OpenMP thread for each GPU
used. If the number of OpenMP threads is less than the number of GPUs, ESSL
issues attention message 2538-2615 and uses the same number of GPUs as there are
OpenMP threads.

ESSL SMP CUDA Library uses the following priorities to assign threads:
• ESSL reserves 1 thread for each GPU used
• Some ESSL subroutines might reserve threads needed to support multiple
  streams
• The remaining threads are used for the CPU, but a subroutine might not run in
  hybrid mode if there are not enough threads left or if the problem size is too
  small.

MPI Applications

There are two ways to use the ESSL SMP CUDA Library with MPI Applications
depending on how the GPUs are used by the local MPI tasks:
• GPUs are not shared, meaning that each MPI task on a node uses unique GPUs.
  You can use the the SETGPU$ subroutine or the environmental variable
  CUDA_VISIBLE_DEVICES with the local rank of the MPI tasks to ensure each
  task uses unique GPUs. See the OMPI_COMM_WORLD_LOCAL_RANK
  description at the following Open MPI URL:
  https://www.open-mpi.org/
• GPUs are shared, meaning that the number of MPI tasks per node oversubscribe
  the GPUs. For this case we recommend you run using the NVIDIA MPS which
  is a runtime service designed to let multiple MPI processes using CUDA run
  concurrently on a single GPU in a way that's transparent to the MPI program.
  NVIDIA MPS supports at most 16 MPI Tasks per GPU, but if you are using
  ESSL, it is recommended that you use Core Affinity and no more tasks than the
  number of cores being used.

If you are sharing GPUs, it's possible that ESSL will be unable to allocate work
space on the GPU. In that case you can reduce the number of MPI tasks per
node or, if possible, increase the number of GPUs being used per node to
eliminate the allocation failures.

If error cudaStreamCreate failed with CUDA message: all CUDA-capable
devices are busy or unavailable occurs when using ESSL with MPI
applications and NVIDIA MPS, confirm that the NVIDIA MPS Daemons are
running on all nodes that the MPI job is using.
You can use SETGPUS (see “SETGPUS (Set the Number of GPUs and Identify Which GPUs ESSL Should Use)” on page 1305) or the environmental variable CUDA_VISIBLE_DEVICES with the local rank of the MPI tasks to inform ESSL which GPUs your MPI Tasks should use.

For best performance, consider increasing the block size you are using to distribute your data across the MPI tasks. Consider block sizes in the range 1024-4096 elements.

**NVIDIA GPU Power Capping**

The ESSL and NVIDIA library subroutines are highly optimized and for some problem sizes your application may exceed the SW Power cap for one or more of the GPUs. If this happens your performance will be degraded because the frequency of the corresponding GPU clock will be reduced because the GPU is consuming too much power.

You can confirm that this is happening by using nvidia-smi to monitor the GPUs while your application is running.

```
nvidia-smi dmon
```

If you wish to adjust the Power Cap Limit follow these steps:

1. Determine the current, default and maximum power limit as follows:
   ```
nvidia-smi -q | grep 'Power Limit'
```

2. Ensure that persistence mode is being used.

3. Increase the SW Power Cap limit for all GPUs as follows, where xxx is the desired value in watts:
   ```
nvidia-smi -pl xxx
```

**Note:** You must increase the power limit and set persistence each time the server is booted.

For additional information, see the following URL:


**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 63. For a description of these data types, see “How Do You Set Up Your Scalar Data?” on page 48.
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 “Function” 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. Internal ESSL routine names that are exported all begin with the ESV prefix. Therefore, it is sufficient for you to avoid using this prefix for your own names.

Setting Up Your Data

There are various items to consider when setting up your scalar and array data.

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. The scalar data types and how you should code them for each programming language are listed in “Coding Your Scalar Data” specific to each language in Chapter 4, “Coding Your Program,” on page 133.

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, “Setting Up Your Data Structures,” on page 75. 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 134. On the other hand, C, and C++ programs store arrays in row-major order. For details on what you can do, see:

- For C, see “Setting Up Arrays in C” on page 155.
- For C++, see “Setting Up Arrays in C++” on page 172.

**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.

For all subroutines running on VSX enabled processors, see “SIMD Algorithms on VSX-Enabled Processors” on page 32.

For short-precision real and short-precision complex subroutines running on POWER6 Altivec-enabled processors, see “SIMD Algorithms on POWER 6 Altivec-Enabled Processors” on page 35.

For information about 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 choose 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, “Setting Up Your Data Structures,” on page 75. You can also find special storage considerations, where applicable, in “Notes” in each subroutine description.

**How Do You Convert from One Storage Mode to Another?**

ESSL provides conversion subroutines and sample programs to help you 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, “Setting Up Your Data Structures,” on page 75. 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 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:

- “Fortran Programs” on page 133
- “C Programs” on page 151
- “C++ Programs” on page 166

### 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: 32-bit or 64-bit integer, 32-bit or 64-bit logical, 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 in “Setting Up Complex Data Types in C” on page 154 and “Using Logical Data in C” on page 155.
  - Short-precision complex and logical data in C++ programs, using the guidelines in “On AIX—Setting Up Short-Precision Complex Data Types If You Are Using the IBM Open Class Complex Mathematics Library in C++” on page 170 and “Using Logical Data in C++” on page 172.
- 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:

\[
\begin{align*}
N & \quad X & \quad \text{INCX} \\
X & = \text{SASUM(} & 4, & A(3), & 2) \\
\end{align*}
\]

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:

\[
\begin{align*}
A & \quad \text{LDA} & \quad \text{TRANSA} & \quad B & \quad \text{LDB} & \quad \text{TRANSB} & \quad C & \quad \text{LDC} & \quad M & \quad N \\
\text{CALL} & \quad \text{SGEADD(} & A(1,4), & 11, & 'N', & B, & 4, & 'N', & C, & 4, & 3) \\
\end{align*}
\]

For more examples of specifying vectors and matrices, see Chapter 3, “Setting Up Your Data Structures,” on page 75.

Using Auxiliary Storage in ESSL
For the ESSL subroutines listed in Table 41, you need to provide extra working storage to perform the computation. It is necessary to understand the use of dynamic allocation for providing auxiliary storage in ESSL and, if dynamic allocation is not an option, how to calculate the amount of auxiliary storage you need by use of formulas or error-handling capabilities provided in ESSL.

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 for details and variations.

Table 41. ESSL Subroutines Requiring Auxiliary Working Storage

<table>
<thead>
<tr>
<th>Subroutine Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Algebra Subprograms:</td>
</tr>
<tr>
<td>D8MTM</td>
</tr>
</tbody>
</table>
Table 41. ESSL Subroutines Requiring Auxiliary Working Storage (continued)

<table>
<thead>
<tr>
<th>Subroutine Names</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Operations:</td>
<td></td>
</tr>
<tr>
<td>_GEMMS</td>
<td></td>
</tr>
<tr>
<td>Dense Linear Algebraic Equations:</td>
<td></td>
</tr>
<tr>
<td>_GEFCD _PPFCD _GEICD _PIICD _POFCD _POICD</td>
<td></td>
</tr>
<tr>
<td>DGEFPA DPPFPA</td>
<td></td>
</tr>
<tr>
<td>Sparse Linear Algebraic Equations:</td>
<td></td>
</tr>
<tr>
<td>DGSF DGSS DGKFS DGKFSPA DSKFS DSKFSPA</td>
<td></td>
</tr>
<tr>
<td>DSRIS DSMCG DSCDG DSMCGC DSDGCG</td>
<td></td>
</tr>
<tr>
<td>Linear Least Squares:</td>
<td></td>
</tr>
<tr>
<td>_GESVF _GELLS</td>
<td></td>
</tr>
<tr>
<td>Fourier Transforms:</td>
<td></td>
</tr>
<tr>
<td>_CFTD _RCFTD _CRFT _COSF _SINF</td>
<td></td>
</tr>
<tr>
<td>SCOSFAPA _CFT2 _RCFT2 _CRFT2 _CFT3</td>
<td></td>
</tr>
<tr>
<td>_RCFT3 _CRFT3 SCFTAPA SCFT2APA SCFT3APA</td>
<td></td>
</tr>
<tr>
<td>Convolutions and Correlations:</td>
<td></td>
</tr>
<tr>
<td>SCONF SCORF SACORF</td>
<td></td>
</tr>
<tr>
<td>Related Computations:</td>
<td></td>
</tr>
<tr>
<td>_WLEV</td>
<td></td>
</tr>
<tr>
<td>Interpolation:</td>
<td></td>
</tr>
<tr>
<td>_TPINT _CSIN2</td>
<td></td>
</tr>
<tr>
<td>Random Number Generation:</td>
<td></td>
</tr>
<tr>
<td>_NRAND</td>
<td></td>
</tr>
<tr>
<td>Utilities:</td>
<td></td>
</tr>
<tr>
<td>DGKTRN DSKTRN</td>
<td></td>
</tr>
</tbody>
</table>

^ 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.
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 \textit{aux} work area in the \textit{naux} argument.

Who Do You Want to Calculate the Size of Auxiliary Storage? 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 \textit{sufficient values} for your current and future needs. Use them if \textit{ease of migration} to future machines and future releases of ESSL is your primary concern. For details, see “How Do You Calculate the Size of Auxiliary Storage Using the Formulas?”
- Use the ESSL error-handling facilities to return to you a \textit{minimum value} for the particular processor you are currently running on. (Values vary by platform.) Use this approach if \textit{conserving storage} is your primary concern. For details, see “How Do You Get ESSL to Calculate the Size of Auxiliary Storage Using ESSL Error Handling?”

How Do You Calculate the Size of Auxiliary Storage Using the Formulas?

The formulas provided for calculating \textit{naux} indicate a \textit{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:

- \textbf{Simple formulas}
  These are given in the \textit{naux} argument syntax descriptions. In general, these formulas result in the minimum required value, but, in a few cases, they provide overestimates.

- \textbf{Processor-independent formulas}
  These are given separately in each 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 \textit{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 of Auxiliary Storage Using ESSL Error Handling?

When getting ESSL to calculate auxiliary storage, ask yourself which of the following ways you prefer to obtain the information from ESSL:

- \textbf{By leaving error 2015 unrecoverable}, you can obtain the minimum required value of \textit{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.

**Having ESSL Calculate Auxiliary Storage Size with Unrecoverable Error 2015:**

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 55 illustrates what happens when error 2015 is unrecoverable.
Having ESSL Calculate Auxiliary Storage Size with Recoverable Error 2015: In this case, you obtain the minimum required value of \( n_{aux} \) from the error message and from the updated \( n_{aux} \) 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 \( n_{aux} \) arguments. Run your program with the \( n_{aux} \) 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 \( n_{aux1} \) 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 \( n_{aux} \) arguments. You can then react to these updated values during run time in your program.

* This check applies only to several \( n_{aux1} \) arguments in the Fourier transform, convolution, and correlation subroutines.

Figure 1. How to Obtain an \( n_{aux} \) Value from an Error Message, but Terminate

Chapter 2. Planning Your Program
program. ESSL does not perform any computation when this error occurs. For details on how to do this, see Chapter 4, “Coding Your Program,” on page 133. (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**

**Example of Input-Argument Error Recovery for Auxiliary Storage Sizes:** 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 \textit{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**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(naux &lt; 20)</td>
<td>An input-argument error message is issued. The value in the error message is the lower limit, 20. The application program stops.</td>
<td>An input-argument error message is issued. The value in the error message is the lower limit, 20. The application program stops.</td>
</tr>
<tr>
<td>(20 \leq naux &lt; 300)</td>
<td>An input-argument error message is issued. The value in the error message is the minimum required value, 300. The application program stops.</td>
<td>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.</td>
</tr>
<tr>
<td>(naux \geq 300)</td>
<td>Your application program runs successfully.</td>
<td>Your application program runs successfully.</td>
</tr>
</tbody>
</table>

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

**Coding Your Program to Obtain Auxiliary Storage Sizes:** 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 “Input-Argument Errors in Fortran Example” on page 143
- For C, see “Input-Argument Errors in C Example” on page 161
- For C++, see “Input-Argument Errors in C++ Example” on page 178

You may want to provide a separate subroutine to calculate the auxiliary storage size whenever you need it. Figure 3 on page 58 shows how you might code a separate Fortran subroutine. Before calling SCFT in your program, call this subroutine, SCFTQ, 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.
Providing a Correct Transform Length to ESSL

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

For the ESSL subroutines listed in Table 43, 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>
<thead>
<tr>
<th>Subroutine Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fourier Transforms:</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Subroutine Names</td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>SCFTQ(INIT, X, INC1X, INC2X, Y, INC1Y, INC2Y, * N, M, ISIGN, SCALE, AUX1, NAUX1, AUX2, NAUX2)</td>
</tr>
<tr>
<td>REAL<em>4 X(0:</em>),Y(0:*),SCALE</td>
</tr>
<tr>
<td>REAL<em>8 AUX1(7),AUX2(0:</em>),N,M,ISIGN,NAUX1,NAUX2</td>
</tr>
<tr>
<td>EXTERNAL ENOTRM</td>
</tr>
<tr>
<td>CALL EINFO(0)</td>
</tr>
<tr>
<td>CALL ERRSAV(2015,SCFTQ)</td>
</tr>
<tr>
<td>CALL ERRSET(2015,0,-1,1,ENOTRM,0)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>SUBROUTINE SCFTQ(INIT, X, INC1X, INC2X, Y, INC1Y, INC2Y, * N, M, ISIGN, SCALE, AUX1, NAUX1, AUX2, NAUX2, *10)</td>
</tr>
<tr>
<td>CALL SCFT(INIT,X,INC1X,INC2X,Y,INC1Y,INC2Y,</td>
</tr>
<tr>
<td>N,M,ISIGN,SCALE,AUX1,NAUX1,AUX2,NAUX2,*10)</td>
</tr>
<tr>
<td>CALL ERRSTR(2015,SCFTQ)</td>
</tr>
<tr>
<td>RETURN</td>
</tr>
<tr>
<td>10 CONTINUE</td>
</tr>
<tr>
<td>CALL ERRSTR(2015,SCFTQ)</td>
</tr>
<tr>
<td>RETURN 1</td>
</tr>
<tr>
<td>END</td>
</tr>
</tbody>
</table>

Figure 3. Sample Fortran Subroutine to Calculate Auxiliary Storage Sizes in a 32-bit Integer, 64-bit Pointer Environment

Who Do You Want to Calculate the Transform 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 1028 to determine an acceptable length. For details, see “How Do You Calculate the Transform Length Using the Table or Formula?” on page 59.
- Use the ESSL error-handling facilities to return to you an acceptable length. For details, see “How Do You Get ESSL to Calculate the Transform Length Using ESSL Error Handling?” on page 59.
How Do You Calculate the Transform 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 1028. You should use the information in that table 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 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 Transform Length Using ESSL Error Handling?

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

Having ESSL Calculate the Transform Length with Unrecoverable Error 2030

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 on page 60 illustrates what happens when error 2030 is unrecoverable.
Having ESSL Calculate the Transform Length with Recoverable Error 2030

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, “Coding Your Program,” on page 133. (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 61 illustrates what happens when error 2030 is recoverable.
Example of Input-Argument Error Recovery for Transform Lengths

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:

<table>
<thead>
<tr>
<th>( N )</th>
<th>Meaning of the ( N ) Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>7208960</td>
<td>An acceptable transform length, required for successful computing of a Fourier transform</td>
</tr>
<tr>
<td>7340032</td>
<td>The next larger acceptable transform length, required for successful computing of a Fourier transform</td>
</tr>
</tbody>
</table>

Table 44 on page 62 describes the actions taken by ESSL in every possible situation for the values given in this example.

Figure 5. How to Obtain an \( N \) Value from an Error Message and in Your Program
Table 44. Example of Input-Argument Error Recovery for Transform Lengths

<table>
<thead>
<tr>
<th>N Value</th>
<th>Action When 2030 Is an Unrecoverable Input-Argument Error</th>
<th>Action When 2030 Is a Recoverable Input-Argument Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 7208960$</td>
<td>Your application program runs successfully.</td>
<td>Your application program runs successfully.</td>
</tr>
<tr>
<td>or $n = 7340032$</td>
<td>An input-argument error message is issued. The value in the error message is 7340032. The application program stops.</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Coding Your Program to Obtain Transform Lengths**

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 “Input-Argument Errors in Fortran Example” on page 143.
- For C, see “Input-Argument Errors in C Example” on page 161.
- For C++, see “Input-Argument Errors in C++ Example” on page 178.

You may want to provide a separate subroutine to calculate the transform length whenever you need it. Figure 6 shows how you might code a separate Fortran subroutine. Before calling SCFT in your program, you call this subroutine, SCFTQ, 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 SCFTQ(INIT,X,INC1X,INC2X,Y,INC1Y,INC2Y, *
                 * 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. Sample Fortran Subroutine to Calculate Transform Length in a 32-bit Integer, 64-bit Pointer Environment

You might want to combine the request for auxiliary storage sizes along with your request for transform lengths. Figure 7 on page 63 shows how you might code a
separate Fortran subroutine combining both requests. It combines the functions performed by the subroutine shown above and that shown in “Coding Your Program to Obtain Auxiliary Storage Sizes” on page 57.

| SUBROUTINE SCFTQ (INIT, X, INC1X, INC2X, Y, INC1Y, INC2Y,  
| *  
| REAL*4 X(0:*),Y(0:*),SCALE  
| REAL*8 AUX1(7),AUX2(0:*), SCALE  
| INTEGER*4 INIT,INC1X,INC2X,INC1Y,INC2Y,N,M,ISIGN,NAUX1,NAUX2  
| EXTERNAL ENTRM  
| 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. Sample Fortran Subroutine to Calculate Auxiliary Storage Sizes and Transform Length in a 32-bit Integer, 64-bit Pointer Environment

Getting the Best Accuracy

This 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, a 32-bit or 64-bit 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 48. The scalar data types and how you should code them for each programming language are listed under “Coding Your Scalar Data” specific to each programming language in Chapter 4, “Coding Your Program,” on page 133.
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, or whether CPUs, GPUs, or both are 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.

Many of the short-precision subprograms provide increased accuracy by accumulating results in long precision. However, when short-precision subroutines use the AltiVec or VSX unit to improve performance, they do not accumulate intermediate results in long precision. This is noted in the functional description of each subprogram.

Where applicable, the ESSL subroutines use the Multiply-Add instructions, which combine a Multiply and Add operation without an intermediate rounding operation.

The ESSL Serial Libraries and ESSL SMP Libraries allow you to run applications in any of the following environments, and results obtained in any of these environments using the same ESSL library are mathematically equivalent but may not be bitwise-identical:

- 32-bit integer, 32-bit pointer environment (AIX only)
- 32-bit integer, 64-bit pointer environment
- 64-bit integer, 64-bit pointer environment

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


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 \( \alpha A \), where \( \alpha \) is a scalar and \( A \) is a matrix, if \( \alpha \) is zero and one (or more) of the elements of \( A \) 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.
How is Underflow Handled?
ESSL does not mask underflow. If your program incurs a number of unmasked underflows, its overall performance decreases. 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, “Migrating Your Programs,” on page 201.
- Specific information on accuracy for each area of ESSL is given in “Performance and Accuracy Considerations” associated with the subroutine descriptions for that area.
- 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.
- For details on accuracy considerations when using GPUs see: https://developer.nvidia.com/content/precision-performance-floating-point-and-ieee-754-compliance-nvidia-gpus

What about Bitwise-Identical Results?
There are several circumstances where you may not get bitwise-identical results, although the results are mathematically equivalent:

- Results obtained on different hardware platforms
- Results obtained using different ESSL releases
- Results obtained using different ESSL Libraries
- Results obtained using a different number of threads
- Results obtained using arrays that are aligned differently. For example, the Power VSX/VMX unit require specific data alignments. If a subroutine uses one of these units and the input and/or output arrays are not aligned as required, some data may be processed using the floating point unit before or after the main SIMD loop.
- Results obtained using the ESSL SMP CUDA Library with environment variables ESSL_CUDA_HYBRID=yes and ESSL_CUDA_HYBRID=no. See “Using the ESSL SMP CUDA Library” on page 43.

Getting the Best Performance
This 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 78.
- 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
- On VSX enabled processors, specify the size of the leading dimension of a long or short-precision array as follows:
  - Long-precision real arrays - multiple of 2
  - Short-precision real arrays - multiple of 4
  - Short-precision complex arrays - multiple of 2

Vectors and matrices are quadword aligned.

- On Altivec-Enabled Processors, specify the size of the leading dimension of a short-precision array as follows:
  - Short-precision real array - multiple of 4
  - Short-precision complex array - multiple of 2
- 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. For short-precision real and short-precision complex subroutines running on Altivec-enabled processors, see “SIMD Algorithms on POWER Altivec-Enabled Processors” on page 33. For VSX enabled processors, see “SIMD Algorithms on VSX-Enabled Processors” on page 32. 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 1307. For additional information, see “Setting Up Your Data” on page 1031.
If you are using the ESSL SMP CUDA library, performance might improve if you pin your host memory buffers. See “Using the ESSL SMP CUDA Library” on page 43.

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 Chapter 6, “Migrating Your Programs,” on page 201.
- Specific information on performance for each area of ESSL is given in “Performance and Accuracy Considerations” for each grouping of subroutine descriptions.
- Detailed performance information for selected subroutines can be found in reference [38 on page 1365], [49 on page 1366], [50 on page 1366].

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

There are specific ways to handle all these situations.

What Can You Do about Program Exceptions?


What Can You Do about ESSL Input-Argument Errors?

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

All Input-Argument Errors

ESL checks the validity of most input arguments. If it finds that any are invalid, it issues the appropriate error messages. Also, except for the three 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, “Handling Problems,” on page 207.

You can determine the input-argument errors that can occur in a subroutine by looking under “Error Conditions” in each subroutine description. Error messages for all input-argument errors are listed in “Input-Argument Error Messages(2001-2099)” on page 212.

Recoverable Errors 2015, 2030 and 2200 Can Return Updated Values in the NAUX, N and NSINFO Arguments

For three input-argument errors, 2015, 2030, and 2200 in Fortran, C, and C++ 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 51.

• 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 58.

• Determine the minimal size of the array AP for DBSTRF and DBSSV by using error 2200. For help deciding whether you want to use this capability, see "DBSTRF (Symmetric Indefinite Matrix Factorization)" on page 673 and "DBSSV (Symmetric Indefinite Matrix Factorization and Multiple Right-Hand Side Solve)" on page 667.

If you chose to leave errors 2015, 2030 and 2200 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, 2030 and 2200 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 occur, 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, 2030 and 2200 recoverable, see the following:

• "Input-Argument Errors in Fortran" on page 141
• "Input-Argument Errors in C" on page 158
• "Input-Argument Errors in C++" on page 175

What Can You Do about ESSL Computational Errors?
This 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 each subroutine description. 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(2100-2199)" on page 217.

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 69 shows what happens when a computational error occurs.
Recoverable Computational Errors Can Return Values Through EINFO

In Fortran, C, and C++ 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 “EINFO (ESSL Error Information-Handler Subroutine)” on page 1296. 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 1296. 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:

- “Computational Errors in Fortran” on page 144
- “Computational Errors in C” on page 163
- “Computational Errors in C++” on page 180
Figure 9 shows what happens if you make a computational error recoverable.

**What Can You Do about ESSL Resource Errors?**

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

**All Resource Errors**

ESL 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 “ESL Resource Error Messages” on page 210.

You can determine the resource errors that can occur in a subroutine by looking under “Error Conditions” in each subroutine description. Error messages for all resource errors are listed in “Resource Error Messages(2400-2499)” on page 222.

**What Can You Do about ESSL Attention Messages?**

This 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 210.

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 attention messages, see “Informational and Attention Error Messages(2600-2699)” on page 222.

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

This 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 45 shows the default values established in the table when ESSL is installed.

Table 45. ESSL Error Option Table Default Values

<table>
<thead>
<tr>
<th>Range of Error Messages (From–To)</th>
<th>Number of Allowable Errors (ALLOW)</th>
<th>Number of Messages Printed (PRINT)</th>
<th>Modifiable Table Entry (MODENT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2538–2000</td>
<td>Unlimited</td>
<td>255</td>
<td>NO</td>
</tr>
<tr>
<td>2538–2001 through 2538–2073</td>
<td>Unlimited</td>
<td>255</td>
<td>YES</td>
</tr>
<tr>
<td>2538–2074</td>
<td>Unlimited</td>
<td>5</td>
<td>YES</td>
</tr>
<tr>
<td>2538–2075 through 2538–2098</td>
<td>Unlimited</td>
<td>255</td>
<td>YES</td>
</tr>
<tr>
<td>2538–2099</td>
<td>1</td>
<td>255</td>
<td>YES</td>
</tr>
<tr>
<td>2538–2100 through 2538–2101</td>
<td>1</td>
<td>255</td>
<td>YES</td>
</tr>
<tr>
<td>2538–2102</td>
<td>Unlimited</td>
<td>255</td>
<td>YES</td>
</tr>
<tr>
<td>2538–2103 through 2538–2113</td>
<td>1</td>
<td>255</td>
<td>YES</td>
</tr>
<tr>
<td>2538–2114</td>
<td>Unlimited</td>
<td>255</td>
<td>YES</td>
</tr>
<tr>
<td>2538–2115 through 2538–2122</td>
<td>1</td>
<td>255</td>
<td>YES</td>
</tr>
<tr>
<td>2538–2123 through 2538–2124</td>
<td>Unlimited</td>
<td>255</td>
<td>YES</td>
</tr>
<tr>
<td>2538–2125 through 2538–2126</td>
<td>1</td>
<td>255</td>
<td>YES</td>
</tr>
<tr>
<td>2538–2127</td>
<td>Unlimited</td>
<td>255</td>
<td>YES</td>
</tr>
<tr>
<td>2538–2128 through 2538–2137</td>
<td>1</td>
<td>255</td>
<td>YES</td>
</tr>
<tr>
<td>2538–2138 through 2538–2143</td>
<td>Unlimited</td>
<td>255</td>
<td>YES</td>
</tr>
<tr>
<td>2538–2144 through 2538–2145</td>
<td>1</td>
<td>255</td>
<td>YES</td>
</tr>
<tr>
<td>2538–2146 through 2538–2149</td>
<td>Unlimited</td>
<td>255</td>
<td>YES</td>
</tr>
<tr>
<td>2538–2150</td>
<td>1</td>
<td>255</td>
<td>YES</td>
</tr>
<tr>
<td>2538–2151 through 2538–2166</td>
<td>Unlimited</td>
<td>255</td>
<td>YES</td>
</tr>
<tr>
<td>2538–2167 through 2538–2198</td>
<td>1</td>
<td>255</td>
<td>YES</td>
</tr>
<tr>
<td>2538–2199</td>
<td>1</td>
<td>255</td>
<td>YES</td>
</tr>
<tr>
<td>2538–2200 through 2538–2299</td>
<td>Unlimited</td>
<td>255</td>
<td>YES</td>
</tr>
</tbody>
</table>
Table 45. ESSL Error Option Table Default Values (continued)

<table>
<thead>
<tr>
<th>Range of Error Messages (From–To)</th>
<th>Number of Allowable Errors (ALLOW)</th>
<th>Number of Messages Printed (PRINT)</th>
<th>Modifiable Table Entry (MODENT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2538–2400 through 2538–2499</td>
<td>1</td>
<td>255</td>
<td>NO</td>
</tr>
<tr>
<td>2538–2600 through 2538–2609</td>
<td>Unlimited</td>
<td>255</td>
<td>NO</td>
</tr>
<tr>
<td>2538–2610 through 2538–2612</td>
<td>Unlimited</td>
<td>-1</td>
<td>YES</td>
</tr>
<tr>
<td>2538–2613 through 2538–2613</td>
<td>Unlimited</td>
<td>255</td>
<td>NO</td>
</tr>
<tr>
<td>2538–2614 through 2538–2615</td>
<td>Unlimited</td>
<td>1</td>
<td>NO</td>
</tr>
<tr>
<td>2538–2616 through 2538–2699</td>
<td>Unlimited</td>
<td>255</td>
<td>NO</td>
</tr>
<tr>
<td>2538–2700 through 2538–2799</td>
<td>1</td>
<td>255</td>
<td>NO</td>
</tr>
</tbody>
</table>

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, “Utilities,” on page 1293.

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 67. For how to use ERRSET to do this, see the information specific to your programming language in Chapter 4, “Coding Your Program,” on page 133.

To Obtain the Minimal Size of the Array AP for DBSTRF and DBSSV: You may want to make ESSL input-argument error 2200 recoverable, so ESSL returns an updated size to your program. For a more detailed discussion, see “What Can You Do about ESSL Input-Argument Errors?” on page 67. For how to use ERRSET to do this, see the information specific to your programming language in Chapter 4, “Coding Your Program,” on page 133.

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 68. For how to do use ERRSET to do this, see the information specific to your programming language in Chapter 4, “Coding Your Program,” on page 133.

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?” For how to use ERRSAV and ERRSTR, see the information specific to your programming language in Chapter 4, “Coding Your Program,” on page 133.

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, “Coding Your Program,” on page 133.

How does Error Handling Work in a Threaded Environment?

When your application program or the open MP library first creates a thread, ESSL initializes the error option table information to the default settings shown in “What Values Are Set in the ESSL Error Option Table?” on page 73. 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 149.

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 existing threads (for example, the number of allowable errors was exceeded), ESSL terminates your application program. One set of summary information corresponding to the terminating thread is always printed. Summary information corresponding to other threads may also be printed.
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, “Coding Your Program,” on page 133.
- All ESSL error messages are listed under “Messages” on page 211.
- The errors and return codes associated with each ESSL subroutine are listed under “Error Conditions” in each subroutine description.
- 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, “Handling Problems,” on page 207.
Chapter 3. Setting Up Your Data Structures

This 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 here is consistent with standard mathematical definitions, and their representations are consistent with conventions used in mathematical texts.

Overlapping Data Structures: Most of the subroutines do not allow vectors, matrices, or sequences to overlap. If this occurs, results are unpredictable. 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 x₁, y₁, x₂, y₂, x₃, y₃, ... 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.

The column vector appears symbolically as follows:
A row vector appears symbolically as follows:

\[ \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \cdot \\ \cdot \\ x_n \end{bmatrix} \]

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:

\[ \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \cdot \\ \cdot \\ x_n \end{bmatrix} \quad \mathbf{x}^T = \begin{bmatrix} x_1 & x_2 & x_3 \ldots & x_n \end{bmatrix} \quad (\mathbf{x}^T)^T = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \cdot \\ \cdot \\ x_n \end{bmatrix} \]

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, “Reference Information,” on page 223, both types of vectors are represented in the same way, showing the elements of the array that make up the vector \( \mathbf{x} \), 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 \( \mathbf{x} \), containing complex numbers, is denoted by \( \mathbf{x}^H \) and is expressed as follows:

\[ \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \cdot \\ \cdot \\ x_n \end{bmatrix} \quad \mathbf{x}^H = \begin{bmatrix} \overline{x}_1 & \overline{x}_2 & \overline{x}_3 \ldots & \overline{x}_n \end{bmatrix} \]
Just as for the transpose of a vector, no movement of data is necessary for the conjugate transpose of a vector.

**Vector Storage Representation**

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 \( x \) as starting at the first element of array \( A \), specify \( A \) as the argument, such as in:

\[
X = \text{SASUM} \left(4, A, 2\right)
\]

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 \( x \) as starting at element 3 in array \( A \), which is declared as \( A(1:12) \), specify:

\[
X = \text{SASUM} \left(4, A(3), 2\right)
\]

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

\[
X = \text{SASUM} \left(4, A(1), 2\right)
\]

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

\[
X = \text{SASUM} \left(4, A(2,3), 2\right)
\]

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:

\[
1+(n-1)\mid inc \mid
\]

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” on page 78.

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 48 for a description of real and complex numbers, and “How Do You Set Up Your Arrays?” on page 48 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 unpredictable, and you may get program errors.

This 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 \( \text{BEGIN} \) as the starting point in an array \( X \) and \( \text{inc} \) as the stride, this results in the following elements being selected from the array:

\[
\begin{align*}
X(\text{BEGIN}) \\
X(\text{BEGIN}+\text{inc}) \\
X(\text{BEGIN}+(2)\text{inc}) \\
X(\text{BEGIN}+(3)\text{inc}) \\
&\quad \cdot \\
&\quad \cdot \\
&\quad \cdot \\
X(\text{BEGIN}+(n-1)\text{inc})
\end{align*}
\]

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

\[
x_i = X(\text{BEGIN} + (i-1)(\text{inc})) \text{ for } i = 1, n
\]
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 134. 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:

\[
\begin{bmatrix}
1.0 & 8.0 & 15.0 & 22.0 \\
2.0 & 9.0 & 16.0 & 23.0 \\
3.0 & 10.0 & 17.0 & 24.0 \\
4.0 & 11.0 & 18.0 & 25.0 \\
5.0 & 12.0 & 19.0 & 26.0 \\
6.0 & 13.0 & 20.0 & 27.0 \\
7.0 & 14.0 & 21.0 & 28.0 \\
\end{bmatrix}
\]

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:

\[
x_i = X(BEGIN) \text{ for } i = 1, n
\]

**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 \( (x_i) \) 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:

\[
X(SP) \\
X(SP+inc) \\
X(SP+(2)inc) \\
X(SP+(3)inc) \\
\vdots \\
X(SP+(n-1)inc)
\]

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:

\[
x_i = X(BEGIN + (-n+i)(inc)) \text{ for } i = 1, n
\]

### 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, \( \text{INDX} \), also of length 5, contains:

\[
\text{INDX} = (3, 5, 6, 8, 10)
\]

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 \( \text{INDX} \) contains:

\[
\text{INDX} = (10, 6, 8, 3, 5)
\]

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 \( X(i) = x_j \) and \( \text{INDX}(i) = j \).

where:

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

To avoid an error when using the \( \text{INDX} \) array to access the elements in any other target vector, the length of the target vector must be greater than or equal to \( \max(\text{INDX}(i)) \) 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:
where the matrix is named $A$ 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 $A$ is a matrix formed from $A$ by interchanging the rows and columns such that row $i$ of matrix $A$ becomes column $i$ of the transposed matrix. The transpose of $A$ is denoted by $A^T$. Each element $a_{ij}$ in $A$ becomes element $a_{ji}$ in $A^T$. If $A$ is an $m$ by $n$ matrix, then $A^T$ is an $n$ by $m$ matrix. The following represents a matrix and its transpose:

$$
A = \begin{bmatrix}
  a_{11} & \ldots & a_{1n} \\
  \vdots & \ddots & \vdots \\
  a_{m1} & \ldots & a_{mn}
\end{bmatrix}
$$

$$
A^T = \begin{bmatrix}
  a_{11} & \ldots & a_{m1} \\
  \vdots & \ddots & \vdots \\
  a_{1n} & \ldots & a_{mn}
\end{bmatrix}
$$

ESL assumes that all matrices are stored in untransformed format, such as matrix $A$ 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 $A^H$ and is expressed as follows:

$$
A = \begin{bmatrix}
  a_{11} & \ldots & a_{1n} \\
  \vdots & \ddots & \vdots \\
  a_{m1} & \ldots & a_{mn}
\end{bmatrix}
$$

$$
A^H = \begin{bmatrix}
  \bar{a}_{11} & \ldots & \bar{a}_{m1} \\
  \vdots & \ddots & \vdots \\
  \bar{a}_{1n} & \ldots & \bar{a}_{mn}
\end{bmatrix}
$$

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.

**Matrix Storage Representation**

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" 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 48 for a description of real and complex numbers, and "How Do You Set Up Your Arrays?" on page 48 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
```
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 48, 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:

\[
A(BEGINI, BEGINJ) \\
A(BEGINI, BEGINJ+1) \\
A(BEGINI, BEGINJ+2) \\
\ldots \\
A(BEGINI, BEGINJ+n-1)
\]

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:

\[
A(BEGINI, BEGINJ) \\
A(BEGINI+1, BEGINJ) \\
A(BEGINI+2, BEGINJ) \\
\ldots \\
A(BEGINI+m-1, BEGINJ)
\]
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:

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

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 \(A = A^T\), 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 \(1 \leq i \leq n\) and \(1 \leq j \leq n\).

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:

\[
A = \begin{bmatrix}
  a_{11} & a_{21} & a_{31} & \cdots & a_{n1} \\
  a_{21} & a_{22} & a_{32} & \cdot & \cdot \\
  a_{31} & a_{32} & a_{33} & \cdot & \cdot \\
  \cdot & \cdot & \cdot & \cdot & \cdot \\
  a_{n1} & \cdots & \cdots & \cdots & a_{nn}
\end{bmatrix}
\]

**Symmetric Matrix Storage Representation**

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

- “Lower-Packed Storage Mode”
- “Upper-Packed Storage Mode” on page 87
- “Lower Storage Mode” on page 88
- “Upper Storage Mode” on page 89

The storage technique you should use depends on the ESSL subroutine you are using.

**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 \( A \) in an array, \( AP \), using the lower triangular packed technique, use the following formula:

\[
AP(i + ((2n-j)(j-1)/2)) = a_{ij} \quad \text{where} \quad i \geq j
\]

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

\[
\begin{align*}
AP(1) &= a_{11} \text{ (start the first column)} \\
AP(2) &= a_{21} \\
AP(3) &= a_{31} \\
& \quad \vdots \\
AP(n) &= a_{n1} \\
AP(n+1) &= a_{22} \text{ (start the second column)} \\
& \quad \vdots \\
AP(n+2) &= a_{32} \\
& \quad \vdots \\
AP(2n-1) &= a_{n2} \\
AP(2n) &= a_{33} \text{ (start the third column and so forth)} \\
& \quad \vdots \\
AP(2n+1) &= a_{43} \\
& \quad \vdots \\
AP(n(n+1)/2) &= a_{nn}
\end{align*}
\]

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 SPPE, DPPF, SPPS, and DPPS. See the description of those subroutines in Part 2, ‘Reference Information,’ on page 223 for details.

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

```fortran
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:

\[
AP(i+(j(j-1)/2)) = a_{ij} \quad \text{where} \quad j \geq i
\]

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)

\[
\ldots
\]

\[
AP(j(j-1)/2+1) = a_{ij} \quad \text{(start the} \ j-th \ \text{column)}
\]

- \( AP(j(j-1)/2+2) = a_{2j} \)
- \( AP(j(j-1)/2+3) = a_{3j} \)

\[
\ldots
\]
$AP(j(j-1)/2+j) = a_{jj}$ (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:

\[\begin{align*}
&K = 0 \\
&DO 1 J=1,N \\
& \quad DO 2 I=1,J \\
& \quad \quad K = K+1 \\
& \quad \quad AP(K) = A(I,J) \\
& 2 \quad \text{CONTINUE} \\
& 1 \quad \text{CONTINUE}
\end{align*}\]

**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:

\[\begin{bmatrix}
1 & * & * & * & * \\
2 & 6 & * & * & * \\
3 & 7 & 10 & * & *
\end{bmatrix}\]
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 $A_U$.

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:

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

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 $x^T Ax$ is positive for all nonzero vectors $x$.

A real symmetric matrix $A$ is negative definite if and only if $x^T Ax$ is negative for all nonzero vectors $x$.

**Positive Definite or Negative Definite Symmetric Matrix Storage Representation**

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 85.

**Indefinite Symmetric Matrix**

A symmetric matrix $A$ is indefinite if and only if $(x^T Ax) (y^T Ay) < 0$ for some non-zero vectors $x$ and $y$. 
Indefinite Symmetric Matrix Storage Representation
The indefinite 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 85.

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

\[ H = H^* \]

Complex Hermitian Matrix Storage Representation
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 85. (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 87. (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 88.
- **Upper storage mode**, as described in “Upper Storage Mode” on page 89.

Following is an example of a complex Hermitian matrix \( H \) of order 5.

Given the following matrix \( H \):

\[
\begin{bmatrix}
(11, 0) & (21, -1) & (31, 1) & (41, -1) & (51, -1) \\
(21, 1) & (22, 0) & (32, -1) & (42, -1) & (52, 1) \\
(31, -1) & (32, 1) & (33, 0) & (43, -1) & (53, -1) \\
(41, 1) & (42, 1) & (43, 1) & (44, 0) & (54, -1) \\
(51, 1) & (52, -1) & (53, 1) & (54, 1) & (55, 0)
\end{bmatrix}
\]

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**:
  \[
  HP = \begin{bmatrix}
  (11, *) & * & * & * & * \\
  (21, 1) & (22, *) & * & * & * \\
  (31, -1) & (32, 1) & (33, *) & * & * \\
  (41, 1) & (42, 1) & (43, 1) & (44, *) & * \\
  (51, 1) & (52, -1) & (53, 1) & (54, 1) & (55, *)
  \end{bmatrix}
  \]

- **In upper storage mode**

90  ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
\[
\begin{pmatrix}
(11, *) & (21, -1) & (31, 1) & (41, -1) & (51, -1) \\
* & (22, *) & (32, -1) & (42, -1) & (52, 1) \\
* & * & (33, *) & (43, -1) & (53, -1) \\
* & * & * & (44, *) & (54, -1) \\
* & * & * & * & (55, *)
\end{pmatrix}
\]

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 \( A \) is positive definite if and only if \( x^H A x \) is positive for all nonzero vectors \( x \).

A complex Hermitian matrix \( A \) is negative definite if and only if \( x^H A x \) is negative for all nonzero vectors \( x \).

**Positive Definite or Negative Definite Complex Hermitian Matrix Storage Representation**

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 90.

**Indefinite Complex Hermitian Matrix**

A complex Hermitian matrix \( A \) is indefinite if and only if \( (x^H A x) (y^H A y) \) < 0 for some non-zero vectors \( x \) and \( y \).

**Indefinite Complex Hermitian Matrix Storage Representation**

The indefinite 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 90.

**Positive Definite or Negative Definite Symmetric Toeplitz Matrix**

A positive definite or negative definite symmetric matrix \( A \) of order \( n \) is also a Toeplitz matrix if and only if:

\[
a_{ij} = a_{i+1,j-1} \quad \text{for} \quad i = 2, n \quad \text{and} \quad j = 2, n
\]

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 89.

The following matrix illustrates a symmetric Toeplitz matrix of order \( n \); that is, it has \( n \) rows and \( n \) columns:
A symmetric Toeplitz matrix of order \( n \) is represented by a vector \( x \) of length \( n \) containing the elements of the first column of the matrix (or the elements of the first row), such that \( x_i = a_{11} \) for \( i = 1, n \).

The following vector represents the matrix \( A \) shown above:

\[
A = \begin{bmatrix}
a_{11} & a_{21} & a_{31} & \cdots & a_{n1} \\
a_{21} & a_{11} & a_{21} & \cdots & \vdots \\
a_{31} & a_{21} & a_{11} & \cdots & \vdots \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
a_{n1} & \cdots & a_{31} & a_{21} & a_{11}
\end{bmatrix}
\]

A symmetric Toeplitz matrix of order \( n \) is represented by a vector \( x \) of length \( n \) containing the elements of the first column of the matrix (or the elements of the first row), such that \( x_i = a_{11} \) for \( i = 1, n \).

The following vector represents the matrix \( A \) shown above:

\[
x = \begin{bmatrix}
a_{11} \\
a_{21} \\
a_{31} \\
\vdots \\
a_{n1}
\end{bmatrix}
\]

**Positive Definite or Negative Definite Symmetric Toeplitz Matrix Storage Representation**

The elements of the vector \( x \), 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 \( A \) and how it is stored in an array \( x \).

Given the following matrix \( A \):

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

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 \( A \) of order \( n \) is also a Toeplitz matrix if and only if:

\[
a_{ij} = a_{n-i+1, n-j+1}
\]

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 91.

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

$$A = \begin{bmatrix}
a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\
a_{12} & a_{11} & a_{12} & \cdots & \vdots \\
a_{13} & a_{12} & a_{11} & \cdots & \vdots \\
\vdots & \vdots & \vdots & \ddots & a_{13} \\
a_{1n} & \cdots & a_{13} & \cdots & a_{11}
\end{bmatrix}$$

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

The following vector represents the matrix $A$ shown above.

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

**Positive Definite or Negative Definite Complex Hermitian Toeplitz Matrix Storage Representation**

The elements of the vector $x$, 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 $A$ and how it is stored in an array $X$.

Given the following matrix $A$:

$$A = \begin{bmatrix}
(10.0, 0.0) & (2.0, -3.0) & (-3.0, 1.0) & (1.0, 1.0) \\
(2.0, 3.0) & (10.0, 0.0) & (2.0, -3.0) & (-3.0, 1.0) \\
(-3.0, -1.0) & (2.0, 3.0) & (10.0, 0.0) & (2.0, -3.0) \\
(1.0, -1.0) & (-3.0, -1.0) & (2.0, 3.0) & (10.0, 0.0)
\end{bmatrix}$$

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:
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:

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

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

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

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:

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

**Triangular Matrix Storage Representation**

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

- “Upper-Triangular-Packed Storage Mode”
- “Lower-Triangular-Packed Storage Mode” on page 95
- “Upper-Triangular Storage Mode” on page 95
- “Lower-Triangular Storage Mode” on page 96

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 87.

Following is an example of an upper triangular matrix \( U \) of order 5 and how it is stored in array \( U_P \). 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:

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

**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 85.

Following is an example of a lower triangular matrix \( L \) of order 5 and how it is stored in array \( L_P \). 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:

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

**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 \( U_T A \).

Given the following matrix \( U \):

\[
\begin{bmatrix}
11 & 12 & 13 & 14 & 15 \\
0 & 22 & 23 & 24 & 25 \\
0 & 0 & 33 & 34 & 35 \\
\end{bmatrix}
\]
the array is:

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

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:

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

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

**Trapezoidal Matrix**

There are two types of trapezoidal matrices: upper trapezoidal matrix and lower trapezoidal matrix. Trapezoidal matrices have \( m \) rows and \( n \) columns.

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

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

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

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

The following matrices, \( U \) and \( L \), illustrate upper and lower trapezoidal matrices with \( m \) rows and \( n \) columns, respectively:
If $m \geq n$:

$$U = \begin{bmatrix}
  u_{11} & u_{12} & u_{13} & \ldots & u_{1n} \\
  0 & u_{22} & u_{23} & \ldots & 0 \\
  0 & 0 & u_{33} & \ldots & 0 \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  0 & \ldots & 0 & u_{nn} & 0 \\
  0 & \ldots & 0 & 0 & 0 \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  0 & \ldots & 0 & 0 & 0 \\
\end{bmatrix}$$

$$L = \begin{bmatrix}
  l_{11} & 0 & 0 & \ldots & 0 \\
  l_{21} & l_{22} & 0 & \ldots & 0 \\
  l_{31} & l_{32} & l_{33} & \ldots & 0 \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  l_{n1} & \ldots & l_{n,n-1} & l_{nn} & 0 \\
  l_{n+1,1} & \ldots & l_{n+1,n} & 0 & \ldots \\
  l_{n+2,1} & \ldots & l_{n+2,n} & 0 & \ldots \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  l_{m1} & \ldots & l_{m,m-1} & l_{mn} & 0 \\
\end{bmatrix}$$

If $m < n$:

$$U = \begin{bmatrix}
  u_{11} & u_{12} & u_{13} & \ldots & u_{1n} & u_{1,n+1} & \ldots & u_{1m} \\
  0 & u_{22} & u_{23} & \ldots & u_{2,n} & u_{2,n+1} & \ldots & u_{2m} \\
  0 & 0 & u_{33} & \ldots & u_{3,n} & u_{3,n+1} & \ldots & u_{3m} \\
  \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
  0 & \ldots & 0 & u_{nn} & u_{n,n+1} & \ldots & u_{nm} \\
  0 & \ldots & 0 & 0 & u_{n,n+1} & \ldots & u_{nm} \\
  \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \ddots & \vdots \\
  0 & \ldots & 0 & 0 & 0 & u_{m,n} & \ldots & u_{mn} \\
\end{bmatrix}$$

$$L = \begin{bmatrix}
  l_{11} & 0 & 0 & \ldots & 0 & 0 & \ldots & 0 \\
  l_{21} & l_{22} & 0 & \ldots & 0 & 0 & \ldots & 0 \\
  l_{31} & l_{32} & l_{33} & \ldots & 0 & 0 & \ldots & 0 \\
  \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
  l_{n1} & \ldots & l_{n,n-1} & l_{nn} & 0 & 0 & \ldots & 0 \\
  l_{n+1,1} & \ldots & l_{n+1,n} & 0 & \ldots & 0 & \ldots & 0 \\
  l_{n+2,1} & \ldots & l_{n+2,n} & 0 & \ldots & 0 & \ldots & 0 \\
  \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \ddots & \vdots \\
  l_{m1} & \ldots & l_{m,m-1} & l_{mn} & 0 & 0 & \ldots & 0 \\
\end{bmatrix}$$

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

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

The following matrices, $U$ and $L$, illustrate upper and lower unit real trapezoidal matrices with $m$ and $n$ columns, respectively:

If $m \geq n$:

$$U = \begin{bmatrix}
  1 & u_{12} & u_{13} & \ldots & u_{1n} \\
  0 & 1 & u_{23} & \ldots & 0 \\
  0 & 0 & 1 & \ldots & 0 \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  0 & \ldots & 0 & 1 & 0 \\
  0 & \ldots & 0 & 0 & 0 \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  0 & \ldots & 0 & 0 & 0 \\
\end{bmatrix}$$

$$L = \begin{bmatrix}
  1 & 0 & 0 & \ldots & 0 \\
  0 & 1 & 0 & \ldots & 0 \\
  1 & l_{32} & 0 & \ldots & 0 \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  l_{n1} & \ldots & l_{n,n-1} & l_{nn} & 0 \\
  l_{n+1,1} & \ldots & l_{n+1,n} & 0 & \ldots \\
  l_{n+2,1} & \ldots & l_{n+2,n} & 0 & \ldots \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  l_{m1} & \ldots & l_{m,m-1} & l_{mn} & 0 \\
\end{bmatrix}$$

If $m < n$:
Trapezoidal Matrix Storage Representation

The storage modes used for storing trapezoidal matrices are described in the following:

- "Upper-Trapezoidal Storage Mode"
- "Lower-Trapezoidal Storage Mode" on page 99

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

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

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

Given the following matrix $U$:

$$
U = \begin{bmatrix}
1 & u_{12} & u_{13} & \ldots & u_{1n} & u_{1,n+1} & \ldots & u_{1m} \\
0 & 1 & u_{23} & \ldots & u_{2,n-1} & u_{2n} & \ldots & u_{2m} \\
0 & 0 & 1 & \ldots & \ldots & \ldots & \ldots & \ldots \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\
0 & \ldots & 0 & 1 & u_{m,n-1} & \ldots & u_{mn}
\end{bmatrix}
$$

the array is:

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

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

Following is an example of an upper trapezoidal matrix $U$ with 5 rows and 7 columns and how it is stored in array $UTA$.

Given the following matrix $U$:

$$L = \begin{bmatrix}
1 & 0 & 0 & \ldots & 0 & 0 & \ldots & 0 \\
l_{11} & 1 & 0 & \ldots & \ldots & \ldots \\
l_{12} & l_{22} & 1 & \ldots & \ldots & \ldots \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
l_{m1} & \ldots & 1 & 0 & \ldots & \ldots \\
l_{m2} & \ldots & \ldots & 1 & 0 & \ldots & \ldots \\
\end{bmatrix}
$$
the array is:

\[
\begin{bmatrix}
11 & 12 & 13 & 14 & 15 & 16 & 17 \\
0 & 22 & 23 & 24 & 25 & 26 & 27 \\
0 & 0 & 33 & 34 & 35 & 36 & 37 \\
0 & 0 & 0 & 44 & 45 & 46 & 47 \\
0 & 0 & 0 & 0 & 55 & 56 & 57 \\
\end{bmatrix}
\]

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

**Lower-Trapezoidal Storage Mode:** A trapezoidal matrix is stored in lower-trapezoidal storage mode in a two-dimensional array. Only the elements in the lower trapezoid of the matrix, including the diagonal, are stored in the lower trapezoid of the array.

Following is an example of a lower trapezoidal matrix \( L \) of order 5 and how it is stored in array \( LTA \).

Given the following matrix \( L \):

\[
\begin{bmatrix}
11 & 0 & 0 & 0 & 0 \\
21 & 22 & 0 & 0 & 0 \\
31 & 32 & 33 & 0 & 0 \\
41 & 42 & 43 & 44 & 0 \\
51 & 52 & 53 & 54 & 55 \\
61 & 62 & 63 & 64 & 65 \\
71 & 72 & 73 & 74 & 75 \\
\end{bmatrix}
\]

the array is:

\[
\begin{bmatrix}
11 & * & * & * & * \\
21 & 22 & * & * & * \\
31 & 32 & 33 & * & * \\
41 & 42 & 43 & 44 & * \\
51 & 52 & 53 & 54 & 55 \\
61 & 62 & 63 & 64 & 65 \\
71 & 72 & 73 & 74 & 75 \\
\end{bmatrix}
\]

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

Following is an example of an lower trapezoidal matrix \( U \) with 5 rows and 7 columns and how it is stored in array \( LTA \).

Given the following matrix \( L \):

\[
\begin{bmatrix}
11 & 0 & 0 & 0 & 0 & 0 & 0 \\
21 & 22 & 0 & 0 & 0 & 0 & 0 \\
31 & 32 & 33 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]
the array is:
\[
\begin{bmatrix}
11 & * & * & * & * & * \\
21 & 22 & * & * & * & * \\
31 & 32 & 33 & * & * & * \\
41 & 42 & 43 & 44 & * & * \\
51 & 52 & 53 & 54 & 55 & * & * \\
\end{bmatrix}
\]

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 \(mu = p-1\):

\[
A = \begin{bmatrix}
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\end{bmatrix}
\]

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 \(m > n\):
Following is a general band matrix with $m < n$:

\[
\begin{pmatrix}
  a_{11} & a_{12} & a_{13} & \ldots & a_{1p} & 0 & \ldots & 0 \\
  a_{21} & a_{22} & a_{23} & \ldots & 0 & \ldots & 0 \\
  a_{31} & a_{32} & a_{33} & \ldots & 0 & \ldots & 0 \\
  \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
  \vdots & \vdots & \vdots & \ddots & a_{n-1,n-1} & \ldots & 0 \\
  0 & 0 & 0 & \ldots & 0 & 1 & 1 \\
  0 & 0 & 0 & \ldots & \vdots & \vdots & \vdots \\
  0 & 0 & 0 & \ldots & a_{n-1,n-1} & a_n & a_{nn} \\
\end{pmatrix}
\]

\[
A = \begin{pmatrix}
  a_{11} & & & & & & \\
  a_{21} & a_{22} & a_{23} & & & & \\
  a_{31} & a_{32} & a_{33} & \ldots & & & \\
  \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \\
  \vdots & \vdots & \vdots & \ddots & a_{n-1,n-1} & a_n & a_{nn} \\
  \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
  0 & 0 & 0 & \ldots & 0 & 1 & 1 \\
\end{pmatrix}
\]

**General Band Matrix Storage Representation**

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

- “General-Band Storage Mode”
- “BLAS-General-Band Storage Mode” on page 103

**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 $A$ 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 $A$ are stored in each column in array $AGB$ as follows, where $a_{11}$ is stored at
AGB\((ml+mu+1, 1)\):

\[
AGB = \begin{bmatrix}
    * & . & . & . & . & . & . & . & . \\
    . & . & . & . & . & . & . & . & . \\
    . & . & . & . & . & . & . & . & . \\
    . & . & . & . & . & . & . & . & . \\
    . & . & . & . & . & . & . & . & . \\
    . & . & . & . & . & . & . & . & . \\
    . & . & . & . & . & . & . & . & . \\
    \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
    . & . & . & . & . & . & . & . & . \\
    . & . & . & . & . & . & . & . & . \\
\end{bmatrix}
\]

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 \(AGB(6,1)\):

\[
\begin{bmatrix}
    * & * & * & * & * & * & * & * & * \\
    * & * & * & * & * & * & * & * & * \\
    * & * & 14 & 25 & 36 & 47 & 58 & 69 \\
    * & * & 13 & 24 & 35 & 46 & 57 & 68 & 79
\end{bmatrix}
\]
Following is an example of how to transform your general band matrix, of order \( n \), to general-band storage mode:

\[
\begin{array}{ccccccccc}
* & 12 & 23 & 34 & 45 & 56 & 67 & 78 & 89 \\
11 & 22 & 33 & 44 & 55 & 66 & 77 & 88 & * \\
21 & 32 & 43 & 54 & 65 & 76 & 87 & 98 & * \\
31 & 42 & 53 & 64 & 75 & 86 & 97 & * & * \\
\end{array}
\]

\[
AGB = \begin{bmatrix}
* & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * \\
\end{bmatrix}
\]

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 \( A \):

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

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

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

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 \( A \):

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

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

where \( k \) is the half band width.

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

\[
A = \begin{bmatrix}
 a_{11} & a_{21} & a_{31} & \ldots & a_{q1} & 0 & \ldots & 0 \\
 a_{21} & a_{22} & a_{32} & \ldots & 0 & \ldots & \ldots & \ldots \\
 a_{31} & a_{32} & a_{33} & \ldots & 0 & \ldots & \ldots & \ldots \\
 \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \ddots & \ddots \\
 a_{q1} & \ldots & \ldots & \ldots & a_{qq} & 0 & \ldots & 0 \\
 0 & \ldots & \ldots & \ldots & 0 & \ldots & \ldots & \ldots \\
 \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \ddots & \ddots \\
 0 & \ldots & 0 & \ldots & 0 & \ldots & \ldots & a_{nn}
\end{bmatrix}
\]

**Symmetric Band Matrix Storage Representation**

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

- “Upper-Band-Packed Storage Mode” on page 106
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 \( A \) 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:

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

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

Following is an example of a symmetric band matrix \( A \) of order 6 and a half band width of 3.

Given the following matrix \( A \):

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

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

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

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:

\[
\begin{bmatrix}
  a_{11} & a_{22} & \cdots & & \cdots & a_{nn} \\
  a_{21} & a_{32} & \cdots & & \cdots & \ast \\
  a_{31} & a_{42} & \cdots & & \cdots & \cdot \\
  \cdot & \cdot & \cdots & & \cdots & \cdot \\
  \cdot & \cdot & \cdots & & \cdots & \cdot \\
  a_{q1} & a_{q+1,2} & \cdots & a_{n,n-k} & \ast & \cdots & \ast
\end{bmatrix}
\]

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:

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

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

\[
\begin{align*}
\text{DO } & 20 \text{ J=1,N} \\
\text{DO } & 10 \text{ I=J,MIN(J+K,N)} \\
\text{ASB}(I-J+1,J) & = A(I,J) \\
\text{10 CONTINUE} \\
\text{20 CONTINUE}
\end{align*}
\]

**Positive Definite Symmetric Band Matrix**

A real symmetric band matrix $A$ is positive definite if and only if $x^TAx$ is positive for all nonzero vectors $x$. 

---

*Chapter 3. Setting Up Your Data Structures 107*
Positive Definite Symmetric Band Matrix Storage Representation
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 105.

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

\[ H = H^\dagger \]

Complex Hermitian Band Matrix Storage Representation
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 107
- Upper-band-packed storage mode, as described in “Upper-Band-Packed Storage Mode” on page 106

Following is an example of a complex Hermitian band matrix \( H \) of order 5, having a half band width of 2.

Given the following matrix \( H \):

\[
\begin{pmatrix}
(11, 0) & (21, -1) & (31, 1) & (0, 0) & (0, 0) \\
(21, 1) & (22, 0) & (32, -1) & (42, -1) & (0, 0) \\
(31, -1) & (32, 1) & (33, 0) & (43, -1) & (53, -1) \\
(0, 0) & (42, 1) & (43, 1) & (44, 0) & (54, -1) \\
(0, 0) & (0, 0) & (53, 1) & (54, 1) & (55, 0)
\end{pmatrix}
\]

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

- In lower-band-packed storage mode:

\[
HP = \begin{bmatrix}
(11, \ast) & (22, \ast) & (33, \ast) & (44, \ast) & (55, \ast) \\
(21, 1) & (32, 1) & (43, 1) & (54, 1) & \ast \\
(31, -1) & (42, 1) & (53, 1) & \ast & \ast
\end{bmatrix}
\]

- In upper-band-packed storage mode:

\[
HP = \begin{bmatrix}
\ast & \ast & (31, 1) & (42, -1) & (53, -1) \\
\ast & (21, -1) & (32, -1) & (43, -1) & (54, -1) \\
(11, \ast) & (22, \ast) & (33, \ast) & (44, \ast) & (55, \ast)
\end{bmatrix}
\]

where “\( \ast \)” 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.

Positive Definite Complex Hermitian Band Matrix
A complex Hermitian band matrix \( A \) is positive definite if and only if \( x^\dagger A x \) is positive for all nonzero vectors \( x \).
Positive Definite Complex Hermitian Band Matrix Storage Representation

The positive definite complex Hermitian band matrix is stored in the same way a complex Hermitian band matrix is stored. For a description of this storage technique, see “Complex Hermitian Band Matrix” on page 108.

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 \( U \) 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:

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

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

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

The following matrix \( U \) illustrates an upper triangular band matrix of order \( n \) with an upper band width \( k = q-1 \):

\[
U = \begin{bmatrix}
\vdots & \vdots & \vdots & \cdots & \vdots \\
\vdots & \vdots & \vdots & \cdots & \vdots \\
\vdots & \vdots & \vdots & \cdots & \vdots \\
0 & \cdots & \vdots & \cdots & \vdots \\
0 & 0 & \cdots & \cdots & \vdots \\
0 & 0 & 0 & \cdots & \vdots \\
0 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & \cdots & 0 \\
\end{bmatrix}
\]

A band matrix \( L \) 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:

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

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

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

The following matrix \( L \) illustrates an upper triangular band matrix of order \( n \) with a lower band width \( k = q-1 \):
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 93.

**Triangular Band Matrix Storage Representation**

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

- “Upper-Triangular-Band-Packed Storage Mode”
- “Lower-Triangular-Band-Packed Storage Mode” on page 111

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:

$$
UTB = \begin{bmatrix}
* & * & \cdots & u_{1,p} & u_{2,p-1} & \cdots & u_{n-k,n} \\
& * & \cdots & u_{13} & u_{24} & \cdots & \\
& & * & u_{12} & u_{23} & \cdots & \\
& & & * & u_{11} & u_{22} & \cdots & u_{nn}
\end{bmatrix}
$$

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:

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

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

\[
DO \ 20 \ J=1,N \\
M=K+1-J \\
DO \ 10 \ I=\text{MAX}(1,J-K),J \\
\quad \text{UTB}(M+I,J)=U(I,J) \\
10 \ \text{CONTINUE} \\
20 \ \text{CONTINUE}
\]

**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 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:

\[
LTB = \begin{bmatrix}
l_{11} & l_{22} & \ldots & l_{nn} \\
l_{21} & l_{32} & \ldots & * \\
l_{31} & l_{42} & \ldots & \cdot \\
& \cdot & \cdot & \cdot \\
& \cdot & \cdot & \cdot \\
l_{q1} & l_{q+1,2} & \ldots & l_{n,n-k} & * & \ldots & * \\
\end{bmatrix}
\]

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:

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

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

\[
\begin{array}{c}
\text{DO 20 } J=1,N \\
\text{M=1-J} \\
\text{DO 10 } I=J,\text{MIN(N,J+K)} \\
\text{ LTB(M+I,J)=L(I,J)} \\
\text{ 10 CONTINUE} \\
\text{20 CONTINUE}
\end{array}
\]

**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:

\[a_{ij} = 0 \quad \text{if} \quad |i-j| > 1\]

The following matrix illustrates a general tridiagonal matrix of order $n$:

\[
A = \begin{bmatrix}
a_{11} & a_{12} & 0 & \ldots & 0 \\
a_{21} & a_{22} & a_{23} & 0 & \ldots \\
0 & a_{32} & a_{33} & a_{34} & 0 \\
& 0 & a_{43} & a_{44} & \ldots \\
& & 0 & \ldots \\
& & & 0 & \ldots \\
0 & \ldots & \ldots & \ldots & a_{nn}
\end{bmatrix}
\]

**General Tridiagonal Matrix Storage Representation**

The storage modes used for storing trapezoidal matrices are described in the following:

- “LAPACK-General Tridiagonal Storage Mode”
- “General Tridiagonal Storage Mode” on page 113

**LAPACK-General Tridiagonal Storage Mode:** This storage mode is for use with LAPACK compatible tridiagonal subroutines.
Only the diagonal, subdiagonal, and superdiagonal elements of the general tridiagonal matrix are stored for LAPACK-general-tridiagonal storage mode. The diagonal elements of a general tridiagonal matrix, \( A \), of order \( n \) are stored in a one-dimensional array \( D \) of length \( n \).

The subdiagonal and superdiagonal elements of a general tridiagonal matrix \( A \) of order \( n \) are stored in one dimensional arrays \( DL \) and \( DU \) of length \( n-1 \), respectively. \( DL, D, \) and \( DU \) are stored as follows:

\[
DL = (a_{21}, a_{32}, a_{43}, \ldots a_{n,n-1})
\]
\[
D = (a_{11}, a_{22}, a_{33}, \ldots a_{n,n})
\]
\[
DU = (a_{12}, a_{23}, a_{34}, \ldots a_{n-1,n})
\]

Following is an example of a general tridiagonal matrix \( A \) 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 LAPACK-general tridiagonal storage mode in arrays \( DL, D, \) and \( DU \), as follows:

\[
DL = (21, 32, 43, 54)
\]
\[
D = (11, 22, 33, 44, 55)
\]
\[
DU = (12, 23, 34, 45)
\]

**General Tridiagonal Storage Mode:** This storage mode is for use with non-LAPACK compatible tridiagonal subroutines.

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, \( A \), 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:

\[
C = (*, a_{21}, a_{32}, a_{43}, \ldots, a_{n,n-1})
\]

and array \( D \) contains the main diagonal elements, stored as follows:

\[
D = (a_{11}, a_{22}, a_{33}, \ldots, a_{nn})
\]

and array \( E \) contains the superdiagonal elements, stored as follows:

\[
E = (a_{12}, a_{23}, a_{34}, \ldots, a_{n-1,n}, *)
\]

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

Following is an example of a general tridiagonal matrix \( A \) of order 5:

\[
\begin{bmatrix}
11 & 12 & 0 & 0 & 0 \\
21 & 22 & 23 & 0 & 0 \\
0 & 32 & 33 & 34 & 0 \\
\end{bmatrix}
\]
which you store in tridiagonal storage mode in arrays C, D, and E, each of length 5, as follows:

\[
\begin{align*}
C &= (*, 21, 32, 43, 54) \\
D &= (11, 22, 33, 44, 55) \\
E &= (12, 23, 34, 45, *)
\end{align*}
\]

**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:

\[
(a_{ij} = 0 \text{ if } |i-j| > 1) \quad \text{and} \quad (a_{ij} = a_{ji} \text{ if } |i-j| = 1)
\]

The following matrix illustrates a symmetric tridiagonal matrix of order \( n \):

\[
A = \begin{bmatrix}
a_{11} & a_{21} & 0 & \cdots & 0 \\
a_{21} & a_{22} & a_{32} & 0 & \cdots \\
0 & a_{32} & a_{33} & a_{43} & 0 \\
\vdots & 0 & a_{43} & a_{44} & \ddots \\
\vdots & \vdots & \ddots & \ddots & \vdots \\
0 & \cdots & \cdots & \cdots & a_{nn}
\end{bmatrix}
\]

### Symmetric Tridiagonal Matrix Storage Representation

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

- “LAPACK-Symmetric-Tridiagonal Storage Mode”
- “Symmetric-Tridiagonal Storage Mode” on page 115

**LAPACK-Symmetric-Tridiagonal Storage Mode:** This storage mode is for use with LAPACK compatible tridiagonal subroutines.

Only the diagonal and subdiagonal elements of the symmetric tridiagonal matrix are stored for LAPACK-symmetric-tridiagonal storage mode. The diagonal elements of a symmetric tridiagonal matrix \( A \) of order \( n \) are stored in a one dimensional array \( D \) length \( n \). The subdiagonal elements of a symmetric matrix \( A \) are stored in a one dimensional array \( E \) of length \( n-1 \). \( D \) and \( E \) are stored as follows:

\[
D = (a_{11}, a_{22}, a_{33}, ..., a_{nn}) \\
E = (a_{21}, a_{32}, a_{43}, ..., a_{n,n-1})
\]

Following is an example of a symmetric tridiagonal matrix \( A \) of order 5:
which you store in LAPACK-symmetric-tridiagonal storage mode in arrays \( D \) and \( E \), each of length 4, as follows:

\[
D = (10, 20, 30, 40, 50) \\
E = (1, 2, 3, 4)
\]

**Symmetric-Tridiagonal Storage Mode:** This storage mode is for use with non-LAPACK compatible tridiagonal subroutines.

Only the diagonal and subdiagonal elements of the symmetric tridiagonal matrix are stored for 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 \( C \) contains the subdiagonal elements, stored as follows:

\[
C = (*, a_{21}, a_{32}, a_{43}, \ldots, a_{n,n+1})
\]

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:

\[
D = (a_{11}, a_{22}, a_{33}, \ldots, a_{nn})
\]

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 \( x^T A x \) is positive for all nonzero vectors \( x \).

#### Positive Definite Symmetric Tridiagonal Matrix Storage Representation

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 114.
Complex Hermitian Tridiagonal Matrix

A complex tridiagonal matrix is Hermitian if it is equal to its conjugate transpose: \( H = H^H \).

Complex Hermitian Tridiagonal Storage Representation

Only the diagonal and subdiagonal elements of the complex Hermitian tridiagonal matrix are stored for LAPACK-complex-Hermitian-tridiagonal storage mode. The diagonal elements of a complex Hermitian tridiagonal matrix \( A \) of order \( n \) are stored in a one dimensional array \( D \) of length \( n \). The subdiagonal elements of a complex Hermitian matrix \( A \) are stored in a one dimensional array \( E \) of length \( n-1 \). \( D \) and \( E \) are stored as follows:

\[
D = (a_{11}, a_{22}, a_{33}, \ldots, a_{nn})
\]

\[
E = (*, a_{21}, a_{32}, a_{43}, \ldots, a_{n,n-1})
\]

Following is an example of a symmetric tridiagonal matrix \( A \) of order 5:

\[
\begin{bmatrix}
10, & 0 & 1, & 1 & 1, & 2 & 1, & 3 & 1, & 4 \\
1, & -1 & 20, & 0 & 2, & 1 & 2, & 2 & 2, & 3 \\
1, & -2 & 30, & 0 & 3, & 1 & 3, & 1 & 3, & 2 \\
1, & -3 & 40, & 0 & 4, & 1 & 4, & 1 & 4, & 1 \\
1, & -4 & 50, & 0 & 5, & 0 & 5, & 0 & 5, & 0 \\
\end{bmatrix}
\]

which you store in LAPACK-complex-Hermitian-tridiagonal storage mode in arrays \( D \) of length 5 and complex array \( E \), each of length 4, as follows:

\[
D = (10, 20, 30, 40, 50)
\]

\[
E = ((1,-1), (2,-1), (3,-1), (4,-1))
\]

Positive Definite Complex Hermitian Tridiagonal Matrix

A complex Hermitian tridiagonal matrix is positive definite if and only if \( x^H A x \) is positive for all nonzero vectors \( x \).

Positive Definite Complex Hermitian Tridiagonal Matrix Storage Representation

The positive definite complex Hermitian tridiagonal matrix is stored in the same way a complex Hermitian tridiagonal matrix is stored. For a description of this storage technique, see "Complex Hermitian Tridiagonal Matrix."

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 \( A \):

\[
\begin{bmatrix}
11 & 0 & 13 & 0 & 0 & 0 \\
21 & 22 & 0 & 24 & 0 & 0 \\
0 & 32 & 33 & 0 & 35 & 0 \\
0 & 0 & 43 & 44 & 0 & 46 \\
51 & 0 & 0 & 54 & 55 & 0 \\
61 & 62 & 0 & 0 & 65 & 66 \\
\end{bmatrix}
\]

Sparse Matrix Storage Representation

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:

- "Compressed-Matrix Storage Mode"
- "Compressed-Diagonal Storage Mode" on page 118
- "Storage-by-Indices" on page 121
- "Storage-by-Columns" on page 121
- "Storage-by-Rows" on page 122
- "Diagonal-Out Skyline Storage Mode" on page 124
- "Profile-In Skyline Storage Mode" on page 126

**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 [87 on page 1368]. 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:

$$
AC = \begin{bmatrix}
11 & 13 & 0 & 0 \\
22 & 21 & 24 & 0 \\
33 & 32 & 35 & 0 \\
44 & 43 & 46 & 0 \\
\end{bmatrix}
$$
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 $KA$,
$AC(i,k) \leq n$

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 $lqa \geq m$.
- Array $KA$ is defined as $KA(lda,nz)$, where $lqa \geq m$.

**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 explains both of these variations; however, the conventions used for numbering the diagonals in the matrix, which apply to the storage descriptions, are explained first.

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 $d_k$, where $k = 1$-$n$, $n$-$1$.

The following matrix shows the starting position of each diagonal, $d_k$:
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 $A$:

$$
A = \begin{bmatrix}
11 & 0 & 13 & 0 & 0 & 0 \\
21 & 22 & 0 & 24 & 0 & 0 \\
0 & 32 & 33 & 0 & 35 & 0 \\
0 & 0 & 43 & 44 & 0 & 46 \\
51 & 0 & 0 & 54 & 55 & 0 \\
61 & 62 & 0 & 0 & 65 & 66
\end{bmatrix}
$$

the arrays are:

$$
AD = \begin{bmatrix}
11 & 13 & 0 & 0 & 0 \\
22 & 24 & 21 & 0 & 0 \\
33 & 35 & 32 & 0 & 0 \\
44 & 46 & 43 & 0 & 0 \\
55 & 0 & 54 & 51 & 0 \\
66 & 0 & 65 & 62 & 61
\end{bmatrix}
$$
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 \( d_k \neq (0, \ldots, 0) \), for \( k = 1-n, n-1 \)

for general square sparse matrices, or

for each unique \( d_k \neq (0, \ldots, 0) \), 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 \( dp_k \).

where:

- Array \( AD \) is defined as \( AD(lda, nd) \), where \( lda \equiv n \), and where \( nd \) is the number of nonzero diagonals, \( d_k \) that are stored in array \( AD \).
- Array \( LA \) has \( nd \) elements.
- \( k \) is the diagonal number of each diagonal, \( d_{ij} \), where \( k = i-j \).
- \( dp_k \) are the diagonals, \( d_{ij} \) with padding, which are constructed from the sparse matrix \( A \) elements, \( a_{ij} \), for \( i, j = 1, n \) as follows:

For superdiagonals (\( k > 0 \)), \( dp_k \) has \( k \) trailing zeros: \( dp_k = (a_{1,k+1}, a_{2,k+2}, \ldots, a_{n-k,n}, 0, \ldots, 0) \)
For the main diagonal \((k = 0)\), \(dp_0\) has no padding: 
\[dp_0 = (a_{11}, a_{22}, \ldots, a_{nn})\]
For subdiagonals \((k < 0)\), \(dp_k\) has \(|k|\) leading zeros: 
\[dp_k = (0, \ldots, 0_{|k|}, a_{|k|+1,1}, a_{|k|+2,2}, \ldots, a_{nn})\]

**Storage-by-Indices:** For a sparse matrix \(A\), 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 \(A\) 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 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 \(A\).
- \(JA\), 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 sparse matrix \(A\) and how it can be stored in arrays \(AR\), \(IA\), and \(JA\):

Given the following matrix \(A\):

\[
\begin{bmatrix}
11 & 0 & 13 & 0 & 0 & 0 \\
21 & 22 & 0 & 24 & 0 & 0 \\
0 & 32 & 33 & 0 & 35 & 0 \\
0 & 0 & 43 & 44 & 0 & 46 \\
0 & 0 & 0 & 0 & 0 & 0 \\
61 & 62 & 0 & 0 & 65 & 66
\end{bmatrix}
\]

the arrays are:

\[
AR = (11, 22, 32, 33, 13, 21, 43, 24, 66, 46, 35, 62, 61, 65, 44)
\]
\[
IA = (1, 2, 3, 3, 1, 2, 4, 2, 6, 4, 3, 6, 6, 4)
\]
\[
JA = (1, 2, 2, 3, 3, 1, 3, 4, 6, 6, 5, 2, 1, 5, 4)
\]

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 \(A\).

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 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 $AR$. 

122  ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
• 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:

\[
AR = (11, 13, 24, 22, 21, 32, 33, 43, 46, 61, 62, 66)
\]

\[
IA = (1, 3, 6, 8, 11, 11, 14)
\]

\[
JA = (1, 3, 4, 2, 1, 2, 3, 4, 3, 6, 1, 2, 6)
\]

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:
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 $m \times n$ 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 $a_{ij}$ 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:

$AU = (0, 22, 12, 33, 0, 13, 44, 34, 24, 55, 45, 66, 56, 0, 36)$
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 $a_{ij}$ is the topmost $a_{ij} \neq 0$ in each column $j$, there exists $m$, where $1 \leq m \leq nu$, such that

$$AU(m+j-i) = a_{ij}$$
$$IDU(j) = m \text{ for each } a_{ij}$$
$$IDU(n+1) = nu + 1$$

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

$$AL(m+i-j) = a_{ij}$$
$$IDL(i) = m \text{ for each } a_{ij}$$
$$IDL(n+1) = nl + 1$$

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:

\[
AU = (0, 12, 22, 13, 0, 33, 24, 34, 44, 45, 55, 36, 0, 56, 66)
\]

\[
IDU = (1, 3, 6, 9, 11, 15, 16) \text{ where } nu=15
\]

\[
AL = (*, 21, *, 31, 0, *, 41, 42, 43, *, 54, *, 63, 0, 65, *)
\]

\[
IDL = (1, 3, 6, 10, 12, 16, 17) \text{ where } nl=16
\]

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:

\[
AU = (0, 12, 22, 13, 0, 33, 24, 34, 44, 45, 55, 36, 0, 56, 66)
\]

\[
IDU = (1, 3, 6, 9, 11, 15, 16) \text{ where } nu=15
\]

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 \( a_{ij} \) is the topmost \( a_{ij} \neq 0 \) in each column \( j \),
there exists \( m \), where \( 1 \leq m \leq nu \), such that
\[
AU(m-j+i) = a_{ij}
\]
\[
IDU(j) = m \text{ for each } a_{ij}
\]
\[
IDU(n+1) = nu+1
\]

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
\[
AL(m-i+j) = a_{ij}
\]
\[
IDL(i) = m \text{ for each } a_{ii}
\]
\[
IDL(n+1) = nl+1
\]

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 48 for a description of real and complex numbers, and “How Do You Set Up Your Arrays?” on page 48 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:

\[(x_1, x_2, x_3, \ldots, x_n)\]

#### One-Dimensional Sequence Storage Representation

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 78.

### 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
Two-Dimensional Sequence Storage Representation

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. In the simplest form, it uses a stride of 1 for the first dimension; however, certain subroutines may allow you to specify a stride for the first dimension that is greater than 1. For details, see “How Leading Dimension Is Used for Matrices” on page 83. (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:
Three-Dimensional Sequence Storage Representation

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,32,2056,16,20,10,1.0,AUX,30000)
```

See “How Stride Is Used for Three-Dimensional Sequences” on page 131 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 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 83, such as the starting point, the number of rows and columns, and the leading dimension. In the simplest form, the stride for the first dimension, $inc1$, of a three-dimensional sequence is assumed to be 1, as for matrices; however, certain subroutines may allow you to specify a stride for the first dimension that is greater than 1. 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:

$$\begin{align*}
inc1 &= 1 \\
inc2 &= (E2-E1+1) \\
inc3 &= (E2-E1+1)(F2-F1+1)
\end{align*}$$

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:

<table>
<thead>
<tr>
<th>1.0</th>
<th>8.0</th>
<th>15.0</th>
<th>22.0</th>
<th>29.0</th>
<th>36.0</th>
<th>43.0</th>
<th>50.0</th>
<th>57.0</th>
<th>64.0</th>
<th>71.0</th>
<th>78.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0</td>
<td>9.0</td>
<td>16.0</td>
<td>23.0</td>
<td>30.0</td>
<td>37.0</td>
<td>44.0</td>
<td>51.0</td>
<td>58.0</td>
<td>65.0</td>
<td>72.0</td>
<td>79.0</td>
</tr>
<tr>
<td>3.0</td>
<td>10.0</td>
<td>17.0</td>
<td>24.0</td>
<td>31.0</td>
<td>38.0</td>
<td>45.0</td>
<td>52.0</td>
<td>59.0</td>
<td>66.0</td>
<td>73.0</td>
<td>80.0</td>
</tr>
<tr>
<td>4.0</td>
<td>11.0</td>
<td>18.0</td>
<td>25.0</td>
<td>32.0</td>
<td>39.0</td>
<td>46.0</td>
<td>53.0</td>
<td>60.0</td>
<td>67.0</td>
<td>74.0</td>
<td>81.0</td>
</tr>
<tr>
<td>5.0</td>
<td>12.0</td>
<td>19.0</td>
<td>26.0</td>
<td>33.0</td>
<td>40.0</td>
<td>47.0</td>
<td>54.0</td>
<td>61.0</td>
<td>68.0</td>
<td>75.0</td>
<td>82.0</td>
</tr>
<tr>
<td>6.0</td>
<td>13.0</td>
<td>20.0</td>
<td>27.0</td>
<td>34.0</td>
<td>41.0</td>
<td>48.0</td>
<td>55.0</td>
<td>62.0</td>
<td>69.0</td>
<td>76.0</td>
<td>83.0</td>
</tr>
<tr>
<td>7.0</td>
<td>14.0</td>
<td>21.0</td>
<td>28.0</td>
<td>35.0</td>
<td>42.0</td>
<td>49.0</td>
<td>56.0</td>
<td>63.0</td>
<td>70.0</td>
<td>77.0</td>
<td>84.0</td>
</tr>
</tbody>
</table>
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, $inc_3$, 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:

<table>
<thead>
<tr>
<th>Plane 0:</th>
<th>Plane 1:</th>
<th>Plane 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_{000}$</td>
<td>$a_{001}$</td>
<td>$a_{002}$</td>
</tr>
<tr>
<td>$a_{010}$</td>
<td>$a_{011}$</td>
<td>$a_{012}$</td>
</tr>
<tr>
<td>$a_{200}$</td>
<td>$a_{201}$</td>
<td>$a_{202}$</td>
</tr>
<tr>
<td>$a_{300}$</td>
<td>$a_{301}$</td>
<td>$a_{302}$</td>
</tr>
<tr>
<td><strong>Plane 0:</strong></td>
<td><strong>Plane 1:</strong></td>
<td><strong>Plane 2:</strong></td>
</tr>
<tr>
<td>30.0</td>
<td>51.0</td>
<td>72.0</td>
</tr>
<tr>
<td>31.0</td>
<td>52.0</td>
<td>73.0</td>
</tr>
<tr>
<td>32.0</td>
<td>53.0</td>
<td>74.0</td>
</tr>
<tr>
<td>33.0</td>
<td>54.0</td>
<td>75.0</td>
</tr>
</tbody>
</table>

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 134 for a description of array storage.

**Note:** Three-dimensional sequences are used by the three-dimensional Fourier
transform subroutines and the Multidimensional Fourier transform subroutines. By
specifying certain stride values for $inc_1$, $inc_2$, and $inc_3$ 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 1031 for each
subroutine.
Chapter 4. Coding Your Program

This provides you with information you need to code your Fortran, C, and C++ programs.

Fortran Programs

This 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:

\[
\text{CALL subroutine-name (argument-1, \ldots, argument-n)}
\]

An example of a calling sequence for the SAXPY subroutine might be:

\[
\text{CALL SAXPY (5,A,X,J+INC,Y,1)}
\]

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:

\[
\begin{align*}
\text{DOUBLE PRECISION DASUM,SUM,X} \\
\text{SUM = DASUM (N,X,INCX)}
\end{align*}
\]

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 \( \text{SUM} \).

See the Fortran publications for details on how to code the CALL statement and a function reference.

Setting Up a User-Supplied Subroutine for ESSL in Fortran

Some ESSL numerical quadrature subroutines call a user-supplied subroutine, \( \text{subf} \), 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 \( \text{subf} \) as EXTERNAL in your program.
- You should code the \( \text{subf} \) subroutine to the specifications given in “Programming Considerations for the SUBF Subroutine” on page 1244. For examples of coding a \( \text{subf} \) subroutine in Fortran, see the subroutine descriptions there.
Setting Up Scalar Data in Fortran

Table 46 lists the scalar data types in Fortran that are used for ESSL. Only those types and lengths used by ESSL are listed.

### Table 46. Scalar Data Types in Fortran Programs

<table>
<thead>
<tr>
<th>Terminology Used by ESSL</th>
<th>Fortran Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character item</td>
<td>CHARACTER*1</td>
</tr>
<tr>
<td>'N', 'T', 'C' or 'n', 't', 'c'</td>
<td>'N', 'T', 'C'</td>
</tr>
<tr>
<td>32-bit logical item</td>
<td>LOGICAL or LOGICAL*4</td>
</tr>
<tr>
<td>.TRUE., .FALSE.</td>
<td>.TRUE., .FALSE.</td>
</tr>
<tr>
<td>64-bit logical item</td>
<td>LOGICAL or LOGICAL*8</td>
</tr>
<tr>
<td>.TRUE., .FALSE.</td>
<td>.TRUE., .FALSE.</td>
</tr>
<tr>
<td>32-bit integer</td>
<td>INTEGER or INTEGER*4</td>
</tr>
<tr>
<td>12345, -12345</td>
<td>12345, -12345</td>
</tr>
<tr>
<td>64-bit integer</td>
<td>INTEGER or INTEGER*8</td>
</tr>
<tr>
<td>12345, -12345</td>
<td>12345_8, -12345_8</td>
</tr>
<tr>
<td>Short-precision real number</td>
<td>REAL or REAL*4</td>
</tr>
<tr>
<td>12.345</td>
<td>0.12345E2</td>
</tr>
<tr>
<td>Long-precision real number</td>
<td>DOUBLE PRECISION, REAL, or REAL*8</td>
</tr>
<tr>
<td>12.345</td>
<td>0.12345D2</td>
</tr>
<tr>
<td>Short-precision complex number</td>
<td>COMPLEX or COMPLEX*4</td>
</tr>
<tr>
<td>(123.45, -54321.0)</td>
<td>(123.45E0, -543.21E2)</td>
</tr>
<tr>
<td>Long-precision complex number</td>
<td>COMPLEX or COMPLEX*16</td>
</tr>
<tr>
<td>(123.45, -54321.0)</td>
<td>(123.45D0, -543.21D2)</td>
</tr>
</tbody>
</table>

**Note:**
1. ESSL accepts character data in either upper- or lowercase in its calling sequences.
2. For a 32-bit integer, 64-bit pointer environment, in accordance with the LP64 data model, all ESSL integer arguments remain 32 bits except for the **iusadr** argument for **ERRSET**.
3. Short- and long-precision numbers look the same in this documentation.
4. The default size for INTEGER and LOGICAL data entities that have no length or kind specified is 32 bits. However, the **qintsize=8** compiler option sets the size of such INTEGER and LOGICAL data entities to 64 bits.

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 48 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:

\[
C(I) \leftarrow (a_i, b_i) \quad \text{for a one-dimensional array}
\]
\[
C(1,J) \leftarrow (a_{ij}, b_{ij}) \quad \text{for a two-dimensional array}
\]
\[
C(1,J,K) \leftarrow (a_{ijk}, b_{ijk}) \quad \text{for a three-dimensional array}
\]

**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:

<table>
<thead>
<tr>
<th>Relative Location</th>
<th>Array Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( A(E1) )</td>
</tr>
<tr>
<td>2</td>
<td>( A(E1+1) )</td>
</tr>
<tr>
<td>3</td>
<td>( A(E1+2) )</td>
</tr>
<tr>
<td>( \cdots )</td>
<td>( \cdots )</td>
</tr>
<tr>
<td>( E2-E1+1 )</td>
<td>( A(E2) )</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Relative Location</th>
<th>Array Element Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

**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 334.

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**

<table>
<thead>
<tr>
<th>Array Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  A(E1,F1) (starting column 1)</td>
</tr>
<tr>
<td>2  A(E1+1,F1)</td>
</tr>
<tr>
<td>.  .  .</td>
</tr>
<tr>
<td>E2–E1+1 A(E2,F1)</td>
</tr>
<tr>
<td>(E2–E1+1)+1 A(E1,F1+1) (starting column 2)</td>
</tr>
<tr>
<td>(E2–E1+1)+2 A(E1+1,F1+1)</td>
</tr>
<tr>
<td>.  .  .</td>
</tr>
<tr>
<td>(E2–E1+1)(2) A(E2,F1+1)</td>
</tr>
<tr>
<td>(E2–E1+1)(2)+1 A(E1,F1+2) (starting column 3)</td>
</tr>
<tr>
<td>(E2–E1+1)(2)+2 A(E1+1,F1+2)</td>
</tr>
<tr>
<td>.  .  .</td>
</tr>
<tr>
<td>(E2–E1+1)(F2–F1) A(E2,F2–1)</td>
</tr>
<tr>
<td>(E2–E1+1)(F2–F1)+1 A(E1,F2) (starting column F2–F1+1)</td>
</tr>
<tr>
<td>(E2–E1+1)(F2–F1)+2 A(E1+1,F2)</td>
</tr>
<tr>
<td>.  .  .</td>
</tr>
<tr>
<td>(E2–E1+1)(F2–F1+1) A(E2,F2)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Array Element Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>21</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>31</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>23</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>33</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>34</td>
</tr>
</tbody>
</table>

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:

$$\text{address } \{A(I,J)\} = \text{address } \{A\} + (I-1 + n(J-1))f$$

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:

```fortran
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 1123.

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. 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

<table>
<thead>
<tr>
<th>Array Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>$A(E1,F1,G1)$</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>$A(E1+1,F1,G1)$</td>
</tr>
</tbody>
</table>
\((E_2-E_1+1)(F_2-F_1+1)\)
\[ A(E_2,F_2,G_1) \]
\((E_2-E_1+1)(F_2-F_1+1)+1\)
\[ A(E_1,F_1,G_1+1) \] (starting the second set)
\((E_2-E_1+1)(F_2-F_1+1)+2\)
\[ A(E_1+1,F_1,G_1+1) \]
\((E_2-E_1+1)(F_2-F_1+1)(2)\)
\[ A(E_2,F_2,G_1+1) \]
\((E_2-E_1+1)(F_2-F_1+1)(2)+1\)
\[ A(E_1,F_1,G_1+2) \] (starting the third set)
\((E_2-E_1+1)(F_2-F_1+1)(2)+2\)
\[ A(E_1+1,F_1+2) \]
\((E_2-E_1+1)(F_2-F_1+1)(G_2-G_1)\)
\[ A(E_2,F_2,G_2-1) \]
\((E_2-E_1+1)(F_2-F_1+1)(G_2-G_1)+1\)
\[ A(E_1,F_1,G_2) \] (starting the last set*)
\((E_2-E_1+1)(F_2-F_1+1)(G_2-G_1)+2\)
\[ A(E_1+1,F_1,G_2) \]
\((E_2-E_1+1)(F_2-F_1+1)(G_2-G_1+1)\)
\[ A(E_2,F_2,G_2) \]

* The last set is the \(G_2-G_1+1\) 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:

\[
A = \begin{bmatrix}
111 & 121 \\
112 & 122 \\
113 & 123 \\
114 & 124 \\
211 & 221 \\
212 & 222 \\
213 & 223 \\
214 & 224 \\
311 & 321 \\
312 & 322 \\
313 & 323 \\
314 & 324 
\end{bmatrix}
\]

has its elements arranged in storage as follows:

**Relative Location**

<table>
<thead>
<tr>
<th>Array Element Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>111</td>
</tr>
<tr>
<td>2</td>
<td>211</td>
</tr>
<tr>
<td>3</td>
<td>311</td>
</tr>
<tr>
<td>4</td>
<td>121</td>
</tr>
<tr>
<td>5</td>
<td>221</td>
</tr>
<tr>
<td>6</td>
<td>321</td>
</tr>
<tr>
<td>7</td>
<td>112</td>
</tr>
<tr>
<td>8</td>
<td>212</td>
</tr>
<tr>
<td>9</td>
<td>312</td>
</tr>
</tbody>
</table>
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:

$$\text{address} \{A(I,J,K)\} = \text{address} \{A\} + (I-1 + n(J-1) + mn(K-1)f$$

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 following example shows how to create up to a maximum of eight threads, where each thread calls the DURAND and DGEICD subroutines.

**Note:** Be sure to compile this program with the `xlf_r` command and the `-qnosave` option.
Handling Errors in Your Fortran Program

ESL provides you with flexibilities in handling both input-argument errors and computational errors:

- For input-argument errors 2015, 2030, and 2200 which are optionally-recoverable errors, ESL 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.

• 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" and "Computational Errors in Fortran" on page 144 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" and "Computational Errors in Fortran" on page 144 on each thread that calls ESSL. An example is shown in "Example of Handling Errors in a Multithreaded Application Program" on page 149.

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, 2030, and 2200 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:

```fortran
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:

```fortran
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 1296.

Step 3. Call ERRSAV:

```fortran
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 73. For an example, see “Computational Errors in Fortran Example 3” on page 148, 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, 2030 or 2200. 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 “How Do You Control Error Handling by Setting Values in the ESSL Error Option Table?” on page 71. For a description of the input-argument errors, see “Input-Argument Error Messages(2001-2099)” on page 212. For a description of ERRSET, see Chapter 17, “Utilities,” on page 1293.

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.

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 73. For an example, see “Computational Errors in Fortran Example 3” on page 148, as the use is the same as for computational errors.

**Input-Argument Errors in Fortran 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.
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 1296.

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 1296.

**Step 2. Call ERRSAV:**

```call
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 73.

**Step 3. Call ERRSET:**

```call
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 for which you want to change the default values. For ESSL, `ierno` should be set to one of the eligible values listed in “EINFO (ESSL Error Information-Handler Subroutine)” on page 1296. 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 in a 32-bit integer, 32-bit pointer environment (0_8 or 1_8 in a 32-bit integer, 64-bit pointer environment or a 64-bit integer, 64-bit pointer environment), so a user exit is not taken.

For a list of the default values set in the ESSL error option table, see “How Do You Control Error Handling by Setting Values in the ESSL Error Option Table?” on page 71. For a description of the computational errors, see “Computational Error Messages (2100-2199)” on page 217. For a description of ERRSET, see Chapter 17, “Utilities,” on page 1293.

**Step 4. Call ESSL:**

```call
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.

**Step 5. Call EINFO for Information:**

\[
\text{nnbr} \quad \text{CALL EINFO (icode,inf1)}
\]

- or -

\[
\text{nnbr} \quad \text{CALL EINFO (icode,inf1,inf2)}
\]

This calls the EINFO subroutine, which returns information about certain computational errors, where:

• \text{nnbr} 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.

• \text{icode} is the error code of interest.

• \text{inf1} and \text{inf2} are the integer variables used to receive the information, where \text{inf1} is assigned a value for all errors, and \text{inf2} is assigned a value for some errors.

For a description of EINFO, see “EINFO (ESSL Error Information-Handler Subroutine)” on page 1296.

**Step 6. Check the Values in the Information Receivers:**

These statements check the values returned in the output argument information receivers, \text{inf1} and \text{inf2}, which contain the information about the computational error.

**Step 7. Call ERRSTR:**

\[
\text{CALL ERRSTR (ienro,tabent)}
\]

(This is an optional step.) This calls the ERRSTR subroutine, which stores an entry in the error option table for error number \text{ienro} from an 8-byte storage area, \text{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 73.

**Computational Errors in Fortran Example 1**

This 32-bit integer, 64-bit pointer environment example shows an error code 2104, which returns one piece of information: the index of the last diagonal with nonpositive value (11).
Computational Errors in Fortran Example 2

This 32-bit integer, 64-bit pointer environment example shows an error code 2103, which returns one piece of information: the index of the zero diagonal (I1) found by DGEF.
### Computational Errors in Fortran Example 3

This 32-bit integer, 64-bit pointer environment example shows an error code 2100, which returns two pieces of information: the lower range (I1) and the upper range (I2). It uses ERRSAV and ERRSTR to insulate the effects of the error handling for error 2100 by this program.
Example of Handling Errors in a Multithreaded Application Program

This 32-bit integer, 64-bit pointer environment example shows how to modify the MATINV_EXAMPLE program in “Creating Multiple Threads and Calling ESSL from Your Fortran Program” on page 139 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)
    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_8,2015)
!
! 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
C Programs

This describes how to code your C program.

Calling ESSL Subroutines and Functions in C

This 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 here. 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:

```c
#include <essl.h>
```

If you are planning to create your own threads for the ESSL Thread-Safe or SMP Libraries, 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 156.

Coding the Calling Sequences

In C programs, the ESSL subroutines, not returning a function value, are invoked with the following type of statement:

```c
subroutine-name (argument-1, . . . , argument-n);
```

An example of a calling sequence for SAXPY might be:

```c
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
```
saxpy (5,a,x,incx,y,1);

The ESSL subroutines returning a function value are invoked with the following type of statement:

\[
\text{function-value-name=subroutine-name (argument-1, \ldots, argument-n)};
\]

An example of invoking DASUM might be:

\[
\text{sum = dasum (n,x,incx)};
\]

See the C publications for details about how to code the function calls.

**Passing Arguments in C**

This describes how to pass arguments in your C program.

**About the Syntax Shown in this Documentation**

The argument syntax shown 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 151.

**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 \( \text{incx} \), \( m \), and \( \text{lda} \).)

**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.**
- **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 \( \text{transa} \) and \( \text{transb} \) arguments as strings \( n \) and \( t \), respectively:
  
  \[
  \text{sgeadd (a,5,"n",b,3,"t",c,4,4,3)};
  \]
- **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 158, 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 158 and in step 1 in “Computational Errors in C” on page 163. The two types of ESSL arguments are:
- \texttt{naux} arguments for auxiliary storage
- \texttt{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, \texttt{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 \texttt{subf} subroutine:
- You can code the \texttt{subf} subroutine using only C or Fortran.
- You must declare \texttt{subf} as an external subroutine in your application program.
- You should code the \texttt{subf} subroutine to the specifications given in “Programming Considerations for the SUBF Subroutine” on page 1244. For an example of coding a \texttt{subf} subroutine in C, see Example 1.

### Setting Up Scalar Data in C

Table 47 lists the scalar data types in C that are used for ESSL. Only those types and lengths used by ESSL are listed.

<table>
<thead>
<tr>
<th>Terminology Used by ESSL</th>
<th>C Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character item</td>
<td>char *</td>
</tr>
<tr>
<td>'N', 'T', 'C' or 'n', 't', 'c'</td>
<td>“n”, “t”, “c”</td>
</tr>
<tr>
<td>32-bit logical item</td>
<td>int</td>
</tr>
<tr>
<td>.TRUE., .FALSE.</td>
<td>int</td>
</tr>
<tr>
<td>64-bit logical item</td>
<td>long</td>
</tr>
<tr>
<td>.TRUE., .FALSE.</td>
<td>long</td>
</tr>
<tr>
<td>32-bit integer</td>
<td>int</td>
</tr>
<tr>
<td>12345, -12345</td>
<td>int</td>
</tr>
<tr>
<td>64-bit integer</td>
<td>long</td>
</tr>
<tr>
<td>12345l, -12345l</td>
<td>long</td>
</tr>
<tr>
<td>Short-precision real number</td>
<td>float</td>
</tr>
<tr>
<td>12.345</td>
<td>float</td>
</tr>
<tr>
<td>Long-precision real number</td>
<td>double</td>
</tr>
<tr>
<td>12.345</td>
<td>double</td>
</tr>
<tr>
<td>Short-precision complex number</td>
<td>Specify it as described in “Setting Up Complex Data Types in C” on page 154</td>
</tr>
<tr>
<td>(123.45, -54321.0)</td>
<td>Specify it as described in “Setting Up Complex Data Types in C” on page 154</td>
</tr>
<tr>
<td>Long-precision complex number</td>
<td>Specify it as described in “Setting Up Complex Data Types in C” on page 154</td>
</tr>
<tr>
<td>(123.45, -54321.0)</td>
<td>Specify it as described in “Setting Up Complex Data Types in C” on page 154</td>
</tr>
</tbody>
</table>
Table 47. Scalar Data Types in C Programs (continued)

<table>
<thead>
<tr>
<th>Terminology Used by ESSL</th>
<th>C Equivalent</th>
</tr>
</thead>
</table>

**Note:**
1. ESSL accepts character data in either upper- or lowercase in its calling sequences.
2. There are no equivalent data types for logical data in C. These require special procedures. For details, see "Using Logical Data in C" on page 155.
3. For a 32-bit integer, 64-bit pointer environment, in accordance with the LP64 data model, all ESSL integer arguments remain 32-bits except for the `iusadr` argument for ERRSET.
4. Short- and long-precision numbers look the same in this documentation.
5. If you are using the ESSL header file in a 64-bit integer, 64-bit pointer environment, add `-D_ESV6464` to your compiler command to define the integer and logical arguments as long.

**Setting Up Complex Data Types in C**

You can set up complex data as follows:

- "Complex Data on AIX"
- "Complex Data on Linux (little endian mode)" on page 155

**Complex Data on AIX**

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:

```c
#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
#ifndef _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 151.

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`:

```c
#include <essl.h>
#define CSET(x,a,b) (RE(x)=a, IM(x)=b)
main()
{
```

154 ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
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 Chapter 5, “Processing Your Program,” on page 185.

Complex Data on Linux (little endian mode)
The ESSL header file supports C99 complex floating-point types for complex arithmetic (<complex.h>).

Assuming you are using the ESSL header file, if you declare data items to be of type float_Complex or double_Complex, you can pass them as short- and long-precision complex data to ESSL, respectively.

Using Logical Data in C
Logical data types are not part of the C language; however, some ESSL subroutines require arguments of these data types.

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:

For 32-bit logical arguments
Use this macro definition:
#define FALSE 0
#define TRUE 1

For 64-bit logical arguments
Use this macro definition:
#define FALSE 0l
#define TRUE 1l

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 134. 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 455.

Creating Multiple Threads and Calling ESSL from Your C Program

The 32-bit integer, 64-bit pointer environment example shown below shows how to create two threads, where each thread calls the ISAMAX subroutine. To use the pthread library, you must specify the pthread.h header file as the first include file in your program.

Note: Be sure to compile this program with the cc_r command.

#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.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 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
", i);
    else
        printf("max for sx1 should be 6 = %d
", i);
    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, 2030, and 2200, 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.

- 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 163 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 163 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, 2030, and 2200, 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 ERSSET temporary.

Step 1. Code the Global Statements for ESSL Error Handling:

```c
/* 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:
This declares a pointer, *iusadr*, to be used for the ESSL error exit routine ENOTRM. Also included are declarations 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:**

```c
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 declarations 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:**

```c
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 1296. These statements should be coded only once in the beginning of your program before calls to ERRSET.

**Step 4. Call ERRSAV:**

```c
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 73. For an example, see “Computational Errors in C Example” on page 165, as the use is the same as for computational errors.

**Step 5. Call ERRSET:**

```c
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, iberno should have a value of 2015, 2030, or 2200. 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 “How Do You Control Error Handling by Setting Values in the ESSL Error Option Table?” on page 71. For a description of the input-argument errors, see “Input-Argument Error Messages(2001-2099)” on page 212. For a description of ERRSET, see Chapter 17, “Utilities,” on page 1293.

Step 6. Call ESSL:

```c
irv = name (arg1,...,argn);
if irv == rcl
{
   .
   .
   .
}
```

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.
- irv is the integer variable containing the return code resulting from the computation performed by the ESSL subroutine.
- rcl, 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.

Step 7. Perform the Desired Action:
These are the statements following the test for each value of the return code, returned in irv 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:

```c
errstr (&iberno, storarea);
```

(This is an optional step.) This calls the ERRSTR subroutine, which stores an entry in the error option table for error number iberno 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 73. For an example, see “Computational Errors in C Example” on page 165, as the use is the same as for computational errors.

Input-Argument Errors in C Example
This 32-bit integer, 64-bit pointer environment 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,irange,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 1296.

Step 1. Code the Global Statements for ESSL Error Handling:

```c
/* Code two underscores */
/* before the letters ESVERR */
#define __ESVERR
#include <essl.h>

int ierno,inoal,inomes,itrace,iusadr,irange,irc;
int inf1,inf2,dummy;
char storarea[8];
```

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:

```c
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:

```c
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 1296. These statements should be coded only once in the beginning of your program before calls to ERRSET.

Step 4. Call ERRSAV:

```c
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 73. For an example, see “Computational Errors in C Example” on page 165.

Step 5. Call ERRSET:

```c
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 for which you want to change the default values. For ESSL, ierno should be set to one of the eligible values listed in “EINFO (ESSL Error Information-Handler Subroutine)” on page 1296. 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 in a 32-bit integer, 32-bit pointer environment (0l or 1l in a 32-bit integer, 64-bit pointer environment or a 64-bit integer, 64-bit pointer environment), so a user exit is not taken.

For a list of the default values set in the ESSL error option table, see “How Do You Control Error Handling by Setting Values in the ESSL Error Option Table?” on page 71. For a description of the computational errors, see “Computational Error Messages (2100-2199)” on page 217. For a description of ERRSET, see Chapter 17, “Utilities,” on page 1293.

Step 6. Call ESSL:

```c
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.
• \textit{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.

• \textit{irc} is the integer variable containing the return code resulting from the computation performed by the ESSL subroutine.

• \textit{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.

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.

\textbf{Step 7. Call EINFO for Information:}

\begin{verbatim}
einfo (ierno,&inf1,&inf2);
\end{verbatim}

This calls the EINFO subroutine, which returns information about certain computational errors, where:

• \textit{ierno} is the error code of interest.

• \textit{inf1} and \textit{inf2} are the integer variables used to receive the information, where \textit{inf1} is assigned a value for all errors, and \textit{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 1296.

\textbf{Step 8. Check the Values in the Information Receivers:}

These statements check the values returned in the output argument information receivers, \textit{inf1} and \textit{inf2}, which contain the information about the computational error.

\textbf{Step 9. Call ERRSTR:}

\begin{verbatim}
errstr (&ierno,storarea);
\end{verbatim}

(This is an optional step.) This calls the ERRSTR subroutine, which stores an entry in the error option table for error number \textit{ierno} from an 8-byte storage area, \textit{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 73. For an example, see “Computational Errors in C Example.”

\textbf{Computational Errors in C Example}

This 32-bit integer, 64-bit pointer environment 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.
C++ Programs

This describes how to code your C++ program.

Calling ESSL Subroutines and Functions in C++

This 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 here. 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:

```cpp
#include <essl.h>
```

If you are creating your own threads for the ESSL Thread-Safe or SMP Libraries, 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 173.

Coding the Calling Sequences

In C++ programs, the ESSL subroutines, not returning a function value, are invoked with the following type of statement:

```cpp
subroutine-name (argument-1, ... , argument-n);
```

An example of a calling sequence for SAXPY might be:

```cpp
saxpy (5,a,x,incx,y,1);
```

The ESSL subroutines returning a function value are invoked with the following type of statement:

```cpp
function-value-name = subroutine-name (argument-1, ... , argument-n);
```

An example of invoking DASUM might be:

```cpp
sum = dasum (n,x,incx);
```

See the C++ publications for details about how to code the function calls.

Passing Arguments in C++

This describes how to pass arguments in your C++ program.

About the Syntax Shown in this Documentation

The argument syntax shown 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 166.

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 scalar argument is a scalar data item,
it must be passed by reference as shown below. This is true for all scalar data
types: real, complex, and so forth.

The ESSL header file supports two alternatives:
• The arguments are declared to be type reference in the function prototype. This
  is the default. Following is an example of how you can call DURAND using this
  alternative:
  
durand (seed, n, x);
• The arguments are declared as pointers in the function prototype. If you wish to
  use this alternative, you must define _ESVCPT using one of the following
  methods:
  – Define _ESVCPT in your program using a #define statement, as shown
  below:
  
#define _ESVCPT
  This statement is coded with your global declares and must be coded before
  the #include statement for the ESSL header file.
  – Define _ESVCPT at compile time by using the job processing procedure
    described in “C++ Program Procedures on AIX” on page 188 and “C++
    Program Procedures on Linux (little endian mode)” on page 196.

Following is an example of how you can call DURAND using this alternative:

durand (&seed, n, x);

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 1244. For an example of
coding a subf subroutine in C++, see Example 1.
### Setting Up Scalar Data in C++

[Table 48](#) lists the scalar data types in C++ that are used for ESSL. Only those types and lengths used by ESSL are listed.

**Table 48. Scalar Data Types in C++ Programs**

<table>
<thead>
<tr>
<th>Terminology Used by ESSL</th>
<th>C++ Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character item</td>
<td>char *</td>
</tr>
<tr>
<td>'N', 'T', 'C' or 'n', 't', 'c'</td>
<td>“N”, “T”, “C”</td>
</tr>
<tr>
<td>32-bit logical item</td>
<td>int</td>
</tr>
<tr>
<td>.TRUE., .FALSE.</td>
<td>For additional information, see &quot;Using Logical Data in C++&quot; on page 172.</td>
</tr>
<tr>
<td>64-bit logical item</td>
<td>long</td>
</tr>
<tr>
<td>.TRUE., .FALSE.</td>
<td>For additional information, see &quot;Using Logical Data in C++&quot; on page 172.</td>
</tr>
<tr>
<td>32-bit integer</td>
<td>int</td>
</tr>
<tr>
<td>12345, -12345</td>
<td></td>
</tr>
<tr>
<td>64-bit integer</td>
<td>long</td>
</tr>
<tr>
<td>12345l, -12345l</td>
<td></td>
</tr>
<tr>
<td>Short-precision real number</td>
<td>float</td>
</tr>
<tr>
<td>12.345</td>
<td></td>
</tr>
<tr>
<td>Long-precision real number</td>
<td>double</td>
</tr>
<tr>
<td>12.345</td>
<td></td>
</tr>
<tr>
<td>Short-precision complex number</td>
<td>complex &lt;float&gt;^5^, float_Complex^7^, or as described in &quot;On AIX—Setting Up Short-Precision Complex Data Types If You Are Using the IBM Open Class Complex Mathematics Library in C++&quot; on page 170.</td>
</tr>
<tr>
<td>(123.45, -54321.0)</td>
<td></td>
</tr>
<tr>
<td>Long-precision complex number</td>
<td>complex &lt;double&gt;^5^, double_Complex^7^, or complex^6^</td>
</tr>
<tr>
<td>(123.45, -54321.0)</td>
<td></td>
</tr>
</tbody>
</table>

**Note:**

1. ESSL accepts character data in either upper- or lowercase in its calling sequences.
2. There are no equivalent data types for logical data in C++. These require special procedures. For details, see "Using Logical Data in C++" on page 172.
3. For a 32-bit integer, 64-bit pointer environment, in accordance with the LP64 data model, all ESSL integer arguments remain 32-bits except for the iusadr argument for ERRSET.
4. Short- and long-precision numbers look the same in this documentation.
5. This data type is defined in file <complex>.
6. This data type is defined in file <complex.h> (supported only on AIX).
7. This data type is defined in <complex.h> (supported only on Linux little endian mode).
8. If you are using the ESSL header file in a 64-bit integer, 64-bit pointer environment, add -D_ESV6464 to your compiler command to define the integer and logical arguments as long.
Using Complex Data in C++
On AIX, the ESSL header file supports both the IBM Open Class® Complex Mathematics Library (<complex.h>) and the Standard Numerics Library facilities for complex arithmetic (<complex.h>.

On Linux (little endian mode), the ESSL header file supports both the Standard Numerics Library and C99 floating-point types for complex arithmetic (<complex.h>.

On AIX—Selecting the <complex> or <complex.h> Header File
Although the header files <complex> and <complex.h> are similar in purpose, they are mutually incompatible and cannot be simultaneously used.

If you wish to use the Standard Numerics Library facilities for complex arithmetic, you must do one of the following:
• Code the #include statement for the Standard Numerics Library facilities for complex arithmetic (#include <complex>) in your program prior to coding the #include statement for the ESSL header file.
• Define _ESV_COMPLEX_, using one of the following methods:
  – Define _ESV_COMPLEX_ in your program using a #define statement, as shown below:
    ```cpp
    #define _ESV_COMPLEX_
    ```
    This statement is coded with your global declares and must be coded before the #include statement for the ESSL header file.
  – Define _ESV_COMPLEX_ at compile time by using the job processing procedures described in “Processing Your Program,” on page 185.

If you take none of the preceding steps, the ESSL header file will use the IBM Open Class Complex Mathematics Library. The ESSL header file will also use the IBM Open Class Complex Mathematics Library if you:
• Code the #include statement for the IBM Open Class Complex Mathematics Library (#include<complex.h>) in your program prior to coding the #include statement for the ESSL header file.

On AIX—Setting Up Short-Precision Complex Data Types If You Are Using the IBM Open Class Complex Mathematics Library in C++
Short-precision complex 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:
```cpp
#ifndef _CMPLX
class cmplx
{
    private:
        float _re, _im;
    public:
        cmplx() { _re = 0.0; _im = 0.0; }
        cmplx(float r, float i = 0.0) { _re = r; _im = i; }
        friend inline float sreal(const cmplx& a) { return a._re; }
        friend inline float simag(const cmplx& a) { return a._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 166.

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:

```c
#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, your definition must conform to the following rules:

- The definition must have exactly two variables of type *float* representing the real and imaginary parts of the short-precision complex data. For example:
  ```c
  struct cmplx { float _re, _im; }
  ```
- The definition cannot include an explicit destructor.

In addition, you must do one of the following:

- Define _CMPLX in your program using the #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, as follows:
  ```c
  #define _CMPLX
  ```
- Use the job processing procedures described in Chapter 5, “Processing Your Program,” on page 185 to define your short-precision complex data at compile time.

**On Linux (little endian mode) —Selecting the <complex> or <complex.h> Header File**

Although the header files <complex> and <complex.h> are similar in purpose, they are mutually incompatible and cannot be simultaneously used.

If you wish to use the C99 complex floating-point types for complex arithmetic, you must do one of the following:

- Code the #include statement for the C99 complex floating point types (#include <complex.h>) in your program prior to coding the #include statement for the ESSL header file.
- Define _ESV_COMPLEX99_, using one of the following methods:
- Define _ESV_COMPLEX99_ in your program using a #define statement, as shown below:
  ```
  #define _ESV_COMPLEX99_
  ```
  This statement is coded with your global declares and must be coded before the #include statement for the ESSL header file.
- Define _ESV_COMPLEX99_ at compile time by using the job processing procedures described in Chapter 5, “Processing Your Program,” on page 185.

If you take none of the preceding steps, the ESSL header file will use the Standard Numerics Library. The ESSL header file will also use the Standard Numerics Library if you code the #include statement for the Standard Numerics Library (#include<complex.h>) in your program prior to coding the #include statement for the ESSL header file.

**Using Logical Data in C++**

Logical data types are not part of the C++ language; however, some ESSL subroutines require arguments of these data types.

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:

**For 32-bit logical arguments**

Use this macro definition:

```
#define FALSE 0
#define TRUE 1
```

**For 64-bit logical arguments**

Use this macro definition:

```
#define FALSE 0l
#define TRUE 1l
```

**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 134. 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 455.
Creating Multiple Threads and Calling ESSL from Your C++ Program

The 32-bit integer, 64-bit pointer environment example shown below shows how to create two threads, where each thread calls the ISAMAX subroutine. To use the pthreads library, you must remember to code the pthread.h header file in your C++ program.

Note: Be sure to compile this program with the xlC_r command.
/* 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 <unistd.h>
#include <stdlib.h>
int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
}

/* Create structure for argument list */
struct arg_list {
  int n;
  float *x;
  int incx;
};

int main()
{
  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;
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, 2030, and 2200 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.

- 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 180 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 180 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, 2030, and 2200, 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:**

```c++
/* 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;
```
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 statements are required to call the ESSL error exit routine ENOTRM as an external reference in your program.

**Step 2. Declare the Variables:**

```c
FN 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:**

```c
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 1296. These statements should be coded only once in the beginning of your program before calls to ERRSET.

**Step 4. Call ERRSAV:**

```c
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 73. For an example, see “Computational Errors in C++ Example” on page 182, as the use is the same as for computational errors.

Step 5. Call ERRSET:

```c
errset (ierno,inool,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 (inool)
- 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, 2030, or 2200. 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 “How Do You Control Error Handling by Setting Values in the ESSL Error Option Table?” on page 71. For a description of the input-argument errors, see “Input-Argument Error Messages(2001-2099)” on page 212. For a description of ERRSET, see Chapter 17, “Utilities,” on page 1293.

Step 6. Call ESSL:

```c
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.

**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:**

```c
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 73. For an example, see “Computational Errors in C++ Example” on page 182, as the use is the same as for computational errors.

**Input-Argument Errors in C++ Example**
This 32-bit integer, 64-bit pointer environment 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>
#ifdef __linux
#include <iostream>
#else
#include <iostream.h>
#endif
#include <stdio.h>
extern "C" int enotrm(int &,int &);
extern "C" typedef int (*FN)(int &,int &);

/*DECLARE THE VARIABLES*/
int main()
{
FN iusadr;
int ierno,inoal,inomes,itrace,irange,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 1296.

Step 1. Code the Global Statements for ESSL Error Handling:

```c
/* Code one underscore */
/* before the letters ESVERR */
#define _ESVERR
#ifdef __linux
#include <iostream>
#else
#include <iostream.h>
#endif
#include <stdio.h>
#include <essl.h>

int ierno, inoal, inomes, itrace, iusadr, irange, irc;
int inf1, inf2, dummy;
char storarea[8];
dummy = 0;
einfo (0,dummy,dummy);
errsav (ierno,storarea);
```

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:

```c
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:

```c
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 1296. These statements should be coded only once in the beginning of your program before calls to ERRSET.

Step 4. Call ERRSAV:

```c
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 73. For an example, see “Computational Errors in C++ Example” on page 182.

Step 5. Call ERRSET:

```c
errset (ierno, inoa1, 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 (inoa1)
- How many times each error message can be printed (inomes)

ERRSET must be called for each error code for which you want to change the default values. For ESSL, ierno should be set to one of the eligible values listed in “EINFO (ESSL Error Information-Handler Subroutine)” on page 1296. To allow your program to continue after an error in the specified range occurs, inoa1 must be set to a value greater than 1. For ESSL, iusadr should be specified as either 0 or 1 in a 32-bit environment (0l or 1l in a 32-bit integer, 64-bit pointer environment or a 64-bit integer, 64-bit pointer environment), so a user exit is not taken.

For a list of the default values set in the ESSL error option table, see “How Do You Control Error Handling by Setting Values in the ESSL Error Option Table?” on page 71. For a description of the computational errors, see “Computational Error Messages (2100-2199)” on page 217. For a description of ERRSET, see Chapter 17, “Utilities,” on page 1293.

Step 6. Call ESSL:

```c
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.
• \textit{irc} is the integer variable containing the return code resulting from the computation performed by the ESSL subroutine.

• \textit{rc1}, \textit{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.

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.

\textbf{Step 7. Call EINFO for Information:}

\begin{verbatim}
einfo (ierno,inf1,inf2);
\end{verbatim}

This calls the EINFO subroutine, which returns information about certain computational errors, where:

• \textit{ierno} is the error code of interest.

• \textit{inf1} and \textit{inf2} are the integer variables used to receive the information, where \textit{inf1} is assigned a value for all errors, and \textit{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 1296.

\textbf{Step 8. Check the Values in the Information Receivers:}

These statements check the values returned in the output argument information receivers, \textit{inf1} and \textit{inf2}, which contain the information about the computational error.

\textbf{Step 9. Call ERRSTR:}

\begin{verbatim}
erstr (ierno,storarea);
\end{verbatim}

(This is an optional step.) This calls the ERRSTR subroutine, which stores an entry in the error option table for error number \textit{ierno} from an 8-byte storage area, \textit{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 73. For an example, see “Computational Errors in C++ Example.”

\textbf{Computational Errors in C++ Example}

This 32-bit integer, 64-bit pointer environment example shows an error code 2105, which returns one piece of information: the index of the pivot element (\textit{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>
#if defined(__linux)
#include <iostream>
#else
#include <iostream.h>
#endif
#include <stdio.h>

/*DECLARE THE VARIABLES*/
int main()
{
int ierno,inoal,inomes,itrace,irange,irc;
long int iusadr;
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 = 0l; /*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*/
    .
    .
}
ierno = 2105;
errstr (ierno,sav2105);
.
}

/*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 ERRSAVE STATEMENT*/
Chapter 5. Processing Your Program

This describes the **ESSL-specific changes** you need to make to your job procedures for compiling, linking, and running your program.

You can use any procedures you are currently using to compile, link, and run your Fortran, C, and C++ programs, as long as you make the necessary modifications required by ESSL.

Processing Your Program on AIX

The following notes apply to processing your program on AIX.

**Notes:**

1. The default search path for the ESSL libraries is: `/usr/lib`. (Note that `/lib` is a symbolic link to `/usr/lib`.)
   If the libraries are installed somewhere else, add the path name of that directory to the beginning of the `LIBPATH` environment variable, being careful to keep `/usr/lib` in the path. The correct `LIBPATH` setting is needed both for linking and executing the program.
   For example, if you installed the ESSL libraries in `/home/me/lib` you would issue ksh commands similar to the following in order to compile and link a program:
   ```bash
   LIBPATH=/home/me/lib:/usr/lib
   export LIBPATH
   xlf -o myprog myprog.f -lessl
   ```
   After setting the `LIBPATH` command, the `/home/me/lib` directory is the directory that gets searched first for the necessary libraries. This same search criterion is used at both compile and link time and run time.

2. For the ESSL SMP Libraries, you can use the `XLSMPOPTS` or `OMP_NUM_THREADS` environment variable to specify options which affect SMP execution. For details, see the IBM Compiler publications.

3. If you are accessing ESSL from a 32-bit integer, 64-bit pointer environment program or a 64-bit integer, 64-bit pointer environment program, you must add the `-q64` compiler option.

4. If you are accessing ESSL from a 64-bit integer, 64-bit pointer environment program, you may want to use the `-qintsize=8` compiler option.

5. ESSL supports the XL Fortran compile-time option `-qextname`. For details, see the Fortran manuals.

6. Fortran 90 programmers may be interested in the `-qessl` compiler option which allows the use of ESSL routines in place of Fortran 90 intrinsic procedures. For details, see the Fortran manuals.

7. In your job procedures, you must use only the required software products listed in "Required Software Products on AIX" on page 9.

Fortran Program Procedures on AIX

You do not need to modify your existing Fortran compilation procedures when using ESSL.
When linking and running your program, you must modify your existing job procedures for ESSL in order to set up the necessary libraries.

If you are accessing ESSL from a Fortran program, you can compile and link using the commands shown in the table below.

Table 49. Fortran Compile Commands on AIX

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>Fortran Compile Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>32-bit integer,</td>
<td>xlf_r -0 -qnosave xyz.f -lessl</td>
</tr>
<tr>
<td></td>
<td>32-bit pointer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32-bit integer,</td>
<td>xlf_r -0 -qnosave -q64 xyz.f -lessl</td>
</tr>
<tr>
<td></td>
<td>64-bit pointer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64-bit integer,</td>
<td>xlf_r -0 -qnosave -q64 xyz.f -lessl6464</td>
</tr>
<tr>
<td></td>
<td>64-bit pointer</td>
<td></td>
</tr>
<tr>
<td>SMP</td>
<td>32-bit integer,</td>
<td>xlf_r -0 -qnosave xyz.f -lesslsmp</td>
</tr>
<tr>
<td></td>
<td>32-bit pointer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32-bit integer,</td>
<td>xlf_r -0 -qnosave -q64 xyz.f -lesslsmp</td>
</tr>
<tr>
<td></td>
<td>64-bit pointer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64-bit integer,</td>
<td>xlf_r -0 -qnosave -q64 xyz.f -lesslsmp6464</td>
</tr>
<tr>
<td></td>
<td>64-bit pointer</td>
<td></td>
</tr>
</tbody>
</table>

where xyz.f is the name of your Fortran program.

If you want to use the FFTW Wrapper libraries with your Fortran program, the header file fftw3.f contains the constant definitions used by the FFTW wrappers. To use these definitions, you can do one of the following:

- Add the following line to your Fortran application:

  ```
  include "fftw3.f"
  ```

- Imbed the fftw3.f header file in your application.

You can compile and link with the FFTW Wrapper libraries using the command shown in the table below (assuming that the FFTW Wrapper header files were installed in /usr/local/include).

Table 50. Fortran Compile Commands on AIX for use with FFTW Wrapper libraries

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>Fortran Compile Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>32-bit integer,</td>
<td>xlf_r -0 -qnosave xyz.f -lessl</td>
</tr>
<tr>
<td></td>
<td>32-bit pointer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32-bit integer,</td>
<td>xlf_r -0 -qnosave -q64 xyz.f -lessl</td>
</tr>
<tr>
<td></td>
<td>64-bit pointer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64-bit integer,</td>
<td>xlf_r -0 -qnosave -q64 xyz.f -lessl6464</td>
</tr>
<tr>
<td></td>
<td>64-bit pointer</td>
<td></td>
</tr>
<tr>
<td>SMP</td>
<td>32-bit integer,</td>
<td>xlf_r -0 -qnosave xyz.f -lesslsmp</td>
</tr>
<tr>
<td></td>
<td>32-bit pointer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32-bit integer,</td>
<td>xlf_r -0 -qnosave -q64 xyz.f -lesslsmp</td>
</tr>
<tr>
<td></td>
<td>64-bit pointer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64-bit integer,</td>
<td>xlf_r -0 -qnosave -q64 xyz.f -lesslsmp6464</td>
</tr>
<tr>
<td></td>
<td>64-bit pointer</td>
<td></td>
</tr>
</tbody>
</table>

For additional information on the FFTW Wrapper libraries, see Appendix C, “FFTW Version 3.1.2 to ESSL Wrapper Libraries,” on page 1349.
ESSL supports the XL Fortran compile-time option `-qextname`. For details, see the Fortran manuals.

**C Program Procedures on AIX**

The ESSL header file `essl.h`, used for C and C++ programs, is installed in the `/usr/include` directory. If you are using the ESSL header file in a 64-bit integer, 64-bit pointer environment, add `-D_ESV6464` to your compile and link command.

If you do want to specify your own definitions for short- and long-precision complex data, add `-D_CMPLX` and `-D_DCMPLX`, respectively, to your compile and link command. Otherwise, you automatically use the definitions of short- and long-precision complex data provided in the ESSL header file (as shown in the table below).

When linking and running your program, you must modify your existing job procedures for ESSL, to set up the necessary libraries.

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>C Compile Command</th>
</tr>
</thead>
</table>
| Serial       | 32-bit integer, 32-bit pointer | `cc_r -O xyz.c -lessl`
|               | 32-bit integer, 64-bit pointer | `cc_r -O -D_CMPLX -D_DCMPLX xyz.c -lessl`
|               | 64-bit integer, 64-bit pointer | `cc_r -O -D_ESV6464 -q64 xyz.c -lessl6464`
|               |                             | `cc_r -O -D_ESV6464 -D_CMPLX -D_DCMPLX -q64 xyz.c -lessl6464`
| SMP          | 32-bit integer, 32-bit pointer | `cc_r -O xyz.c -lessl64`
|               | 32-bit integer, 64-bit pointer | `cc_r -O -D_CMPLX -D_DCMPLX xyz.c -lessl64`
|               | 64-bit integer, 64-bit pointer | `cc_r -O -D_ESV6464 -q64 xyz.c -lessl6464`
|               |                             | `cc_r -O -D_ESV6464 -D_CMPLX -D_DCMPLX -q64 xyz.c -lessl6464`

If you want to use the FFTW Wrapper libraries with your C program, you must use header file `fftw3_essl.h` instead of `fftw3.h`. You can compile and link with the FFTW Wrapper libraries using the command shown in the table below (assuming that the FFTW Wrapper header files were installed in `/usr/local/include`).

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>C Compile Command</th>
</tr>
</thead>
</table>
| Serial       | 32-bit integer, 32-bit pointer | `cc_r -O xyz.c -lessl -I/usr/local/include -lfftw3_essl -L/usr/local/lib -lm`
|               | 32-bit integer, 64-bit pointer | `cc_r -O -q64 xyz.c -lessl -I/usr/local/include -lfftw3_essl_64 -L/usr/local/lib -lm`
|               | 64-bit integer, 64-bit pointer | `cc_r -O -q64 xyz.c -lessl -I/usr/local/include -lfftw3_essl_64 -L/usr/local/lib -lm`
### Table 52. C Compile and Link Commands on AIX for use with FFTW Wrapper Libraries (continued)

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>C Compile Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMP</td>
<td>32-bit integer,</td>
<td><code>cc_r -O xyz.c -lesslsmp</code></td>
</tr>
<tr>
<td></td>
<td>32-bit pointer</td>
<td><code>-lfftw3_essl -L/usr/local/lib -lm</code></td>
</tr>
<tr>
<td></td>
<td>32-bit integer,</td>
<td><code>cc_r -O -q64 xyz.c -lesslsmp</code></td>
</tr>
<tr>
<td></td>
<td>64-bit pointer</td>
<td><code>-lfftw3_essl_64 -L/usr/local/lib -lm</code></td>
</tr>
</tbody>
</table>

For additional information on the FFTW Wrapper libraries, see Appendix C, “FFTW Version 3.1.2 to ESSL Wrapper Libraries,” on page 1349.

### C++ Program Procedures on AIX

The ESSL header file `essl.h`, used for C and C++ programs, is installed in the `/usr/include` directory. When using ESSL, the compiler option `-qnocinc=/usr/include/essl` must be specified.

If you are using the ESSL header file in a 64-bit integer, 64-bit pointer environment, add `-D_ESV6464` to your compile and link command.

If you are using the IBM Open Class Complex Mathematics Library, 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 compile and link commands (as shown in the table below). Otherwise, ESSL will use the IBM Open Class Complex Mathematics Library or the Standard Numerics Library, as described in “On AIX—Selecting the `<complex>` or `<complex.h>` Header File” on page 170.

If you prefer to explicitly specify that you want to use the Standard Numerics Library facilities for complex arithmetic, add `-D_ESV_COMPLEX_` to your compile and link command as shown in the table below.

The ESSL header file supports two alternatives for declaring scalar output arguments. By default, the arguments are declared to be type reference. If you prefer for them to be declared as pointers, add `-D_ESVCPTR` to your compile and link commands as shown in the table below.

When linking and running your program, you must modify your existing job procedures for ESSL, to set up the necessary libraries.
<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>C++ Compile Command</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32-bit integer, 32-bit pointer</td>
<td>xlc_r -O xyz.C -lessl -qnocinc=/usr/include/essl</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xlc_r -O -D_CMPLX xyz.C -lessl -qnocinc=/usr/include/essl</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xlc_r -O -D_ESV_COMPLEX_ xyz.C -lessl -qnocinc=/usr/include/essl</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xlc_r -O -D_ESVCPTR xyz.C -lessl -qnocinc=/usr/include/essl</td>
</tr>
<tr>
<td></td>
<td>32-bit integer, 64-bit pointer</td>
<td>xlc_r -O -q64 xyz.C -lessl -qnocinc=/usr/include/essl</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xlc_r -O -D_CMPLX -q64 xyz.C -lessl -qnocinc=/usr/include/essl</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xlc_r -O -D_ESV_COMPLEX_ -q64 xyz.C -lessl -qnocinc=/usr/include/essl</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xlc_r -O -D_ESVCPTR -q64 xyz.C -lessl -qnocinc=/usr/include/essl</td>
</tr>
<tr>
<td></td>
<td>64-bit integer, 64-bit pointer</td>
<td>xlc_r -O -D_ESV6464 -q64 xyz.C -lessl -qnocinc=/usr/include/essl</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xlc_r -O -D_ESV6464 -D_CMPLX -q64 xyz.C -lessl -qnocinc=/usr/include/essl</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xlc_r -O -D_ESV6464 -D_ESV_COMPLEX_ -q64 xyz.C -lessl -qnocinc=/usr/include/essl</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xlc_r -O -D_ESV6464 -D_ESVCPTR -q64 xyz.C -lessl -qnocinc=/usr/include/essl</td>
</tr>
</tbody>
</table>
If you want to use the FFTW Wrapper libraries with your C++ program, you must use header file ftw3_essl.h instead of ftw3.h. You can compile and link with the FFTW Wrapper libraries using the compile and link commands shown in the table below (assuming that the FFTW Wrapper header files were installed in /usr/local/include).

**Table 54. C++ Compile and Link Commands on AIX for Use with FFTW Wrapper Libraries**

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>C++ Compile Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>32-bit integer, 32-bit pointer</td>
<td>xlc_r -O xyz.C -lessl -qnocinc=/usr/include/essl -I/usr/local/include -lfftw3_essl -L/usr/local/lib -lm</td>
</tr>
<tr>
<td></td>
<td>32-bit integer, 64-bit pointer</td>
<td>xlc_r -O -q64 xyz.C -lessl -qnocinc=/usr/include/essl -I/usr/local/include -lfftw3_essl_64 -L/usr/local/lib -lm</td>
</tr>
<tr>
<td>SMP</td>
<td>32-bit integer, 64-bit pointer</td>
<td>xlc_r -O -D_ESV6464 -q64 xyz.C -lesslmp6464 -qnocinc=/usr/include/essl -I/usr/local/include -lfftw3_essl_64 -L/usr/local/lib -lm</td>
</tr>
<tr>
<td></td>
<td>64-bit integer, 64-bit pointer</td>
<td>xlc_r -O -D_ESV6464 -D_ESV_COMPLEX_ -q64 xyz.C -lesslmp6464 -qnocinc=/usr/include/essl -I/usr/local/include -lfftw3_essl_64 -L/usr/local/lib -lm</td>
</tr>
</tbody>
</table>
Processing Your Program on Linux (little endian mode)

The following notes apply to processing your program on Linux.

Notes:
1. The default search paths for the ESSL shared libraries are as follows:

<table>
<thead>
<tr>
<th>Environment</th>
<th>Shared Library Default Search Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit integer, 64-bit pointer</td>
<td>/usr/lib64</td>
</tr>
<tr>
<td>64-bit integer, 64-bit pointer</td>
<td>/usr/lib64</td>
</tr>
</tbody>
</table>

If the shared libraries are in another location, you must set the link-time and run-time library search paths. There are two ways to set these search paths:

- Use one of the following compile/link options:
  - **-R** (or **-rpath**)
    - Writes the specified run-time library search paths into the executable program.
  - **-L**
    - Searches the library search paths at link time, but does not write them into the executable as run-time library search paths.

—or—

- Use one of the following environment variables:
  - **LD_LIBRARY_PATH**
    - Specifies the directories that are to be searched for libraries at run time.
  - **LD_RUN_PATH**
    - Specifies the directories that are to be searched for libraries at both link and run time.

For example, if you copied the ESSL 32-bit/64-bit pointer libraries in /home/me/lib64, you would issue commands similar to the following in order to compile and link a program:

```bash
LD_LIBRARY_PATH=/home/me/lib64:$LD_LIBRARY_PATH
LD_RUN_PATH=/home/me/lib64:$LD_RUN_PATH
export LD_LIBRARY_PATH
export LD_RUN_PATH
xlf_r -o myprog myprog.f -lessl
```

The result would be that the /home/me/lib64 directory is the directory that gets searched at link time and run time.

For more information on link options and environment variables, see the manpage for the **ld** command.

2. The default search path for the ESSL header files is **/usr/include**. If the header files are in another location, you must set the compile-time search path by setting the search path with **-I**.

3. If you changed Makefile or Makefile.gcc to install the FFTW Wrapper libraries in /usr/local/lib instead of /usr/local/lib64 (see Appendix C, “FFTW Version 3.1.2 to ESSL Wrapper Libraries,” on page 1349), then you must specify **-L/usr/local/lib** instead of **-L/usr/local/lib64** in the commands in Table 56 on page 193, Table 58 on page 194, Table 59 on page 195, and Table 62 on page 197.

4. For the ESSL SMP and SMP CUDA Libraries, you can use the **XLSMPOPTS** or **OMP_NUM_THREADS** environment variable to specify options which affect SMP execution. For details, see the IBM Compiler publications.
5. If you are accessing ESSL from a 64-bit integer, 64-bit pointer environment program, you may want to use the `-qintsize=8` compiler option.

6. ESSL supports the XL Fortran compile-time option `-qextname`. For details, see the Fortran publications.

7. Fortran 90 programmers may be interested in the `-qessl` compiler option which allows the use of ESSL routines in place of Fortran 90 intrinsic procedures. For details, see the Fortran manuals.

8. The commands in the table below assume that you installed:
   - The IBM compilers in the default directory, `/opt/ibm`. If you used different directories, you need to make the appropriate changes to the `-L` and `-R` options.
   - ESSL in the default directory `/opt/ibmmath`. If you used different directories, you need to make the appropriate changes to the `-I`, `-L`, and `-R` options.
   - The CUDA SDK in the default directory, `/usr/local/cuda`. If you used different directories, you need to make the appropriate changes to the `-L` and `-R` options.

9. In your job procedures, you must use only the required software products listed in "Required Software Products on Linux" on page 10.

**Fortran Program Procedures on Linux (little endian mode)**

You do not need to modify your existing Fortran compilation procedures when using ESSL.

When linking and running your program, you must modify your existing job procedures for ESSL in order to set up the necessary libraries.

If you are accessing ESSL from a Fortran program, you can compile and link using the commands shown in the table below.

**Note:** ESSL supports the XL Fortran compile-time option `-qextname`. For details, see the Fortran manuals.

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>Fortran Compile Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>32-bit integer, 64-bit pointer</td>
<td><code>xlf_r -O -qnosave xyz.f -lessl</code></td>
</tr>
<tr>
<td></td>
<td>64-bit integer, 64-bit pointer</td>
<td><code>xlf_r -O -qnosave xyz.f -lessl6464</code></td>
</tr>
<tr>
<td>SMP</td>
<td>32-bit integer, 64-bit pointer</td>
<td><code>xlf_r -O -qnosave -qsmp xyz.f -lesslsmp</code></td>
</tr>
<tr>
<td></td>
<td>64-bit integer, 64-bit pointer</td>
<td><code>xlf_r -O -qnosave -qsmp xyz.f -lesslsmp -lxlsmp</code></td>
</tr>
<tr>
<td>SMP CUDA</td>
<td>32-bit integer, 64-bit pointer</td>
<td><code>xlf_r -O -qnosave -qsmp xyz.f -lesslsmpcuda -lcublas -lcudart -L/usr/local/cuda/11b64 -R/usr/local/cuda/11b64</code></td>
</tr>
<tr>
<td></td>
<td>64-bit integer, 64-bit pointer</td>
<td><code>xlf_r -O -qnosave xyz.f -lesslsmpcuda -lxlsmp -lcublas -lcudart -L/usr/local/cuda/11b64 -R/usr/local/cuda/11b64</code></td>
</tr>
</tbody>
</table>

where `xyz.f` is the name of your Fortran program.
If you want to use the FFTW Wrapper libraries with your Fortran program, the header file ftw3.f contains the constant definitions used by the FFTW wrappers. To use these definitions, you can do one of the following:

- Add the following line to your Fortran application:

  ```fortran
  include "ftw3.f"
  ```

- Imbed the ftw3.f header file in your application.

You can compile and link with the FFTW Wrapper libraries using the command shown in the table below (assuming that the FFTW Wrapper header files were installed in /usr/local/include).

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>Fortran Compile Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>32-bit integer,</td>
<td>xlf_r -O -qnosave xy.z.f -lessl</td>
</tr>
<tr>
<td></td>
<td>64-bit pointer</td>
<td>-I/usr/local/include -lfftw3_essl -L/usr/local/lib64</td>
</tr>
<tr>
<td>SMP</td>
<td>32-bit integer,</td>
<td>xlf_r -O -qnosave xy.z.f -lessl</td>
</tr>
<tr>
<td></td>
<td>64-bit pointer</td>
<td>-I/usr/local/include -lfftw3_essl -L/usr/local/lib64</td>
</tr>
</tbody>
</table>

For additional information on the FFTW Wrapper libraries, see Appendix C, “FFTW Version 3.1.2 to ESSL Wrapper Libraries,” on page 1349.

### C Program Procedures on Linux (little endian mode)

If you are using the ESSL header file in a 64-bit integer, 64-bit pointer environment, add `-D_ESV6464` to your compile and link command.

When linking and running your program, you must modify your existing job procedures for ESSL in order to set up the necessary libraries.

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>C Compile Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>32-bit integer,</td>
<td>cc_r -O x.y.c</td>
</tr>
<tr>
<td></td>
<td>64-bit pointer</td>
<td>-less1 -lxlf90_r -lxlfmath</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-L/opt/ibm/xlsmp/xlsmp_version.release/lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-L/opt/ibm/xlf/xlf_version.release/lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-R/opt/ibm/lib</td>
</tr>
<tr>
<td></td>
<td>64-bit integer,</td>
<td>cc_r -O -D_ESV6464 x.y.c</td>
</tr>
<tr>
<td></td>
<td>64-bit pointer</td>
<td>-less16464 -lxlf90_r -lxlfmath</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-L/opt/ibm/xlsmp/xlsmp_version.release/lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-L/opt/ibm/xlf/xlf_version.release/lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-R/opt/ibm/lib</td>
</tr>
<tr>
<td>SMP</td>
<td>32-bit integer,</td>
<td>cc_r -O x.y.c</td>
</tr>
<tr>
<td></td>
<td>64-bit pointer</td>
<td>-less3lsmp -lxlf90_r -lxlsmp -lxlfmath</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-L/opt/ibm/xlsmp/xlsmp_version.release/lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-L/opt/ibm/xlf/xlf_version.release/lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-R/opt/ibm/lib</td>
</tr>
<tr>
<td></td>
<td>64-bit integer,</td>
<td>cc_r -O -D_ESV6464 x.y.c</td>
</tr>
<tr>
<td></td>
<td>64-bit pointer</td>
<td>-less3lsmp6464 -lxlf90_r -lxlsmp -lxlfmath</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-L/opt/ibm/xlsmp/xlsmp_version.release/lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-L/opt/ibm/xlf/xlf_version.release/lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-R/opt/ibm/lib</td>
</tr>
</tbody>
</table>
Table 57. C Compile and Link Commands on Linux (little endian mode) (continued)

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>C Compile Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMP CUDA</td>
<td>32-bit integer,</td>
<td>`cc_r -O xyz.c</td>
</tr>
<tr>
<td></td>
<td>64-bit pointer</td>
<td>-lesslmpcuda -lxlf90_r -lxlmp -lxlfmath -lcublas -lcudart</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-L/usr/local/cuda/lib64</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-R/usr/local/cuda/lib64</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-L/opt/ibm/xlmp/xlmp_version.release/lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-L/opt/ibm/xlf/xlf_version.release/lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-R/opt/ibm/lib</td>
</tr>
</tbody>
</table>

**Note:** In the commands listed in the table above, you must specify the following values:

- `xlf_version.release`
  - 15.1.4 or later
- `xlsmp_version.release`
  - 4.1.4 or later

If you want to use the FFTW Wrapper libraries with your C program, you must use header file `fftw3_essl.h` instead of `fftw3.h`. You can compile and link with the FFTW Wrapper libraries using the command shown in the table below (assuming that the FFTW Wrapper header files were installed in `/usr/local/include`).

Table 58. C Compile and Link Commands on Linux for Use with FFTW Wrapper Libraries (little endian mode)

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>C Compile Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>32-bit integer,</td>
<td>`cc_r -O xyz.c</td>
</tr>
<tr>
<td></td>
<td>64-bit pointer</td>
<td>-lesslmp -lxlf90_r -lxlfmath</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-L/opt/ibm/xlmp/xlmp_version.release/lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-L/opt/ibm/xlf/xlf_version.release/lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-R/opt/ibm/lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-I/usr/local/include -lfftw3_essl -L/usr/local/lib64</td>
</tr>
</tbody>
</table>

| SMP          | 32-bit integer,     | `cc_r -O xyz.c                                        |
|              | 64-bit pointer      | -lesslmp -lxlf90_r -lxlfmath                           |
|              |                     | -L/opt/ibm/xlmp/xlmp_version.release/lib               |
|              |                     | -L/opt/ibm/xlf/xlf_version.release/lib                 |
|              |                     | -R/opt/ibm/lib                                        |
|              |                     | -I/usr/local/include -lfftw3_essl -L/usr/local/lib64   |

**Note:** In the commands listed in the table above, you must specify the following values:

- `xlf_version.release`
  - 15.1.4 or later
- `xlsmp_version.release`
  - 4.1.4 or later

For additional information on the FFTW Wrapper libraries, see Appendix C, “FFTW Version 3.1.2 to ESSL Wrapper Libraries,” on page 1349.

If you want to use gcc compile and link commands, use the commands shown in Table 59 on page 195.

---

194 ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
**Table 59. gcc Compile and Link Commands on Linux (little endian mode)**

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>C Compile Command</th>
</tr>
</thead>
</table>
| Serial       | 32-bit integer, 64-bit pointer | gcc `xyz.c`  
- `lessl`  
- `-lxlf90_r`  
- `-lxlfmath`  
- `-lm`  
- `-L/opt/ibm/xlsmp/xlsmp_version.release/lib`  
- `-L/opt/ibm/xlf/xlf_version.release/lib`  
- `-R/opt/ibm/lib` |
|              | 64-bit integer, 64-bit pointer | gcc `xyz.c`  
- `-D_ESV6464`  
- `-lxlf90_r`  
- `-lxlfmath`  
- `-lm`  
- `-L/opt/ibm/xlsmp/xlsmp_version.release/lib`  
- `-L/opt/ibm/xlf/xlf_version.release/lib`  
- `-R/opt/ibm/lib` |
| SMP*         | 32-bit integer, 64-bit pointer | gcc `xyz.c`  
- `lesslsmpl`  
- `-lxlf90_r`  
- `-lxlfmath`  
- `-lm`  
- `-L/opt/ibm/xlsmp/xlsmp_version.release/lib`  
- `-L/opt/ibm/xlf/xlf_version.release/lib`  
- `-R/opt/ibm/lib` |
|              | 64-bit integer, 64-bit pointer | gcc `xyz.c`  
- `-D_ESV6464`  
- `-lxlf90_r`  
- `-lxlfmath`  
- `-lm`  
- `-L/opt/ibm/xlsmp/xlsmp_version.release/lib`  
- `-L/opt/ibm/xlf/xlf_version.release/lib`  
- `-R/opt/ibm/lib` |
| SMP CUDA*    | 32-bit integer, 64-bit pointer | gcc `xyz.c`  
- `lesslsmplcuda`  
- `-lxlf90_r`  
- `-lxlfmath`  
- `-lm`  
- `-lcublas`  
- `-lcudart`  
- `-L/usr/local/cuda/lib64`  
- `-R/usr/local/cuda/lib64`  
- `-L/opt/ibm/xlsmp/xlsmp_version.release/lib`  
- `-L/opt/ibm/xlf/xlf_version.release/lib`  
- `-R/opt/ibm/lib` |

* The ESSL SMP libraries require XL OpenMP runtime. The gcc OpenMP runtime is not compatible with XL OpenMP runtime.

**Note:** In the commands listed in Table 59, you must specify the following values:

- `xlf_version.release`  
  15.1.4 or later
- `xlsmp_version.release`  
  4.1.4 or later

If you want to use the FFTW Wrapper libraries with your C program, you must use header file `ftw3_essl.h` instead of `ftw3.h`. You can compile and link with the FFTW Wrapper libraries using the command shown in the table below (assuming that the FFTW Wrapper header files were installed in `/usr/local/include`).

**Table 60. gcc Compile and Link Commands on Linux for Use with FFTW Wrapper Libraries (little endian mode)**

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>C Compile Command</th>
</tr>
</thead>
</table>
| Serial       | 32-bit integer, 64-bit pointer | gcc `xyz.c`  
- `lessl`  
- `-lxlf90_r`  
- `-lxlfmath`  
- `-lm`  
- `-L/opt/ibm/xlsmp/xlsmp_version.release/lib`  
- `-L/opt/ibm/xlf/xlf_version.release/lib`  
- `-R/opt/ibm/lib`  
- `-l/usr/local/include -lfftw3_essl -L/usr/local/lib64` |
Table 60. gcc Compile and Link Commands on Linux for Use with FFTW Wrapper Libraries (little endian mode) (continued)

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>C Compile Command</th>
</tr>
</thead>
</table>
| SMP*         | 32-bit integer, 64-bit pointer | gcc xyz.c
- lesslsmpl -lxlf90 -lxlsmp -lxlfmath -lm
- L/opt/ibm/xlsmp/xlsmp_version.release/lib
- L/opt/ibm/xlf/xlf_version.release/lib
- R/opt/ibm/lib
- I/usr/local/include -lfftw3_essl -L/usr/local/lib64 |

* The ESSL SMP libraries require XL OpenMP runtime. The gcc OpenMP runtime is not compatible with XL OpenMP runtime.

Note: In the commands listed in Table 60 on page 195 you must specify the following values:

- `xlf_version.release` 15.1.4 or later
- `xlsmp_version.release` 4.1.4 or later

C++ Program Procedures on Linux (little endian mode)

The ESSL header file supports two alternatives for handling complex floating-point arguments. By default the Standard Numerics Library complex floating-point types are used. If you prefer to use the C99 complex floating-point types, add `-D_ESV_COMPLEX99_` to your compile and link commands.

The ESSL header file supports two alternatives for declaring scalar output arguments. By default, the arguments are declared to be type reference. If you prefer for them to be declared as pointers, add `-D_ESVCPT` to your compile and link commands.

If you are using the ESSL header file in a 64-bit integer, 64-bit pointer environment, add `-D_ESV6464` to your compile and link command.

When linking and running your program, you must modify your existing job procedures for ESSL, to set up the necessary libraries.

Table 61. C++ Compile and Link Commands on Linux (little endian mode)

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>C++ Compile Command</th>
</tr>
</thead>
</table>
| Serial       | 32-bit integer, 64-bit pointer | x1C_r -O xyz.C
- lessl -lxlf90 -lxlfmath
- L/opt/ibm/xlsmp/xlsmp_version.release/lib
- L/opt/ibm/xlf/xlf_version.release/lib
- R/opt/ibm/lib |
|              | 64-bit integer, 64-bit pointer | x1C_r -O -D_ESV6464 xyz.C
- lessl6464 -lxlf90 -lxlfmath
- L/opt/ibm/xlsmp/xlsmp_version.release/lib
- L/opt/ibm/xlf/xlf_version.release/lib
- R/opt/ibm/lib |
Table 61. C++ Compile and Link Commands on Linux (little endian mode) (continued)

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>C++ Compile Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMP</td>
<td>32-bit integer, 64-bit pointer</td>
<td><code>x1C_r -O xyz.C -lesslmp -lxlf90_r -lxlsmp -lxlfmath -L/opt/ibm/xlsmp/xlsmp_version.release/lib -R/opt/ibm/lib</code></td>
</tr>
<tr>
<td></td>
<td>64-bit integer, 64-bit pointer</td>
<td><code>x1C_r -D_ESV6464 xyz.C -lesslmp6464 -lxlf90_r -lxlsmp -lxlfmath -L/opt/ibm/xlsmp/xlsmp_version.release/lib -R/opt/ibm/lib</code></td>
</tr>
</tbody>
</table>

**Note:** In the commands listed in the table above, you must specify the following values:

- `xlf_version.release` 15.1.4 or later
- `xlsmp_version.release` 4.1.4 or later

If you want to use the FFTW Wrapper libraries with your C++ program, you must use header file `fftw3_essl.h` instead of `fftw3.h`. You can compile and link with the FFTW Wrapper libraries using the command shown in the table below (assuming that the FFTW Wrapper header files were installed in `/usr/local/include`).

Table 62. C++ Compile and Link Commands on Linux for Use with FFTW Wrapper Libraries (little endian mode)

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>C++ Compile Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>32-bit integer, 64-bit pointer</td>
<td><code>x1C_r -O xyz.C -lessl -lxlf90_r -lxlfmath -lm -L/opt/ibm/xlsmp/xlsmp_version.release/lib -R/opt/ibm/lib -I/usr/local/include -lfftw3_essl -L/usr/local/lib64</code></td>
</tr>
<tr>
<td>SMP</td>
<td>32-bit integer, 64-bit pointer</td>
<td><code>x1C_r -O xyz.C -lesslmp -lxlf90_r -lxlsmp -lxlfmath -lm -L/opt/ibm/xlsmp/xlsmp_version.release/lib -R/opt/ibm/lib -I/usr/local/include -lfftw3_essl -L/usr/local/lib64</code></td>
</tr>
</tbody>
</table>

**Note:** In the commands listed in the table above, you must specify the following values:

- `xlf_version.release` 15.1.4 or later
For additional information on the FFTW Wrapper libraries, see Appendix C, “FFTW Version 3.1.2 to ESSL Wrapper Libraries,” on page 1349.

If you want to use g++ compile and link commands, use the commands shown in Table 63.

Table 63. g++ Compile and Link Commands on Linux (little endian mode)

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>C++ Compile Command</th>
</tr>
</thead>
</table>
| Serial       | 32-bit integer, 64-bit pointer | g++  
- lessl -lxlf90_r -lxlfmath -lm  
- L:/opt/ibm/xlsmp/xlsmp_version.release/lib  
- L:/opt/ibm/xlf/xlf_version.release/lib  
- R:/opt/ibm/lib |
|              | 64-bit integer, 64-bit pointer | g++  
- D_ESV6464  
- lessl6464 -lxlf90_r -lxlfmath -lm  
- L:/opt/ibm/xlsmp/xlsmp_version.release/lib  
- L:/opt/ibm/xlf/xlf_version.release/lib  
- R:/opt/ibm/lib |
| SMP*         | 32-bit integer, 64-bit pointer | g++  
- lesslsm -lxlf90_r -lxlsm -lxlfmath -lm  
- L:/opt/ibm/xlsmp/xlsmp_version.release/lib  
- L:/opt/ibm/xlf/xlf_version.release/lib  
- R:/opt/ibm/lib |
|              | 64-bit integer, 64-bit pointer | g++  
- D_ESV6464  
- lesslsm6464 -lxlf90_r -lxlsm -lxlfmath -lm  
- L:/opt/ibm/xlsmp/xlsmp_version.release/lib  
- L:/opt/ibm/xlf/xlf_version.release/lib  
- R:/opt/ibm/lib |
| SMP CUDA*    | 32-bit integer, 64-bit pointer | g++  
- lesslsmcuda -lxlf90_r -lxlsm -lxlfmath -lm -lcublas -lcudart  
- L:/usr/local/cuda/lib64  
- R:/usr/local/cuda/lib64 |
|              | 64-bit integer, 64-bit pointer | g++  
- D_ESV6464  
- lesslsmcuda -lxlf90_r -lxlsm -lxlfmath -lm -lcublas -lcudart  
- L:/usr/local/cuda/lib64  
- R:/usr/local/cuda/lib64 |

* The ESSL SMP libraries require XL OpenMP runtime. The gcc OpenMP runtime is not compatible with XL OpenMP runtime.

Note: In the commands listed in Table 63, you must specify the following values:

- xlf_version.release  
  15.1.4 or later
- xlsmp_version.release  
  4.1.4 or later

If you want to use the FFTW Wrapper libraries with your C program, you must use header file fftw3_essl.h instead of fftw3.h. You can compile and link with the FFTW Wrapper libraries using the command shown in the table below (assuming that the FFTW Wrapper header files were installed in /usr/local/include).
Table 64. g++ Compile and Link Commands on Linux for Use with FFTW Wrapper Libraries (little endian mode)

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>C++ Compile Command</th>
</tr>
</thead>
</table>

* The ESSL SMP libraries require XL OpenMP runtime. The gcc OpenMP runtime is not compatible with XL OpenMP runtime.

Note: In the commands listed in Table 64, you must specify the following values:

- `xlf_version.release` 15.1.4 or later
- `xlsmp_version.release` 4.1.4 or later
Chapter 6. Migrating Your Programs

This explains what is required to migrate your application programs to the current release of ESSL.

Migrating Programs from ESSL for Linux on Power Version 5 Release 4 to Version 5 Release 5

The calling sequence for all subroutines except the CBLAS subroutines in ESSL Version 5 Release 4 and Version 5 Release 5 are identical, and therefore no changes to your application programs for these subroutines are required. However, if your programs contain calls to CBLAS functions using the arguments in Table 65 changes are required. As shown in Table 65 CBLAS enumerated types have changed to the following type definitions:

<table>
<thead>
<tr>
<th>Table 65. CBLAS enumerated types changing to type definitions for ESSL Version 5 Release 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESSL VSR4 enumerated type</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>enum CBLAS_ORDER {CblasRowMajor=101, CblasColMajor=102};</td>
</tr>
<tr>
<td>enum CBLAS_TRANSPOSE {CblasNoTrans=111, CblasTrans=112, CblasConjTrans=113};</td>
</tr>
<tr>
<td>enum CBLAS_UPLO {CblasUpper=121, CblasLower=122};</td>
</tr>
<tr>
<td>enum CBLAS_DIAG {CblasNonUnit=131, CblasUnit=132};</td>
</tr>
<tr>
<td>enum CBLAS_SIDE {CblasLeft=141, CblasRight=142};</td>
</tr>
</tbody>
</table>

1 You may continue to use the name CLAS_ORDER, but it is recommended that you change to CBLAS_LAYOUT.

For example, in ESSL Version 5 Release 5 the following type of variable definition is no longer supported:

```c
enum CBLAS_UPLO variable;
```

Instead, use something like the following:

```c
CBLAS_UPLO variable;
```

Migrating Programs from ESSL for Linux on Power Version 5 Release 3.2 to Version 5 Release 4

The calling sequences for the subroutines in ESSL Version 5 Release 3.2 and ESSL Version 5 Release 4 are identical; therefore, no changes to your application programs are required.

Migrating Programs from ESSL for Linux on Power Version 5 Release 3.1 to Version 5 Release 3.2

The calling sequences for the subroutines in ESSL Version 5 Release 3.1 and ESSL Version 5 Release 3.2 are identical; therefore, no changes to your application programs are required.

Migrating Programs from ESSL for Linux on Power Version 5 Release 2 or ESSL Version 5 Release 3 to Version 5 Release 3.1

The following support is not provided for ESSL 5.3.1 (little endian mode)

- 32-bit applications
• C applications that use the ESSL header file and user-defined definitions for short- and long-precision complex data. You must change these applications to use C99 complex floating point types instead.

No source code changes to your other application programs are required to migrate to ESSL 5.3.1.

Migrating Programs from ESSL for Linux on Power Version 5 Release 2 to Version 5 Release 3

The calling sequences for the subroutines in ESSL Version 5 Release 3 and ESSL Version 5 Release 2 are identical; therefore, no changes to your application programs are required.

Migrating Programs from ESSL for AIX 5.1 and ESSL for Linux on Power Version 5 Release 1.1 to Version 5 Release 2

Source code changes may be required in C or C++ application programs that call the cpocon or zpocon subroutines because the prototype contained in the ESSL Header Files (essl.h) prior to ESSL 5.2 incorrectly specified WORK as real instead of complex. This has been corrected in the ESSL 5.2 essl.h file.

The following non-LAPACK-conforming subroutines are no longer provided in ESSL 5.2. To run with ESSL 5.2, existing applications using these subroutines require source code changes to replace these subroutines as shown in Table 66:

Table 66. Replacing Non-LAPACK-Conforming subroutines with LAPACK subroutines

<table>
<thead>
<tr>
<th>Non-LAPACK Conforming Subroutines in ESSL 5.1</th>
<th>Corresponding ESSL LAPACK Subroutines in ESSL 5.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGEEV, DGEEV, CGEEV, ZGEEV</td>
<td>SGEEVX, DGEEVX, CGEEVX, ZGEEVX</td>
</tr>
<tr>
<td></td>
<td>See “SGEEV, DGEEV, CGEEV, ZGEEV, SGEEVX, DGEEVX, CGEEVX, ZGEEVX” (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix)” on page 942</td>
</tr>
<tr>
<td>SSPEV, DSPEV, CHPEV, ZHPEEV</td>
<td>SSPEVX, DSPEVX, CHPEVX, ZHPEVX</td>
</tr>
<tr>
<td>SSPSV, DSPSV, CHPSV, ZHPSV</td>
<td>See “SSYEV, DSYEV, CHEEV, ZHEEV, SSYEVX, DSYEVX, CHEEVX, ZHEEVX” (Eigenvalues and, Optionally, the Eigenvectors of a Real Symmetric or Complex Hermitian Matrix)” on page 959</td>
</tr>
</tbody>
</table>

ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
Table 66. Replacing Non-LAPACK-Conforming subroutines with LAPACK subroutines (continued)

<table>
<thead>
<tr>
<th>Non-LAPACK Conforming Subroutines in ESSL 5.1</th>
<th>Corresponding ESSL LAPACK Subroutines in ESSL 5.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGEGV, DGEGV</td>
<td>SGGEV, DGGEV</td>
</tr>
<tr>
<td></td>
<td>See &quot;SGGEV, DGGEV, CGGEV, ZGGEV, SGGEVX, DGGEVX, CGGEVX, and ZGGEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix Generalized Eigenproblem)&quot; on page 991</td>
</tr>
<tr>
<td>SSYGV, DSYGV</td>
<td>SSYGVX, DSYGVX</td>
</tr>
<tr>
<td></td>
<td>See &quot;SSPGVX, DSPGVX, CHPGVX, ZHPGVX, SSYGVX, DSYGVX, CHEGVX, and ZHEGVX (Eigenvalues and, Optionally, the Eigenvectors of a Positive Definite Real Symmetric or Complex Hermitian Generalized Eigenproblem)&quot; on page 1008</td>
</tr>
</tbody>
</table>

Existing applications that do not use these non_LAPACK-conforming subroutines will work without source code changes for migration from ESSL 5.1 to ESSL 5.2.

Migrating Programs from ESSL for Linux on Power Version 5 Release 1 to Version 5 Release 1.1

The calling sequences for the subroutines in ESSL Version 5 Release 1 and ESSL Version 5 Release 1.1 are identical; therefore, no changes to your application programs are required.

Migrating Programs from ESSL Version 4 Release 4 to Version 5 Release 1

The Processor-Independent Formulas for SCFTD and DCFTD for NAUX2 have been corrected. For the corrected formulas, see "SCFTD and DCFTD (Multidimensional Complex Fourier Transform)" on page 1036.

Otherwise, the calling sequences for the subroutines in ESSL Version 4 Release 4 and ESSL Version 5 Release 1 are identical; therefore, no changes to your application programs are required.

Migrating Programs from ESSL Version 4 Release 3 to Version 4 Release 4

The calling sequences for the subroutines in ESSL Version 4 Release 3 and ESSL Version 4 Release 4 are identical; therefore, no changes to your application programs are required.
Migrating Programs from ESSL Version 4 Release 2.2 or Later to ESSL Version 4 Release 3

For 32-bit integer, 32-bit pointer environments and 32-bit integer, 64-bit pointer environments, the calling sequences for the subroutines in ESSL Version 4 Release 2.2 or later are identical to those in ESSL Version 4 Release 3; therefore, no changes to those in your application programs are required.

If you wish to use the new ESSL Serial and SMP Libraries that support a 64-bit integer, 64-bit pointer environment, note the following:

- You must modify your application to use 64-bit integers and logicals instead of 32-bit integers and logicals.
- You may need to increase the size of naux and lwork to obtain a larger workspace. (See "Setting Up Auxiliary Storage When Dynamic Allocation Is Not Used" on page 53.)
- You must add -D_ESV6464 to your C and C++ compile commands. (See Chapter 5, “Processing Your Program,” on page 185.)
- You must change the library specified in your compile command to either -lesslsm6464 or -lessl6464, as appropriate. (See Chapter 5, “Processing Your Program,” on page 185.)

Migrating Programs from ESSL Version 4 Release 2.1 to Version 4 Release 2.2

In the ESSL Blue Gene Library, the Fourier Transform subroutines and the Convolutions and Correlations subroutines require that the alignments of certain arrays do not change between initialization and computation. If the array alignment does change, in some cases error message 2152 will be issued and your program will terminate. If you want your program to continue processing with degraded performance, use ERRSET with an ESSL error exit routine, ENOTRM, to make error 2152 recoverable.

For all other subroutines, the calling sequences for the subroutines in ESSL Version 4 Release 2.1 and ESSL Version 4 Release 2.2 are identical; therefore, no changes to your application programs are required.

Migrating Programs from ESSL Version 4 Release 2 to Version 4 Release 2.1

The calling sequences for the subroutines in ESSL Version 4 Release 2 and ESSL Version 4 Release 2.1 are identical; therefore, no changes to your application programs are required.

Migrating Programs from ESSL Version 4 Release 1 to Version 4 Release 2

The calling sequences for the subroutines in ESSL Version 4 Release 1 and ESSL Version 4 Release 2 are identical; therefore, no changes to your application programs are required.

ESSL Version 4 Release 2 does not support SLES8. In most cases, binary compatibility does not exist between SLES8 and SLES9. Therefore, SLES8 applications must be recompiled and rebuilt on SLES9.
On Linux, if you are accessing ESSL from a C or C++ program, you must change your compile and link commands so that they specify IBM XL Fortran Enterprise Edition Version 9.1 for Linux.

**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 From One Hardware Platform to Another**

This describes all the aspects of migrating your ESSL application programs from one hardware platform to another.

**Auxiliary Storage**

The minimum amount of auxiliary storage returned by ESSL error handling may vary from one hardware platform to another for the following subroutines:

- all the Fourier transform subroutines
- SCONF
- SCORF
- 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 service level to another, one ESSL library to another, or one hardware platform to another may not be bitwise-identical. The results, however, are mathematically equivalent.
Migrating from Other Libraries to ESSL

This 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. However you may need to make some coding changes for certain subroutines.

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 Subroutin Library provided with AIX, your calling sequences do not need to be changed.

Migrating from LAPACK

ESSL contains some subroutines that conform to the LAPACK interface. If you are using these subroutines, no coding changes are needed to migrate to ESSL.

Migrating from FFTW Version 3.1.2

ESSL includes header files and C and Fortran wrappers in source form for a subset of the FFTW Version 3.1.2 subroutines. If you want to use these wrappers, you must include the header file fftw3_essl.h instead of fftw3.h. For additional information on the FFTW Wrapper libraries, see Appendix C, “FFTW Version 3.1.2 to ESSL Wrapper Libraries,” on page 1349.

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 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.

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:
   - ESSL for AIX
     5765-H25
   - ESSL for Linux
     5765-L51
   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.
   - On AIX
     Enter the following command:
     
     ```
     oslevel -r
     ```
   - On Linux
     Enter the following command:
     
     ```
     uname -a
     ```
   This is important information that speeds up the correct routing of your call.
4. The names and versions of key products being run.
   - On AIX
     Enter the following command:
     
     ```
     lslpp -h product
     ```
     where the appropriate values of `product` are listed in Table 67 on page 208.
On Linux

Enter the following command:

```
rpm -q package
```

where the appropriate values of `package` are listed in Table 67.

**Table 67. Product File Set and Package Names**

<table>
<thead>
<tr>
<th>Descriptive Name</th>
<th>Product File Sets on AIX</th>
<th>Product Packages on Linux little endian mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESSL</td>
<td>essl.*</td>
<td>essl.rte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>essl.3264.rte</td>
</tr>
<tr>
<td></td>
<td></td>
<td>essl.3264.rtecuda</td>
</tr>
<tr>
<td></td>
<td></td>
<td>essl.6464.rte</td>
</tr>
<tr>
<td>XL Fortran Runtime Environment</td>
<td>xlfrte</td>
<td>libxlf</td>
</tr>
<tr>
<td>SMP Runtime Environment</td>
<td>xlsmp.rte</td>
<td>libxlsmp</td>
</tr>
<tr>
<td>XL Fortran compiler</td>
<td>xlcmp.15.1.0</td>
<td>xlf.15.1.4</td>
</tr>
<tr>
<td>XL C compiler</td>
<td>xlcmp.13.1.0</td>
<td>xlc.13.1.4</td>
</tr>
<tr>
<td>XL C++ compiler</td>
<td>xlcmp.13.1.0</td>
<td>xlc.13.1.4</td>
</tr>
</tbody>
</table>

5. The message that is returned when an error is detected.
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 product, but your site may be using its own translated message catalogs. The 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`.

The ESSL message catalogs are in English, and are located in the following directories:

**On AIX**
```
/usr/lib/nls/msg/C
/usr/lib/nls/msg/En_US
```

208 ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
On Linux (little endian mode)
/opt/ibmmath/essl/5.5/msg/en_US/essl.cat
/usr/share/locale/en_US.UTF-8/essl.cat
/usr/share/locale/en_US/essl.cat
/usr/share/locale/en/essl.cat
/usr/share/locale/C/essl.cat

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 AIX General Programming Concepts: Writing and Debugging Programs.

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 (2200-2299)
- Computational errors (2100-2199)
- Resource errors (2401-2499)
- Informational and Attention messages (2600-2699)
- Miscellaneous errors (2700-2799)

Program Exceptions


ESSL Input-Argument Error Messages

If you receive an error message in the form 2538-20nn or 2538–22nn, 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, 2030 or 2200. For details on how to do this, see Chapter 4, "Coding Your Program," on page 133.
- 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.

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(2001-2099)" on page 212 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.
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 2538-21nn, 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(2100-2199)” on page 217 for a complete description of the information in each message. Return code values are described under “Error Conditions” for each ESSL subroutine.

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, “Coding Your Program,” on page 133.

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 2538-24nn, 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.
- For a 32-bit integer, 32-bit pointer environment application on AIX, consider specifying the -bmaxdata binder option when linking your program. For details see the Fortran publications.
- Check the setting of your user ID's user limit (ulimit). (See the AIX Commands Reference).

**ESSL Informational and Attention Messages**

If you receive a message in the form 2538-26nn, 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 210.

**Miscellaneous Error Messages**

If you receive a message in the form 2538-27nn, 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 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 209.
Message Conventions

This describes the message conventions for the ESSL product.

About Upper- and Lowercase

Literals, such as, ‘N’, ‘T’, ‘U’, and so forth, appear in the messages in this documentation 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:

$rtn-name : 2538-mmmn$
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. If $rtn-name$ is ESSL, this indicates that at least one ESSL subroutine encountered this 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
22—Input-argument 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 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(2001-2099)

Note: There are more input-argument error messages listed in "Input-Argument Error Messages(2200-2299)" on page 219.

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2538-2001</td>
<td>The number of elements (ARG NO. _) in a vector must be greater than or equal to zero.</td>
</tr>
<tr>
<td>2538-2002</td>
<td>The stride (ARG NO. _) for a vector must be nonzero.</td>
</tr>
<tr>
<td>2538-2003</td>
<td>The number of rows (ARG NO. _) in a matrix must be greater than or equal to zero.</td>
</tr>
<tr>
<td>2538-2004</td>
<td>The number of columns (ARG NO. _) in a matrix must be greater than or equal to zero.</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>2538-2005</td>
<td>The size of the leading dimension (ARG NO. _) of an array must be greater than zero.</td>
</tr>
<tr>
<td>2538-2006</td>
<td>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.</td>
</tr>
<tr>
<td>2538-2007</td>
<td>The degree of a polynomial (ARG NO. _) must be greater than or equal to zero.</td>
</tr>
<tr>
<td>2538-2008</td>
<td>The number of elements (ARG NO. _) to be scanned must be greater than or equal to 2.</td>
</tr>
<tr>
<td>2538-2009</td>
<td>The number of elements (ARG NO. _) in a vector to be processed must be greater than or equal to 3.</td>
</tr>
<tr>
<td>2538-2010</td>
<td>The transform length (ARG NO. _) must be a power of 2.</td>
</tr>
<tr>
<td>2538-2011</td>
<td>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. _).</td>
</tr>
<tr>
<td>2538-2012</td>
<td>The transform length (ARG NO. _) must be less than or equal to ( ).</td>
</tr>
<tr>
<td>2538-2013</td>
<td>The transform length (ARG NO. _) must be greater than or equal to ( ).</td>
</tr>
<tr>
<td>2538-2014</td>
<td>The routine must be initialized with the present value of (ARG NO. _).</td>
</tr>
<tr>
<td>2538-2015</td>
<td>The number of elements (ARG NO. _) in a work array must be greater than or equal to ( ).</td>
</tr>
<tr>
<td>2538-2016</td>
<td>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'.</td>
</tr>
<tr>
<td>2538-2017</td>
<td>The dimension (ARG NO. _) of the matrices must be greater than or equal to zero.</td>
</tr>
<tr>
<td>2538-2018</td>
<td>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. _).</td>
</tr>
<tr>
<td>2538-2019</td>
<td>The number of sequences (ARG NO. _) must be greater than zero.</td>
</tr>
<tr>
<td>2538-2020</td>
<td>(ARG NO. _) must be nonzero.</td>
</tr>
<tr>
<td>2538-2021</td>
<td>The storage control switch (ARG NO. _) must be 1, 2, 3, or 4.</td>
</tr>
<tr>
<td>2538-2022</td>
<td>(ARG NO. _) must be less than ( ).</td>
</tr>
<tr>
<td>2538-2023</td>
<td>The outer loop increment (ARG NO. _) must be greater than or equal to zero.</td>
</tr>
<tr>
<td>2538-2024</td>
<td>The stride (ARG NO. _) for a vector must be greater than or equal to zero.</td>
</tr>
<tr>
<td>2538-2025</td>
<td>The stride (ARG NO. _) for a vector must be greater than zero.</td>
</tr>
<tr>
<td>2538-2026</td>
<td>The stride (ARG NO. _) for a vector must be greater than or equal to ( ).</td>
</tr>
<tr>
<td>2538-2027</td>
<td>The order (ARG NO. _) of a matrix must be greater than or equal to zero.</td>
</tr>
<tr>
<td>2538-2028</td>
<td>The job option argument (ARG NO. _) must be one of the following: 0; 1, or 2; 0, 1, 2, or 3; 0, 1, 2, 10, 11, or 12; 0, 1, 10, or 11; 0, 1, 20, or 21; 0, 1, 10, 20, 21, 30, or 31; 0, 1, 2, 3, or 4].</td>
</tr>
<tr>
<td>2538-2029</td>
<td>The job option argument (ARG NO. _) must be 0 or 1.</td>
</tr>
<tr>
<td>2538-2030</td>
<td>The transform length (ARG NO. _) is not an allowed value. The next higher allowed value is ( ).</td>
</tr>
<tr>
<td>2538-2031</td>
<td>The resulting convolution length obtained from ARG NO. 10 = ( ), ARG NO. 11 = ( ), ARG NO. 13 = ( ), and ARG NO. 14 = ( ) must be less than ( ).</td>
</tr>
</tbody>
</table>
The size of the leading dimension (ARG NO. _) of the matrix must be greater than or equal to (\_), the bandwidth constraint.

The lower bandwidth (ARG NO. _) must be greater than or equal to zero.

The upper bandwidth (ARG NO. _) must be greater than or equal to zero.

The half-band bandwidth (ARG NO. _) must be greater than or equal to zero.

The lower bandwidth (ARG NO. _) must be less than the order (ARG NO. _) of the matrix.

The upper bandwidth (ARG NO. _) must be less than the order (ARG NO. _) of the matrix.

The half-band bandwidth (ARG NO. _) must be less than the order (ARG NO. _) of the matrix.

(ARG NO. _) must be greater than zero.

Insufficient storage allocated for positive definite solve. (\_) additional bytes required.

The resulting correlation length obtained from ARG NO. 8 = (\_) and ARG NO. 10 = (\_) must be less than (\_).

(ARG NO. _) must be greater than or equal to zero.

(ARG NO. _) must be greater than (\_).

The number of initialized coefficients (ARG NO. _) cannot exceed the size of the coefficient vector (ARG NO. _).

The order specified (ARG NO. _) is not supported for this quadrature method. The nearest supported order is (\_).

The scaling parameter (ARG NO. _) must be greater than zero for this quadrature method.

The scaling parameter (ARG NO. _) must be nonzero for this quadrature method.

The sum of (ARG NO. _) and (ARG NO. _) must be nonzero for this quadrature method.

The number of data points (ARG NO. _) must be greater than one in order to perform numerical quadrature.

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 (\_).

The number of columns (ARG NO. _) specified for the matrix used to store the sparse matrix in compressed mode must be greater than zero.

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.

The number of non-zero elements in row (\_) obtained from the row pointer array (ARG NO. _) is less than zero.

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.

Element (\_) of the vector used to store the diagonal numbers (ARG NO. _) is incompatible with the order of the sparse matrix (ARG NO. _).

The matrix is singular because the number of non-zero entries (ARG NO. _) is zero.
Element (_) in the integer parameter vector (ARG NO. _) must be greater than or equal to zero.

Element (_) in the integer parameter vector (ARG NO. _) must be ( ), ( ), or ( ).

Element (_) in the real parameter vector (ARG NO. _) must be greater than zero.

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. _).

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 ( ).

The number of random numbers generated (ARG NO. _) must be even and greater than or equal to zero.

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'.

UPLO (ARG NO. _), which specifies whether an input matrix (ARG NO. _) is upper or lower triangular, must be 'U' or 'L'.

DIAG (ARG NO. _), which specifies whether an input matrix (ARG NO. _) is unit triangular, must be 'U' or 'N'.

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. _).

TRANS A (ARG NO. _) specifies whether an input matrix (ARG NO. _), its transpose, or its conjugate transpose should be used. TRANS A must be 'N', 'T', or 'C'.

The size of the leading dimension (ARG NO. _) of an array must be greater than or equal to zero.

Element (_) in (ARG NO. _) must be [one of the following: 0 or 1; greater than zero; greater than or equal to zero; greater than or equal to zero and less than or equal to 1; greater than the preceding element; greater than or equal to 1 and less than or equal to n; -1 or 1; nonzero; 0, 1, 2, 10, or 11; 0, 1, 2, 10, 11, 100, 102, or 110; 0; 1, 2, 10, 11, 100, 101, 102, 110, or 111; 1, 2, 3, or 4; 1, 2, 3, 4, or 5].

The number of eigenvalues (ARG NO. _) must be less than or equal to the order of the matrix (ARG NO. _).

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. _).

(ARG NO. _) must be a double precision whole number greater than or equal to 1.0 and less than 2147483647.0.

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 ( ).

The data type parameter (ARG NO. _) must be 'S', 'D', 'C', or 'Z'.

(ARG NO. _) must be greater than or equal to ( ) and smaller than ( ).

The matrix is singular. Column (_) is empty in the matrix specified by (ARG NO. _), (ARG NO. _), and (ARG NO. _).

The matrix is singular. Row (_) is empty in the matrix specified by (ARG NO. _), (ARG NO. _), and (ARG NO. _).

The matrix, specified by (ARG NO. _), (ARG NO. _), and (ARG NO. _), contains at least one duplicate column index in row ( ).
Element (_) in (ARG NO. _) must be [one of the following: greater than or equal to (_) and less than or equal to (_); greater than or equal to (_) and less than or equal to (ARG NO. _); greater than or equal to element (_) and less than or equal to (_); zero or must be greater than or equal to (_)].

Element (_) in (ARG NO. _) must be less than or equal to (_).

Element (_) in (ARG NO. _) may cause incorrect or misleading results. [One of the following: A nonzero number with absolute value less than or equal to 1; a positive number less than or equal to 1] is recommended.

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.

The dimension (ARG NO. _) of the array (ARG NO. _) must be greater than or equal to (_).

The number of steps after which the generalized minimum residual method is restarted, element (_) in (ARG NO. _), must be greater than 0.

The acceleration parameter, element (_) in (ARG NO. _), must be greater than 0 when using the SSOR preconditioner.

STOR (ARG NO. _), which specifies the storage variation used to represent the input sparse matrix, must be 'G', 'L', or 'U'.

INIT (ARG NO. _), which specifies the type of computation to be performed, must be 'T' or 'S'.

Element (_) in (ARG NO. _) must be [one of the following: greater than or equal to (_); greater than or equal to element (_)].

For level (_), the number of grid points for dimension (_) must be an odd number greater than 1.

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 (_).

The index into (ARG NO. _) is out of range. This index is element (_,_) of (ARG NO. _).

The index into (ARG NO. _) is out of range. This index is element (_,_,_) of (ARG NO. _).

For dimension (_) on level (_), the mesh spacing must be changed to a positive value.

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.

For level (_), the matrix type, solver, and preconditioner are incompatible.

The solver requested for level (_) requires a square matrix. Elements (_,_,_) and (_,_,_) in (ARG NO. _) must be equal.

Element (_,_) of (ARG NO. _) must be greater than or equal to (_).

End of input argument error reporting. For more information, refer to Engineering and Scientific Subroutine Library Guide and Reference.
### Computational Error Messages (2100-2199)

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2538-2100</td>
<td>The computed index of a vector is out of the range (<strong>) to (</strong>).</td>
</tr>
<tr>
<td>2538-2101</td>
<td>Eigenvalue (<strong>) failed to converge after (</strong>) iterations.</td>
</tr>
<tr>
<td>2538-2102</td>
<td>Eigenvector (<strong>) failed to converge after (</strong>) iterations.</td>
</tr>
<tr>
<td>2538-2103</td>
<td>The matrix (ARG NO. <strong>) is singular. Zero diagonal element (</strong>) has been detected.</td>
</tr>
<tr>
<td>2538-2104</td>
<td>The matrix (ARG NO. <strong>) is not positive definite. The last diagonal element with nonpositive value is (</strong>).</td>
</tr>
<tr>
<td>2538-2105</td>
<td>Factorization failed due to near zero pivot number (__).</td>
</tr>
<tr>
<td>2538-2106</td>
<td>Vector boundary misalignment detected in ESSL scalar library.</td>
</tr>
<tr>
<td>2538-2107</td>
<td>Singular value (<strong>) failed to converge after (</strong>) iterations.</td>
</tr>
<tr>
<td>2538-2108</td>
<td>The matrix specified by (ARG NO. __) and (ARG NO. __) is not definite because the diagonal is not of constant sign.</td>
</tr>
<tr>
<td>2538-2109</td>
<td>The matrix specified by (ARG NO. __) and (ARG NO. <strong>) is not definite and the iterative process is stopped at iteration number (</strong>).</td>
</tr>
<tr>
<td>2538-2110</td>
<td>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.</td>
</tr>
<tr>
<td>2538-2111</td>
<td>The factorization matrix (ARG NO. __) is not consistent with the sparse matrix specified by (ARG NO. __) and (ARG NO. __).</td>
</tr>
<tr>
<td>2538-2112</td>
<td>The incomplete factorization of the sparse matrix specified by (ARG NO. __) and (ARG NO. __) is not stable.</td>
</tr>
<tr>
<td>2538-2113</td>
<td>Unexpected nonzero vector mask detected in ESSL scalar routine. Contact your IBM Service Representative.</td>
</tr>
<tr>
<td>2538-2114</td>
<td>Eigenvalue (<strong>) failed to converge after (</strong>) iterations.</td>
</tr>
<tr>
<td>2538-2115</td>
<td>The matrix (ARG NO. <strong>) is not positive definite. The leading minor of order (</strong>) has a nonpositive determinant.</td>
</tr>
<tr>
<td>2538-2116</td>
<td>The matrix specified by (ARG NO. __) and (ARG NO. __) is singular.</td>
</tr>
<tr>
<td>2538-2117</td>
<td>The pivot element in column (__) is smaller than the first element in (ARG NO. __).</td>
</tr>
<tr>
<td>2538-2118</td>
<td>The pivot element in row (__) is smaller than the first element in (ARG NO. __).</td>
</tr>
<tr>
<td>2538-2119</td>
<td>The storage space, specified by (ARG NO. __), is insufficient.</td>
</tr>
<tr>
<td>2538-2120</td>
<td>The matrix is singular. The last row processed in the matrix was row (__).</td>
</tr>
<tr>
<td>2538-2121</td>
<td>The matrix is singular. the last column processed was column (__).</td>
</tr>
<tr>
<td>2538-2122</td>
<td>The factorization failed. No pivot element was found in the active submatrix.</td>
</tr>
<tr>
<td>2538-2123</td>
<td>Performance can be improved by specifying a larger value for (ARG NO. <strong>). (</strong>) compressions were performed.</td>
</tr>
<tr>
<td>2538-2124</td>
<td>The data contained in AUX1, (ARG NO. __), was computed for a different algorithm.</td>
</tr>
<tr>
<td>2538-2126</td>
<td>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.</td>
</tr>
</tbody>
</table>
The pivot value at row (_) was replaced with element (_) in (ARG NO. _). The matrix (ARG NO. _) may be singular or not definite.

Internal ESSL error. Contact your IBM service representative.

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.

The incomplete factorization of the sparse matrix specified by (ARG NO. _), (ARG NO. _), and (ARG NO. _) is not stable.

The matrix specified by (ARG NO. _), (ARG NO. _), and (ARG NO. _) is singular.

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.

An error occurred on level (_) in the user-supplied subroutine specified by (ARG NO. _).

The data contained in (ARG NO. _) is not consistent with the sparse matrix specified by (ARG NO. _), (ARG NO. _), and (ARG NO. _).

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.

For level (_), the main diagonal element for row (_) of a matrix is 0.

The input matrix (ARG NO. _) is singular. The first diagonal element found to be exactly zero was in column (_).

The input matrix (ARG NO. _) is singular. The first diagonal element found to be exactly zero was in column (_).

The matrix (ARG NO. _) is singular. Zero diagonal element (_) has been detected.

The matrix (ARG NO. _) is not positive definite. The leading minor of order (_) has a nonpositive determinant.

Factorization failed due to near zero pivot number (_).

The inverse of matrix (ARG NO. _) could not be computed. The first diagonal element of the factored matrix found to be exactly zero was in column (_).

The inverse of matrix (ARG NO. _) could not be computed. The first diagonal element of the factored matrix found to be exactly zero was in column (_).

The alignment of (ARG NO. _) changed after initialization. Performance may be significantly degraded.

Eigenvalue (_) failed to converge. Arrays WR (ARG NO. _) and WI (ARG NO. _) contain the eigenvalues successfully computed. For more information, refer to Engineering and Scientific Subroutine Library Guide and Reference.

Bisection failed to converge for some eigenvalues. The eigenvalues may not be as accurate as the absolute and relative tolerances.

The number of eigenvalues computed (ARG NO. _) does not match the number of eigenvalues requested.

No eigenvalues were computed since the Gershgorin interval initially used was incorrect.
(eigen) eigenvectors failed to converge after (iterations). The indices are stored in IFAIL (ARG NO. _).

Eigenvalue (_) failed to converge. Array W (ARG NO. _) contains the eigenvalues successfully computed. For more information, refer to Engineering and Scientific Subroutine Library Guide and Reference.

Eigenvalue (_) failed to converge in the QZ iteration. Arrays ALPHAR (ARG NO. _), ALPHAI (ARG NO. _) and BETA (ARG NO. _) contain the eigenvalues successfully computed. For more information, refer to Engineering and Scientific Subroutine Library Guide and Reference.

Eigenvalue (_) failed to converge in the computation of shifts. Arrays ALPHAR (ARG NO. _), ALPHAI (ARG NO. _) and BETA (ARG NO. _) contain the eigenvalues successfully computed. For more information, refer to Engineering and Scientific Subroutine Library Guide and Reference.

An eigenvector failed to converge because the 2-by-2 block (_ : _) did not have a complex eigenvalue.

The algorithm failed to converge because (_) off-diagonal elements of an intermediate tridiagonal form did not converge to zero.

An eigenvalue failed to converge in the submatrix starting at row and column (_) and ending at row and column (_).

Input-Argument Error Messages (2200-2299)

The dimension (ARG NO. _) of the array (ARG NO. _) must be greater than or equal to (_).

The number of elements in the array (ARG NO. _) must be less than or equal to (_).

The number of elements (ARG NO. _) in a work array (ARG NO. _) must be zero, to indicate dynamic allocation, minus one, to indicate workspace query, or greater than or equal to (_) if a work array is being supplied.

ANORM (ARG NO. _) must be equal to zero or greater than or equal to (_) and less than or equal to (_).

NORM (ARG NO. _), which specifies the computation to be performed, must be 'M', 'I', 'O', 'I', 'F', or 'E'.

The matrix specified by (ARG NO. _), (ARG NO. _), and (ARG NO. _) is singular. The first diagonal element found to be exactly zero was in column (_).

Singular value decomposition failed to converge.

End of computational error reporting. For more information, refer to Engineering and Scientific Subroutine Library Guide and Reference.
NORM (ARG NO. _), which specifies whether to calculate the 1-norm condition number or the infinity-norm condition number, must be '1', 'O', or 'I'.

The alignment of (ARG NO. _) changed after initialization.

JOBZ (ARG NO. _), which specifies whether or not to compute eigenvectors, must be 'N' or 'V'.

RANGE (ARG NO. _), which specifies which eigenvalues to find, must be 'A', 'V', or 'I'.

VU (ARG NO. _), which specifies the upper bound of the interval to be searched for eigenvalues, must be greater than VL (ARG NO. _), which specifies the lower bound of the interval to be searched for eigenvalues.

IL (ARG NO. _), which specifies the index of the smallest eigenvalue to be returned, must be greater than or equal to 1 and less than or equal to the larger of 1 and the order (ARG NO. _) of the matrix (ARG NO. _).

IU (ARG NO. _), which specifies the index of the largest eigenvalue to be returned, must be greater than or equal to the smaller of the order (ARG NO. _) of the matrix (ARG NO. _) and IL (ARG NO. _) and less than or equal to the order of the matrix.

BALANC (ARG NO. _), which specifies whether or not to diagonally scale the input matrix (ARG NO. _) and whether or not to permute the input matrix, must be 'N', 'P', 'S', or 'B'.

JOBVL (ARG NO. _) and JOBVR (ARG NO. _) must be 'V' if SENSE (ARG NO. _) is 'E' or 'B'.

ITYPE (ARG NO. _), which specifies the problem type, must be 1, 2, or 3.

The routine must be initialized with the present value of element (_) of (ARG NO. _).

UPLO (ARG NO. _), which specifies whether off-diagonal E (ARG NO. _) is the superdiagonal or the subdiagonal of the bidiagonal factorization, must be 'U' or 'L'.

The lower bandwidth (ARG NO. _) must be less than the number of rows (ARG NO. _) of the matrix.

The upper bandwidth (ARG NO. _) must be less than the number of columns (ARG NO. _) of the matrix.

JOBU (ARG NO. _), which specifies whether or not to compute left singular vectors, must be 'N', 'A', 'S', or 'O'.

JOBVT (ARG NO. _), which specifies whether or not to compute left singular vectors, must be 'N', 'A', 'S', or 'O'.

JOBU (ARG NO. _) and JOBVT (ARG NO. _) cannot both be 'O'.

The size of the leading dimension (ARG NO. _) of an array must be greater than or equal to the smaller of (ARG NO. _) and (ARG NO. _).

IOPT (ARG NO. _) must be 1 or 2.

IREPEAT (ARG NO. _) must be 0 or 1.
LISEED (ARG NO. _), which depends on IOPT (ARG NO. _), must be greater than or equal to (_).

LISTATE (ARG NO. _), which depends on IOPT (ARG NO. _), must be minus one to indicate an ISTATE (ARG NO. _), size query, or greater than or equal to (_) if the state vector has been supplied.

ISTATE (ARG NO. _) is not initialized.

(ARG NO. _) must be less than (ARG NO. _).

ISTATE (ARG NO. _) must be initialized with IOPT equal to (_).

ESSL_CUDA_HYBRID must be "yes", "no", or unset.

ESSL_CUDA_PIN must be "yes", "no", "pinned", or unset.

Element (_) of array IDS (ARG NO. _) must be greater than or equal to zero or less than the number of CUDA devices (_).

This subroutine may be called only once, and it must be called before any ESSL GPU enabled subroutines.

The CUDA device corresponding to element (_) in array IDS (ARG NO. _) must be in NVIDIA compute mode 0 (DEFAULT) or 3 (EXCLUSIVE_PROCESS).

cblas_layout (ARG NO. _), which specifies whether matrices are stored in row major or column major order, must be CblasRowMajor or CblasColMajor.

The form (ARG NO. _) of a matrix must be CblasNoTrans or CblasConjTrans.

cblas_diag (ARG NO. _), which specifies whether an input matrix (ARG NO. _) is unit triangular, must be CblasUnit or CblasNonUnit.

cblas_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 CblasLeft or CblasRight.

cblas_uplo (ARG NO. _), which specifies whether an input matrix (ARG NO. _) is upper or lower triangular, must be CblasUpper or CblasLower.

matrix_layout (ARG NO. _), which specifies whether matrices are stored in row major or column major order, must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

JOBZ (ARG NO. _), which specifies options for computing all or part of the matrix U, must be 'N', 'A', 'S', or 'O'.

The size of a work array (ARG NO. _), specified by (ARG NO. _) or implicitly computed by ESSL, must be greater than zero and less than or equal to 2147483647 when 32-bit integers are used.

BALANC (ARG NO. _), which specifies whether or not to diagonally scale the input matrices (ARG NO. _ and ARG NO. _) and whether or not to permute the input matrices, must be 'N', 'P', 'S', or 'O'.

The size of the leading dimension (ARG NO. _) of an array must be greater than or equal to (_), computed as (ARG NO. _) minus (ARG NO. _) plus 1.

The number of columns (ARG NO. _) of a matrix must be less than or equal to the size of the leading dimension (ARG NO. _) of its array.
Resource Error Messages(2400-2499)

2538-2400  An internal buffer allocation has failed due to insufficient memory.

Informational and Attention Error Messages(2600-2699)

2538-2600  Performance may be degraded due to limited buffer space availability.

2538-2601  Execution terminating due to error count for error number ( ). Message summary: Message number - Count


2538-2603  Standard corrective action taken. Execution continuing.

2538-2604  Execution terminating due to error count for error number ( ).

2538-2605  Message summary: _

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. _).

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. _).

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 ( ).

Miscellaneous Error Messages(2700-2799)

2538-2700  Internal ESSL error number ( ). Contact your IBM service representative.

2538-2703  Internal ESSL error: message number requested ( ) is outside of the valid range. Contact your IBM service representative.

2538-2799  Unable to locate message number ( ). Please refer to 'Using Error Handling' in the ESSL Guide and Reference for the full message text.
Part 2. Reference Information

This documentation 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
Chapter 8. Linear Algebra Subprograms

The linear algebra subprograms, provided in four areas, are described here.

Overview of the Linear Algebra Subprograms

This describes the subprograms in each of the four linear algebra subprogram areas:

- Vector-scalar linear algebra subprograms ("Vector-Scalar Linear Algebra Subprograms")
- Sparse vector-scalar linear algebra subprograms ("Sparse Vector-Scalar Linear Algebra Subprograms" on page 227)
- Matrix-vector linear algebra subprograms ("Matrix-Vector Linear Algebra Subprograms" on page 228)
- Sparse matrix-vector linear algebra subprograms ("Sparse Matrix-Vector Linear Algebra Subprograms" on page 229)

Note:
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 [93 on page 1368]. 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 68. List of Vector-Scalar Linear Algebra Subprograms

<table>
<thead>
<tr>
<th>Short-Precision Subprogram</th>
<th>Long-Precision Subprogram</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISAMAX“iclas_isamax’</td>
<td>IDAMAX“cblas_idamax’</td>
<td>“ISAMAX, IDAMAX, ICAMAX, and IZAMAX (Position of the First or Last Occurrence of the Vector Element Having the Largest Magnitude)” on page 233</td>
</tr>
<tr>
<td>ICAMAX“cblas_icamax’</td>
<td>IZAMAX“cblas_izamax’</td>
<td></td>
</tr>
<tr>
<td>cblas_isamax’</td>
<td>cblas_idamax’</td>
<td></td>
</tr>
<tr>
<td>cblas_icamax’</td>
<td>cblas_izamax’</td>
<td></td>
</tr>
<tr>
<td>ISAMIN“iclas_isamin’</td>
<td>IDAMIN“cblas_idamin’</td>
<td>“ISAMIN and IDAMIN (Position of the First or Last Occurrence of the Vector Element Having Minimum Absolute Value)” on page 236</td>
</tr>
<tr>
<td>ISMAX“cblas_ismax’</td>
<td>IDMAX“cblas_idmax’</td>
<td>“ISMAX and IDMAX (Position of the First or Last Occurrence of the Vector Element Having the Maximum Value)” on page 239</td>
</tr>
<tr>
<td>ISMIN“cblas_ismin’</td>
<td>IDMIN“cblas_idmin’</td>
<td>“ISMIN and IDMIN (Position of the First or Last Occurrence of the Vector Element Having Minimum Value)” on page 242</td>
</tr>
<tr>
<td>SCASUM“cblas_scasum’</td>
<td>DASUM“cblas_dasum’</td>
<td>“SASUM, DASUM, SCASUM, and DZASUM (Sum of the Magnitudes of the Elements in a Vector)” on page 245</td>
</tr>
<tr>
<td>SCASUM“cblas_scasum’</td>
<td>DZASUM“cblas_dzasum’</td>
<td></td>
</tr>
<tr>
<td>Short-Precision Subprogram</td>
<td>Long-Precision Subprogram</td>
<td>Descriptive Name and Location</td>
</tr>
<tr>
<td>----------------------------</td>
<td>---------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>SAXPY</td>
<td>DAXPY</td>
<td>“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 248</td>
</tr>
<tr>
<td>CAXPY</td>
<td>ZAXPY</td>
<td></td>
</tr>
<tr>
<td>cblas_saxpy</td>
<td>cblas_daxpy</td>
<td></td>
</tr>
<tr>
<td>cblas_caxpy</td>
<td>cblas_zaxpy</td>
<td></td>
</tr>
<tr>
<td>SCOPY</td>
<td>DCOPY</td>
<td>“SCOPY, DCOPY, CCOPY, and ZCOPY (Copy a Vector)” on page 251</td>
</tr>
<tr>
<td>CCOPY</td>
<td>ZCOPY</td>
<td></td>
</tr>
<tr>
<td>cblas_scopy</td>
<td>cblas_dcopy</td>
<td></td>
</tr>
<tr>
<td>cblas_ccopy</td>
<td>cblas_zcopy</td>
<td></td>
</tr>
<tr>
<td>SDOT</td>
<td>DDOT</td>
<td>“SDOT, DDOT, CDOTU, ZDOTU, CDOTC, and ZDOTC (Dot Product of Two Vectors)” on page 254</td>
</tr>
<tr>
<td>CDOTU</td>
<td>ZDOTU</td>
<td></td>
</tr>
<tr>
<td>CDOTC</td>
<td>ZDOTC</td>
<td></td>
</tr>
<tr>
<td>cblas_sdot</td>
<td>cblas_ddot</td>
<td></td>
</tr>
<tr>
<td>cblas_cdotu_sub</td>
<td>cblas_zdotu_sub</td>
<td></td>
</tr>
<tr>
<td>cblas_cdotc_sub</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAXPY</td>
<td>DNAXPY</td>
<td>“SNAXPY and DNAXPY (Compute SAXPY or DAXPY N Times)” on page 258</td>
</tr>
<tr>
<td>SNDOT</td>
<td>DNDOT</td>
<td>“SNDOT and DNDOT (Compute Special Dot Products N Times)” on page 263</td>
</tr>
<tr>
<td>SNRM2</td>
<td>DNRM2</td>
<td>“SNRM2, DNRM2, SCNRM2, and DZNRM2 (Euclidean Length of a Vector with Scaling of Input to Avoid Destructive Underflow and Overflow)” on page 268</td>
</tr>
<tr>
<td>SCNRM2</td>
<td>ZDNRM2</td>
<td></td>
</tr>
<tr>
<td>cblas_snrm2</td>
<td>cblas_znrm2</td>
<td></td>
</tr>
<tr>
<td>CNORM2</td>
<td>ZNORM2</td>
<td>“SNORM2, DNORM2, CNORM2, and ZNORM2 (Euclidean Length of a Vector with No Scaling of Input)” on page 271</td>
</tr>
<tr>
<td>SROTG</td>
<td>DROTG</td>
<td>“SROTG, DROTG, CROTG, and ZROTG (Construct a Givens Plane Rotation)” on page 274</td>
</tr>
<tr>
<td>CROTG</td>
<td>ZROTG</td>
<td></td>
</tr>
<tr>
<td>CSROT</td>
<td>ZDROTG</td>
<td></td>
</tr>
<tr>
<td>cblas_srot</td>
<td>cblas_drot</td>
<td></td>
</tr>
<tr>
<td>cblas_crot</td>
<td>cblas_zrot</td>
<td></td>
</tr>
<tr>
<td>cblas_csrrot</td>
<td>cblas_zcsrrot</td>
<td></td>
</tr>
<tr>
<td>SROTMG</td>
<td>DROTMG</td>
<td>“SROTMG and DROTMG (Construct a modified Givens Transformation)” on page 284</td>
</tr>
<tr>
<td>cblas_srotmg</td>
<td>cblas_drotmg</td>
<td></td>
</tr>
<tr>
<td>SROTM</td>
<td>DROTM</td>
<td>“SROTM and DROTM (Apply a modified Givens Transformation)” on page 287</td>
</tr>
<tr>
<td>cblas_srotm</td>
<td>cblas_drotm</td>
<td></td>
</tr>
<tr>
<td>SSCAL</td>
<td>DSCAL</td>
<td>“SSCAL, DSCAL, CSCAL, ZSCAL, CSSCAL, and ZDSCAL (Multiply a Vector X by a Scalar and Store in the Vector X)” on page 290</td>
</tr>
<tr>
<td>CSCAL</td>
<td>ZSCAL</td>
<td></td>
</tr>
<tr>
<td>CSSCAL</td>
<td>ZDSCAL</td>
<td></td>
</tr>
<tr>
<td>cblas_sscal</td>
<td>cblas_dscal</td>
<td></td>
</tr>
<tr>
<td>cblas_cscal</td>
<td>cblas_zscal</td>
<td></td>
</tr>
<tr>
<td>cblas_csscal</td>
<td>cblas_zdscal</td>
<td></td>
</tr>
<tr>
<td>SSWAP</td>
<td>DSWAP</td>
<td>“SSWAP, DSWAP, CSWAP, and ZSWAP (Interchange the Elements of Two Vectors)” on page 293</td>
</tr>
<tr>
<td>CSWAP</td>
<td>ZSWAP</td>
<td></td>
</tr>
<tr>
<td>cblas_sswap</td>
<td>cblas_dswap</td>
<td></td>
</tr>
<tr>
<td>cblas_cswap</td>
<td>cblas_zswap</td>
<td></td>
</tr>
</tbody>
</table>

ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
### Table 68. List of Vector-Scalar Linear Algebra Subprograms (continued)

<table>
<thead>
<tr>
<th>Short-Precision Subprogram</th>
<th>Long-Precision Subprogram</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVEA</td>
<td>DVEA</td>
<td>“SVEA, DVEA, CVEA, and ZVEA (Add a Vector X to a Vector Y and Store in a Vector Z)” on page 296</td>
</tr>
<tr>
<td>CVEA</td>
<td>ZVEA</td>
<td></td>
</tr>
<tr>
<td>cblas_svea</td>
<td>cblas_dvea</td>
<td></td>
</tr>
<tr>
<td>cblas_zvea</td>
<td>cblas_zvea</td>
<td></td>
</tr>
<tr>
<td>SVES</td>
<td>DVES</td>
<td>“SVES, DVES, CVEA, and ZVES (Subtract a Vector Y from a Vector X and Store in a Vector Z)” on page 300</td>
</tr>
<tr>
<td>CVEA</td>
<td>ZVES</td>
<td></td>
</tr>
<tr>
<td>cblas_sves</td>
<td>cblas_dves</td>
<td></td>
</tr>
<tr>
<td>cblas_zves</td>
<td>cblas_zves</td>
<td></td>
</tr>
<tr>
<td>SVES</td>
<td>DVEM</td>
<td>“SVES, DVEA, CVEA, and ZVEA (Multiply a Vector X by a Vector Y and Store in a Vector Z)” on page 304</td>
</tr>
<tr>
<td>CVEA</td>
<td>ZVEM</td>
<td></td>
</tr>
<tr>
<td>cblas_svem</td>
<td>cblas_dvem</td>
<td></td>
</tr>
<tr>
<td>cblas_zvem</td>
<td>cblas_zvem</td>
<td></td>
</tr>
<tr>
<td>SYAX</td>
<td>DYAX</td>
<td>“SYAX, DYAX, CYAX, ZYAX, CSYAX, and ZDYAX (Multiply a Vector X by a Scalar and Store in a Vector Y)” on page 308</td>
</tr>
<tr>
<td>CYAX</td>
<td>ZYAX</td>
<td></td>
</tr>
<tr>
<td>CSYAX</td>
<td>ZDYAX</td>
<td></td>
</tr>
<tr>
<td>cblas_syax</td>
<td>cblas_dyax</td>
<td></td>
</tr>
<tr>
<td>cblas_cyax</td>
<td>cblas_zyax</td>
<td></td>
</tr>
<tr>
<td>cblas_csyax</td>
<td>cblas_zdyax</td>
<td></td>
</tr>
<tr>
<td>DZAXPY</td>
<td>ZZAXPY</td>
<td>“SZAXPY, DZAXPY, CZAXPY, and ZZAXPY (Multiply a Vector X by a Scalar, Add to a Sparse Y, and Store in a Vector Z)” on page 312</td>
</tr>
<tr>
<td>CZAXPY</td>
<td>ZZAXPY</td>
<td></td>
</tr>
<tr>
<td>cblas_szaxpy</td>
<td>cblas_dzaxpy</td>
<td></td>
</tr>
<tr>
<td>cblas_czaxpy</td>
<td>cblas_zzaxpy</td>
<td></td>
</tr>
</tbody>
</table>

1 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 [37 on page 1365](#). Both real and complex versions of the subprograms are provided.

### Table 69. List of Sparse Vector-Scalar Linear Algebra Subprograms

<table>
<thead>
<tr>
<th>Short-Precision Subprogram</th>
<th>Long-Precision Subprogram</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSCTR</td>
<td>DSCTR</td>
<td>“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)” on page 317</td>
</tr>
<tr>
<td>CSCTR</td>
<td>ZSCTR</td>
<td></td>
</tr>
<tr>
<td>SGTHR</td>
<td>DGTHR</td>
<td>“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)” on page 320</td>
</tr>
<tr>
<td>CGTHR</td>
<td>ZGTHR</td>
<td></td>
</tr>
<tr>
<td>SGTHRZ</td>
<td>DGTHRZ</td>
<td>“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)” on page 323</td>
</tr>
<tr>
<td>CGTHRZ</td>
<td>ZGTHRZ</td>
<td></td>
</tr>
<tr>
<td>SAXPYI</td>
<td>DAXPYI</td>
<td>“SAXPYI, DAXPYI, CAXPYI, and ZAXPYI (Multiply a Sparse Vector X in Compressed-Vector Storage Mode by a Scalar, Add to a Sparse Y in Full-Vector Storage Mode, and Store in the Vector Y)” on page 326</td>
</tr>
<tr>
<td>CAXPYI</td>
<td>ZAXPYI</td>
<td></td>
</tr>
</tbody>
</table>
Table 69. List of Sparse Vector-Scalar Linear Algebra Subprograms (continued)

<table>
<thead>
<tr>
<th>Short-Precision Subprogram</th>
<th>Long-Precision Subprogram</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDOTI†</td>
<td>DDOTI†</td>
<td>“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)” on page 329</td>
</tr>
<tr>
<td>CDOTCI†</td>
<td>ZDOTCI†</td>
<td></td>
</tr>
<tr>
<td>CDOTUI†</td>
<td>ZDOTUI†</td>
<td></td>
</tr>
</tbody>
</table>

† 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 [42 on page 1365] and [43 on page 1365]. Both real and complex versions of the subprograms are provided.

Table 70. List of Matrix-Vector Linear Algebra Subprograms

<table>
<thead>
<tr>
<th>Short-Precision Subprogram</th>
<th>Long-Precision Subprogram</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGEMV*</td>
<td>DGEMV*</td>
<td>“SGEMV, DGEMV, CGEMV, ZGEMV, SGEMX, DGEMX, SGEMTX, and DGEMTX (Matrix-Vector Product for a General Matrix, Its Transpose, or Its Conjugate Transpose)” on page 334</td>
</tr>
<tr>
<td>CGEMV*</td>
<td>ZGEMV*</td>
<td></td>
</tr>
<tr>
<td>SGEMX§</td>
<td>DGEMX§</td>
<td></td>
</tr>
<tr>
<td>SGEMTX§</td>
<td>DGEMTX§</td>
<td></td>
</tr>
<tr>
<td>cblas_sgemv*</td>
<td>cblas_dgemv*</td>
<td></td>
</tr>
<tr>
<td>cblas_cgemv*</td>
<td>cblas_zgemv*</td>
<td></td>
</tr>
<tr>
<td>SGER*</td>
<td>DGER*</td>
<td>“SGER, DGER, CGERU, ZGERU, CGERC, and ZGERC (Rank-One Update of a General Matrix)” on page 345</td>
</tr>
<tr>
<td>CGERU*</td>
<td>ZGERU*</td>
<td></td>
</tr>
<tr>
<td>CGERC*</td>
<td>ZGERC*</td>
<td></td>
</tr>
<tr>
<td>cblas_sger*</td>
<td>cblas_dger*</td>
<td></td>
</tr>
<tr>
<td>cblas_cgeru*</td>
<td>cblas_zgeru*</td>
<td></td>
</tr>
<tr>
<td>cblas_cgerc*</td>
<td>cblas_zgerc*</td>
<td></td>
</tr>
<tr>
<td>SSPMVM*</td>
<td>DSPMVM*</td>
<td>“SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV, SSLMX, and DSLMX (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Matrix)” on page 353</td>
</tr>
<tr>
<td>CHPMV*</td>
<td>ZHPMV*</td>
<td></td>
</tr>
<tr>
<td>SSYMV*</td>
<td>DSYMV*</td>
<td></td>
</tr>
<tr>
<td>CHEMV*</td>
<td>ZHEMV*</td>
<td></td>
</tr>
<tr>
<td>SSLMX§</td>
<td>DSSMX§</td>
<td></td>
</tr>
<tr>
<td>cblas_sspmv*</td>
<td>cblas_dspmv*</td>
<td></td>
</tr>
<tr>
<td>cblas_chpmv*</td>
<td>cblas_zhpvm*</td>
<td></td>
</tr>
<tr>
<td>cblas_ssymv*</td>
<td>cblas_dsymv*</td>
<td></td>
</tr>
<tr>
<td>cblas_chemv*</td>
<td>cblas_zchemv*</td>
<td></td>
</tr>
<tr>
<td>SSPR*</td>
<td>DSPR*</td>
<td>“SSPR, DSPR, CHPR, ZHPR, SSYR, DSYR, CHER, ZHER, SSLR1, and DSLR1 (Rank-One Update of a Real Symmetric or Complex Hermitian Matrix)” on page 362</td>
</tr>
<tr>
<td>CHPR*</td>
<td>ZHPR*</td>
<td></td>
</tr>
<tr>
<td>SSYR*</td>
<td>DSYR*</td>
<td></td>
</tr>
<tr>
<td>CHER*</td>
<td>ZHER*</td>
<td></td>
</tr>
<tr>
<td>SSLR1§</td>
<td>DSSLR1§</td>
<td></td>
</tr>
<tr>
<td>cblas_sspr*</td>
<td>cblas_dspr*</td>
<td></td>
</tr>
<tr>
<td>cblas_chpr*</td>
<td>cblas_zhpr*</td>
<td></td>
</tr>
<tr>
<td>cblas_ssymr*</td>
<td>cblas_dsymr*</td>
<td></td>
</tr>
<tr>
<td>cblas_cher*</td>
<td>cblas_zher*</td>
<td></td>
</tr>
</tbody>
</table>
Table 70. List of Matrix-Vector Linear Algebra Subprograms (continued)

<table>
<thead>
<tr>
<th>Short-Precision Subprogram</th>
<th>Long-Precision Subprogram</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sspr2*</td>
<td>Dspr2*</td>
<td>“SSPR2, Dspr2, Chpr2, ZHpr2, SSyr2, Dsyr2, Cher2, Zher2, SSlr2, and DSlr2 (Rank-Two Update of a Real Symmetric or Complex Hermitian Matrix)” on page 370</td>
</tr>
<tr>
<td>Chpr2*</td>
<td>Zhpr2*</td>
<td></td>
</tr>
<tr>
<td>Ssy2*</td>
<td>Dsyr2*</td>
<td></td>
</tr>
<tr>
<td>Cher2*</td>
<td>Zher2*</td>
<td></td>
</tr>
<tr>
<td>Sslr2§</td>
<td>Dslr2§</td>
<td></td>
</tr>
<tr>
<td>Cblas_sspr2*</td>
<td>Cblas_dspr2*</td>
<td></td>
</tr>
<tr>
<td>Cblas_chpr2*</td>
<td>Cblas_zhpr2*</td>
<td></td>
</tr>
<tr>
<td>Cblas_ssy2r*</td>
<td>Cblas_dsy2r*</td>
<td></td>
</tr>
<tr>
<td>Cblas_cher2*</td>
<td>Cblas_zher2*</td>
<td></td>
</tr>
<tr>
<td>Sgbmv*</td>
<td>Dgbmv*</td>
<td>“SGBMV, DGBMV, CGBMV, and ZGBMV (Matrix-Vector Product for a General Band Matrix, Its Transpose, or Its Conjugate Transpose)” on page 379</td>
</tr>
<tr>
<td>Cgmv*</td>
<td>Zgbmv*</td>
<td></td>
</tr>
<tr>
<td>Cblas_sgbmv*</td>
<td>Cblas_dgbmv*</td>
<td></td>
</tr>
<tr>
<td>Cblas_cgbmv*</td>
<td>Cblas_zgbmv*</td>
<td></td>
</tr>
<tr>
<td>Ssbmv*</td>
<td>Dsbmv*</td>
<td>“SSBMV, DSBMV, CHBMV, and ZHBMV (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Band Matrix)” on page 386</td>
</tr>
<tr>
<td>Chbmv*</td>
<td>Zhbmv*</td>
<td></td>
</tr>
<tr>
<td>Cblas_ssbmv*</td>
<td>Cblas_dsbmv*</td>
<td></td>
</tr>
<tr>
<td>Cblas_chbmv*</td>
<td>Cblas_zhbmv*</td>
<td></td>
</tr>
<tr>
<td>Strmv*</td>
<td>Dtrmv*</td>
<td>“STRMV, DTRMV, CTRMV, ZTRMV, STPMV, DTPMV, CTPMV, and ZTPMV (Matrix-Vector Product for a Triangular Matrix, Its Transpose, or Its Conjugate Transpose)” on page 391</td>
</tr>
<tr>
<td>Ctrmv*</td>
<td>Ztrmv*</td>
<td></td>
</tr>
<tr>
<td>Stpmv*</td>
<td>Dtpmv*</td>
<td></td>
</tr>
<tr>
<td>Ctpmv*</td>
<td>Ztpmv*</td>
<td></td>
</tr>
<tr>
<td>Cblas_strmv*</td>
<td>Cblas_dtrmv*</td>
<td></td>
</tr>
<tr>
<td>Cblas_ctrmv*</td>
<td>Cblas_ztrmv*</td>
<td></td>
</tr>
<tr>
<td>Cblas_stpmv*</td>
<td>Cblas_dtpmv*</td>
<td></td>
</tr>
<tr>
<td>Cblas_ctpmv*</td>
<td>Cblas_ztpmv*</td>
<td></td>
</tr>
<tr>
<td>Strsv*</td>
<td>Dtrsv*</td>
<td>“STRSV, DTRS, CTRSV, ZTRSV, STPSV, DTPSV, CTPSV, and ZTPSV (Solution of a Triangular System of Equations with a Single Right-Hand Side)” on page 398</td>
</tr>
<tr>
<td>Ctrsv*</td>
<td>Ztrsv*</td>
<td></td>
</tr>
<tr>
<td>Stpsv*</td>
<td>Dtpsv*</td>
<td></td>
</tr>
<tr>
<td>Ctpsv*</td>
<td>Ztpsv*</td>
<td></td>
</tr>
<tr>
<td>Cblas_strsv*</td>
<td>Cblas_dtrsv*</td>
<td></td>
</tr>
<tr>
<td>Cblas_cstrsv*</td>
<td>Cblas_ztrsv*</td>
<td></td>
</tr>
<tr>
<td>Cblas_stpsv*</td>
<td>Cblas_dtpsv*</td>
<td></td>
</tr>
<tr>
<td>Cblas_ctpsv*</td>
<td>Cblas_ztpsv*</td>
<td></td>
</tr>
<tr>
<td>Stbmv*</td>
<td>Dtbmv*</td>
<td>“STBMV, DTBMV, CTBMV, and ZTBMV (Matrix-Vector Product for a Triangular Band Matrix, Its Transpose, or Its Conjugate Transpose)” on page 405</td>
</tr>
<tr>
<td>Ctbmv*</td>
<td>Ztbmv*</td>
<td></td>
</tr>
<tr>
<td>Cblas_stbmv*</td>
<td>Cblas_dtbmv*</td>
<td></td>
</tr>
<tr>
<td>Cblas_ctbmv*</td>
<td>Cblas_ztbmv*</td>
<td></td>
</tr>
<tr>
<td>Stbsv*</td>
<td>Dtbsv*</td>
<td>“STBSV, DTBSV, CTBSV, and ZTBSV (Triangular Band Equation Solve)” on page 411</td>
</tr>
<tr>
<td>Ctbsv*</td>
<td>Ztbsv*</td>
<td></td>
</tr>
<tr>
<td>Cblas_stbsv*</td>
<td>Cblas_dtbmv*</td>
<td></td>
</tr>
<tr>
<td>Cblas_ctbsv*</td>
<td>Cblas_ztbmv*</td>
<td></td>
</tr>
</tbody>
</table>

* 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 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 71. List of Sparse Matrix-Vector Linear Algebra Subprograms

<table>
<thead>
<tr>
<th>Long-Precision Subprogram</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSMMX</td>
<td>“DSMMX (Matrix-Vector Product for a Sparse Matrix in Compressed-Matrix Storage Mode)” on page 418</td>
</tr>
<tr>
<td>DSMTM</td>
<td>“DSMTM (Transpose a Sparse Matrix in Compressed-Matrix Storage Mode)” on page 421</td>
</tr>
<tr>
<td>DSDMX</td>
<td>“DSDMX (Matrix-Vector Product for a Sparse Matrix or Its Transpose in Compressed-Diagonal Storage Mode)” on page 425</td>
</tr>
</tbody>
</table>

Use Considerations

If your program uses a sparse matrix stored by rows, as defined in “Storage-by-Rows” on page 122, you should first convert your sparse matrix to compressed-matrix storage mode by using the subroutine DSRSM (see “DSRSM (Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode)” on page 132). 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 α = 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. However, when short-precision subroutines use the Altivec or VSX unit to improve performance, they do not accumulate intermediate 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 116.

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 64.
Vector-Scalar Subprograms

This 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)

Purpose

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 72. Data Types

<table>
<thead>
<tr>
<th>( x )</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>ISAMAX</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>IDAMAX</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>ICAMAX</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>IZAMAX</td>
</tr>
</tbody>
</table>

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

Fortran

\[
\text{ISAMAX | IDAMAX | ICAMAX | IZAMAX (} n, x, incx \text{)}
\]

C and C++

isamax | idamax | icamax | izamax (n, x, incx);

CBLAS

cblas_isamax | cblas_idamax | cblas_icamax | cblas_izamax (n, x, incx);

On Entry

\( n \) is the number of elements in vector \( x \). Specified as: an integer; \( n \geq 0 \).

\( x \) is the vector \( x \) of length \( n \). Specified as: a one-dimensional array of (at least) length \( 1+(n-1) | \text{incx} | \), containing numbers of the data type indicated in Table 72.

\( \text{incx} \)

is the stride for vector \( x \).

Specified as: an integer. It can have any value.

On Return

Function value

is the position \( i \) of the element in the array, where:

If \( \text{incx} \geq 0 \), \( i \) is the position of the first occurrence.

If \( \text{incx} < 0 \), \( i \) is the position of the last occurrence.

Returned as:

* an integer; \( 0 \leq i \leq n \) (for Fortran, C, and C++)
• a CBLAS_INDEX; 0 ≤ i ≤ n-1

Notes

Declare the ISAMAX, IDAMAX, ICAMAX, and IZAMAX functions in your program as returning an integer value.

Function

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 \( \text{incx} \), \( i \) has the following values:

For \( \text{incx} ≥ 0 \), \( i = k \)
For \( \text{incx} < 0 \), \( i = n-k+1 \)

See reference [93 on page 1368]. 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
\( n < 0 \)

Examples

Example 1
This example shows a vector, \( x \), with a stride of 1.

Function Reference and Input:

\[
\begin{array}{c|c|c}
N & X & \text{INCX} \\
\hline
9 & X & 1 \\
\end{array}
\]

\( \text{IMAX} = \text{ISAMAX}( 9, X, 1 ) \)

\( x = (1.0, 2.0, 7.0, -8.0, -5.0, -10.0, -9.0, 10.0, 6.0) \)

Output:

\( \text{IMAX} = 6 \)

Example 2
This example shows a vector, \( x \), with a stride greater than 1.

Function Reference and Input:

\[
\begin{array}{c|c|c}
N & X & \text{INCX} \\
\hline
5 & X & 2 \\
\end{array}
\]

\( \text{IMAX} = \text{ISAMAX}( 5, X, 2 ) \)

\( x = (1.0, . , 7.0, . , -5.0, . , -9.0, . , 6.0) \)

Output:
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)

Purpose

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 73. Data Types

<table>
<thead>
<tr>
<th>$x$</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>ISAMIN</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>IDAMIN</td>
</tr>
</tbody>
</table>

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>ISAMIN</th>
<th>IDAMIN $(n, x, incx)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>isamin</td>
<td>idamin $(n, x, incx)$;</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_isamin</td>
<td>cblas_idamin $(n, x, incx)$;</td>
</tr>
</tbody>
</table>

On Entry

$n$ is the number of elements in vector $x$. Specified as: an 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 73.

$incx$

is the stride for vector $x$.

Specified as: an 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:

- an integer; $0 \leq i \leq n$ (for Fortran, C, and C++)
- a CBLAS_INDEX; $0 \leq i \leq n-1$
Notes

Declare the ISAMIN and IDAMIN functions in your program as returning an 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 [93 on page 1368]. 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
\( n < 0 \)

Examples

Example 1
This example shows a vector, \( x \), with a stride of 1.

Function Reference and Input:

\[
| \begin{array}{ccc}N & X & INCX \\
| & | & | \\
| IMIN = ISAMIN( 6, X, 1 ) \\
X = (3.0, 4.0, 1.0, 8.0, 1.0, 3.0) \\
| Output \\
IMIN = 3
\end{array}
\]

Example 2
This example shows a vector, \( x \), with a stride greater than 1.

Function Reference and Input:

\[
| \begin{array}{ccc}N & X & INCX \\
| & | & | \\
| IMIN = ISAMIN( 4, X, 2 ) \\
X = (-3.0, . , -9.0, . , -8.0, . , 3.0) \\
| Output: \\
IMIN = 1
\end{array}
\]
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:

\[
\begin{array}{ccc}
N & X & INCX \\
\mid & \mid & \mid \\
IMIN = ISAMIN(4, X, 2) \\
\end{array}
\]

\[
x = (2.0, . , -1.0, . , 4.0, . , 1.0)
\]

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:

\[
\begin{array}{ccc}
N & X & INCX \\
\mid & \mid & \mid \\
IMIN = ISAMIN(4, X, -2) \\
\end{array}
\]

\[
x = (2.0, . , -1.0, . , 4.0, . , 1.0)
\]

Output:

\[
IMIN = 4
\]
ISMAX and IDMAX (Position of the First or Last Occurrence of the Vector Element Having the Maximum Value)

**Purpose**

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 74. Data Types**

<table>
<thead>
<tr>
<th>( x )</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>ISMAX</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>IDMAX</td>
</tr>
</tbody>
</table>

**Note:** On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

**Syntax**

<table>
<thead>
<tr>
<th></th>
<th>Fortran</th>
<th>C and C++</th>
<th>CBLAS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ISMAX</td>
<td>IDMAX (n, x, incx)</td>
<td>ismax</td>
</tr>
</tbody>
</table>

**On Entry**

- \( n \) is the number of elements in vector \( x \). Specified as: an integer; \( n \geq 0 \).
- \( x \) is the vector \( x \) of length \( n \). Specified as: a one-dimensional array of (at least) length \( 1+(n-1) \mid \text{incx} \mid \), containing numbers of the data type indicated in Table 74.
- \( \text{incx} \) is the stride for vector \( x \). Specified as: an integer. It can have any value.

**On Return**

**Function value**

- is the position \( i \) of the element in the array, where:
  - If \( \text{incx} \geq 0 \), \( i \) is the position of the first occurrence.
  - If \( \text{incx} < 0 \), \( i \) is the position of the last occurrence.
- Returned as:
  - an integer; \( 0 \leq i \leq n \) (for Fortran, C, and C++)
  - a CBLAS_INDEX; \( 0 \leq i \leq n-1 \)
Notes

Declare the ISMAX and IDMAX functions in your program as returning an 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 [93 on page 1368]. 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

$n < 0$

Examples

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:

<table>
<thead>
<tr>
<th>$N$</th>
<th>$X$</th>
<th>INCX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$\text{IMAX} = \text{ISMAX}(\ 4\ ,\ X\ ,\ 2\ )$

$X = (2.0, . , 4.0, . , 4.0, . , 1.0)$

Output:

$\text{IMAX} = 2$

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:

<table>
<thead>
<tr>
<th>$N$</th>
<th>$X$</th>
<th>INCX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$\text{IMAX} = \text{ISMAX}(\ 4\ ,\ X\ ,\ -2\ )$

$X = (2.0, . , 4.0, . , 4.0, . , 1.0)$

Output:

$\text{IMAX} = 3$
ISMIN and IDMIN (Position of the First or Last Occurrence of the Vector Element Having Minimum Value)

Purpose

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 75. Data Types

<table>
<thead>
<tr>
<th>$x$</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>ISMIN</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>IDMIN</td>
</tr>
</tbody>
</table>

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>ISMIN</td>
</tr>
<tr>
<td>C and C++</td>
<td>ismin</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_ismin</td>
</tr>
</tbody>
</table>

On Entry

$n$ is the number of elements in vector $x$. Specified as: an integer; $n \geq 0$.

$x$ is the vector $x$ of length $n$. Specified as: a one-dimensional array of (at least) length $1+(n-1)\mid\text{incx}\mid$, containing numbers of the data type indicated in Table 75.

$incx$ is the stride for vector $x$.

Specified as: an 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:

- an integer; $0 \leq i \leq n$ (for Fortran, C, and C++)
- a CBLAS_INDEX; $0 \leq i \leq n-1$
Notes

Declare the ISMIN and IDMIN functions in your program as returning an 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 [93 on page 1368]. 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
$n < 0$

Examples

Example 1
This example shows a vector, $x$, with a stride of 1.

Function Reference and Input:

<table>
<thead>
<tr>
<th>N</th>
<th>X</th>
<th>INCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMIN = ISMIN( 6 , X , 1 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$x = (3.0, 4.0, 1.0, 8.0, 1.0, 3.0)$

Output:

$IMIN = 3$

Example 2
This example shows a vector, $x$, with a stride greater than 1.

Function Reference and Input:

<table>
<thead>
<tr>
<th>N</th>
<th>X</th>
<th>INCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMIN = ISMIN( 4 , X , 2 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$x = (-3.0, . , -9.0, . , -8.0, . , 3.0)$

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:

\[
\begin{align*}
N & \quad X \quad \text{INCX} \\
\text{IMIN} & = \text{ISMIN}( 4, X, 2 ) \\
X & = (2.0, 1.0, 1.0, 4.0, 1.0) \\
\end{align*}
\]

Output:
\[
\text{IMIN} = 2
\]

**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:

\[
\begin{align*}
N & \quad X \quad \text{INCX} \\
\text{IMIN} & = \text{ISMIN}( 4, X, -2 ) \\
X & = (2.0, 1.0, 1.0, 4.0, 1.0) \\
\end{align*}
\]

Output:
\[
\text{IMIN} = 4
\]
SASUM, DASUM, SCASUM, and DZASUM (Sum of the Magnitudes of the Elements in a Vector)

Purpose

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>
<thead>
<tr>
<th>$x$</th>
<th>Result</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SASUM</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DASUM</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>SCASUM</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>DZASUM</td>
</tr>
</tbody>
</table>

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>SASUM</td>
</tr>
<tr>
<td>C and C++</td>
<td>sasum</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_sasum</td>
</tr>
</tbody>
</table>

On Entry

- $n$ is the number of elements in vector $x$. Specified as: an 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 76.
- $incx$ is the stride for vector $x$.
  Specified as: an 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 76.

Notes

Declare this function in your program as returning a value of the type indicated in Table 76.

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:

\[ \sum_{i=1}^{n} (|a_i| + |b_i|) = (|a_1| + |b_1|) + (|a_2| + |b_2|) + \ldots + (|a_n| + |b_n|) \]

where \( x_i = (a_i, b_i) \)

See reference [93 on page 1368]. 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 when the Altivec or VSX unit is not used.

**Error conditions**

**Computational Errors**

None

**Input-Argument Errors**

\( n < 0 \)

**Examples**

**Example 1**

This example shows a vector, \( x \), with a stride of 1.

Function Reference and Input:

\[
\text{SUMM} = \text{SASUM}(7, x, 1)
\]

\( X = (1.0, -3.0, -6.0, 7.0, 5.0, 2.0, -4.0) \)

Output:

\( \text{SUMM} = 28.0 \)

**Example 2**

This example shows a vector, \( x \), with a stride greater than 1.

Function Reference and Input:

\[
\text{SUMM} = \text{SASUM}(4, x, 2)
\]

\( X = (1.0, \ldots, -6.0, \ldots, 5.0, \ldots, -4.0) \)

Output:

\( \text{SUMM} = 16.0 \)

**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 nx1.

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)

Purpose

These subprograms perform the following computation, using the scalar \( \alpha \) and vectors \( x \) and \( y \):

\[
y \leftarrow y + \alpha x
\]

<table>
<thead>
<tr>
<th>Table 77. Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha, x, y )</td>
</tr>
<tr>
<td>Subprogram</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>SAXPY</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
<tr>
<td>DAXPY</td>
</tr>
<tr>
<td>Short-precision complex</td>
</tr>
<tr>
<td>CAXPY</td>
</tr>
<tr>
<td>Long-precision complex</td>
</tr>
<tr>
<td>ZAXPY</td>
</tr>
</tbody>
</table>

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 32.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SAXPY</th>
<th>DAXPY</th>
<th>CAXPY</th>
<th>ZAXPY (n, alpha, x, incx, y, incy)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>saxpy</td>
<td>daxpy</td>
<td>caxpy</td>
<td>zaxpy (n, alpha, x, incx, y, incy);</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_saxpy</td>
<td>cblas_daxpy</td>
<td>cblas_caxpy</td>
<td>cblas_zaxpy (n, alpha, x, incx, y, incy);</td>
</tr>
</tbody>
</table>

On Entry

- \( n \) is the number of elements in vectors \( x \) and \( y \).
  Specified as: an integer; \( n \geq 0 \).

- \( \alpha \) is the scalar \( \alpha \).
  Specified as: a number of the data type indicated in Table 77.

- \( x \) is the vector \( x \) of length \( n \). Specified as: a one-dimensional array of (at least) length \( 1+(n-1) | \text{incx} | \), containing numbers of the data type indicated in Table 77.

- \( \text{incx} \) is the stride for vector \( x \).
  Specified as: an integer. It can have any value.

- \( y \) is the vector \( y \) of length \( n \). Specified as: a one-dimensional array of (at least) length \( 1+(n-1) | \text{incy} | \), containing numbers of the data type indicated in Table 77.

- \( \text{incy} \) is the stride for vector \( y \).
  Specified as: an integer. It can have any value.

On Return
$y$ is the vector $y$, containing the results of the computation $y + \alpha x$. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 77 on page 248.

**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 75.

**Function**

The computation is expressed as follows:

$$
\begin{bmatrix}
y_1 \\
\vdots \\
y_n
\end{bmatrix} 
\leftarrow 
\begin{bmatrix}
y_1 \\
\vdots \\
y_n
\end{bmatrix} 
+ \alpha 
\begin{bmatrix}
x_1 \\
\vdots \\
x_n
\end{bmatrix}
$$

See reference [93 on page 1368]. If $alpha$ or $n$ is zero, no computation is performed. For CAXPY, intermediate results are accumulated in long precision when the Altivec or VSX unit is not used.

**Error conditions**

**Computational Errors**

None

**Input-Argument Errors**

$n < 0$

**Examples**

**Example 1**

This example shows vectors $x$ and $y$ with positive strides.

Call Statement and Input:

```
N ALPHA X INCX Y INCY
```

```
5 2.0 1.0 2.0
```

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

$Y = (1.0, . , 1.0, . , 1.0, . , 1.0, . , 1.0)$

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:
CALL SAXPY( 5, 2.0, X, 1, Y, -1 )

\[
\begin{align*}
X & = (1.0, 2.0, 3.0, 4.0, 5.0) \\
Y & = (5.0, 4.0, 3.0, 2.0, 1.0)
\end{align*}
\]

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:

\[
\begin{align*}
\text{CALL SAXPY( 5, 2.0, X, 0, Y, 1 )}
\end{align*}
\]

\[
\begin{align*}
X & = (1.0) \\
Y & = (5.0, 4.0, 3.0, 2.0, 1.0)
\end{align*}
\]

Output:

\[
Y = (7.0, 6.0, 5.0, 4.0, 3.0)
\]

Example 4

This example shows how \( \text{SAXPY} \) 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:

\[
\begin{align*}
\text{CALL SAXPY( 1, 2.0, X, 0, Y, 0 )}
\end{align*}
\]

\[
\begin{align*}
X & = (1.0) \\
Y & = (5.0)
\end{align*}
\]

Output:

\[
Y = (7.0)
\]

Example 5

This example shows how to use \( \text{CAXPY} \), where vectors \( x \) and \( y \) contain complex numbers. In this case, vectors \( x \) and \( y \) have positive strides.

Call Statement and Input:

\[
\begin{align*}
\text{CALL CAXPY( 3, \text{ALPHA}, X, 1, Y, 2 )}
\end{align*}
\]

\[
\begin{align*}
\text{ALPHA} & = (2.0, 3.0) \\
X & = ((1.0, 2.0), (2.0, 0.0), (3.0, 5.0)) \\
Y & = ((1.0, 1.0), \ldots, (0.0, 2.0), \ldots, (5.0, 4.0)) \\
Y & = ((-3.0, 8.0), \ldots, (4.0, 8.0), \ldots, (-4.0, 23.0))
\end{align*}
\]

250 ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
SCOPY, DCOPY, CCOPY, and ZCOPY (Copy a Vector)

Purpose

These subprograms copy vector $x$ to another vector, $y$:

$$y = x$$

Table 78. Data Types

<table>
<thead>
<tr>
<th>$x$, $y$</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SCOPY</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DCOPY</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CCOPY</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZCOPY</td>
</tr>
</tbody>
</table>

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 32.

Syntax

For Fortran

```fortran
CALL SCOPY | DCOPY | CCOPY | ZCOPY (n, x, incx, y, incy)
```

For C and C++

```c
scopy | dcopy | ccopy | zcopy (n, x, incx, y, incy);
```

For CBLAS

```c
clblas_scopy | cblas_dcopy | cblas_ccopy | cblas_zcopy (n, x, incx, y, incy);
```

On Entry

- $n$ is the number of elements in vectors $x$ and $y$.
  - Specified as: an integer; $n \geq 0$.
- $x$ is the vector $x$ of length $n$. Specified as: a one-dimensional array of (at least) length $1+(n-1) \mid \text{incx} \mid$, containing numbers of the data type indicated in Table 78.
- $\text{incx}$ is the stride for vector $x$.
  - Specified as: an integer. It can have any value.
- $y$ See On Return
- $\text{incy}$ is the stride for vector $y$. Specified as: an 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 $1+(n-1) \mid \text{incy} \mid$, containing numbers of the data type indicated in Table 78.

Notes

1. If you specify the same vector for $x$ and $y$, $\text{incx}$ and $\text{incy}$ 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 75.
Function

The copy is expressed as follows:

\[
\begin{bmatrix}
  y_1 \\
  \cdot \\
  \cdot \\
  \cdot \\
  y_n \\
\end{bmatrix}
\leftarrow
\begin{bmatrix}
  x_1 \\
  \cdot \\
  \cdot \\
  \cdot \\
  x_n \\
\end{bmatrix}
\]

See reference [93 on page 1368]. If \( n \) is 0, no copy is performed.

Error conditions

Computational Errors
None

Input-Argument Errors
\( n < 0 \)

Examples

Example 1

This example shows input vector \( x \) and output vector \( y \) with positive strides.

Call Statement and Input:

\[
\begin{array}{c}
N \quad X \quad \text{INCX} \quad Y \quad \text{INCY} \\
\text{CALL SCOPY( 5, X, 1, Y, 2 )} \\
X & = (1.0, 2.0, 3.0, 4.0, 5.0) \\
Output: \\
Y & = (1.0, . , 2.0, . , 3.0, . , 4.0, . , 5.0)
\end{array}
\]

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:

\[
\begin{array}{c}
N \quad X \quad \text{INCX} \quad Y \quad \text{INCY} \\
\text{CALL SCOPY( 5, X, 1, Y, -1 )} \\
X & = (1.0, 2.0, 3.0, 4.0, 5.0) \\
Output: \\
Y & = (5.0, 4.0, 3.0, 2.0, 1.0)
\end{array}
\]

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:
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:

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)

**Purpose**

SDOT, DDOT, CDOTU, and ZDOTU compute the dot product of vectors $x$ and $y$:

$$x \cdot y$$

CDOTC and ZDOTC compute the dot product of the complex conjugate of vector $x$ with vector $y$:

$$\bar{x} \cdot y$$

**Table 79. Data Types**

<table>
<thead>
<tr>
<th>$x$, $y$, $dotu$, $dotc$, Result</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SDOT</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DDOT</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CDOTU and CDOTC</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZDOTU and ZDOTC</td>
</tr>
</tbody>
</table>

**Note:** On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

**Syntax**

<table>
<thead>
<tr>
<th></th>
<th>SDOT</th>
<th>DDOT</th>
<th>CDOTU</th>
<th>ZDOTU</th>
<th>CDOTC</th>
<th>ZDOTC $(n, x, \text{incx}, y, \text{incy})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORTRAN</td>
<td>sdot</td>
<td>ddot</td>
<td>cdotu</td>
<td>zdotu</td>
<td>cdotc</td>
<td>zdotc $(n, x, \text{incx}, y, \text{incy})$</td>
</tr>
<tr>
<td>C and C++</td>
<td>cblas_sdot</td>
<td>cblas_ddot $(n, x, \text{incx}, y, \text{incy})$;</td>
<td>cblas_cdotu_sub</td>
<td>cblas_zdotu_sub $(n, x, \text{incx}, y, \text{incy}, dotu)$;</td>
<td>cblas_cdotc_sub</td>
<td>cblas_zdotc_sub $(n, x, \text{incx}, y, \text{incy}, dotc)$;</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_cdotu_sub</td>
<td>cblas_zdotu_sub $(n, x, \text{incx}, y, \text{incy}, dotu)$;</td>
<td>cblas_cdotc_sub</td>
<td>cblas_zdotc_sub $(n, x, \text{incx}, y, \text{incy}, dotc)$;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**On Entry**

- $n$ is the number of elements in vectors $x$ and $y$.
  Specified as: an integer; $n \geq 0$.
- $x$ is the vector $x$ of length $n$. Specified as: a one-dimensional array of (at least) length $1+(n-1)|\text{incx}|$, containing numbers of the data type indicated in *Table 79*.
- $\text{incx}$ is the stride for vector $x$.
  Specified as: an integer. It can have any value.
- $y$ is the vector $y$ of length $n$. Specified as: a one-dimensional array of (at least) length $1+(n-1)|\text{incy}|$, containing numbers of the data type indicated in *Table 79*.  
**incy**

is the stride for vector \( y \).

Specified as: an 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 79 on page 254.

**dotu**

is the result of the dot product computation.

Returned as: a number of the data type indicated in Table 79 on page 254.

**dotc**

is the result of the dot product computation.

Returned as: a number of the data type indicated in Table 79 on page 254.

**Notes**

Declare this function in your program as returning a value of the data type indicated in Table 79 on page 254.

**Function**

SDOT, DDOT, CDOTU, and ZDOTU compute the dot product of the vectors \( x \) and \( y \), which is expressed as follows:

\[
x \cdot y = x_1y_1 + x_2y_2 + \ldots + x_ny_n
\]

CDOTC and ZDOTC compute the dot product of the complex conjugate of vector \( x \) with vector \( y \), which is expressed as follows:

\[
\bar{x} \cdot y = \bar{x}_1y_1 + \bar{x}_2y_2 + \ldots + \bar{x}_ny_n
\]

See reference [93 on page 1368]. 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 when the AltiVec or VSX unit is not used.

**Error conditions**

**Computational Errors**

None

**Input-Argument Errors**

\( n < 0 \)

**Examples**

**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:
\[ N \times \text{INCX} \times \text{INCY} \]
\[ \text{DOTT} = \text{SDOT}(5, X, 1, Y, 1) \]
\[ X = (1.0, 2.0, -3.0, 4.0, 5.0) \]
\[ Y = (9.0, 8.0, 7.0, -6.0, 5.0) \]

Output:
\[ \text{DOTT} = (9.0 + 16.0 - 21.0 - 24.0 + 25.0) = 5.0 \]

**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:
\[ N \times \text{INCX} \times \text{INCY} \]
\[ \text{DOTT} = \text{SDOT}(5, X, 1, Y, 2) \]
\[ X = (1.0, 2.0, -3.0, 4.0, 5.0) \]
\[ Y = (9.0, . , 7.0, . , 5.0, . , -3.0, . , 1.0) \]

Output:
\[ \text{DOTT} = (9.0 + 14.0 - 15.0 - 12.0 + 5.0) = 1.0 \]

**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:
\[ N \times \text{INCX} \times \text{INCY} \]
\[ \text{DOTT} = \text{SDOT}(5, X, -1, Y, 2) \]
\[ X = (1.0, 2.0, -3.0, 4.0, 5.0) \]
\[ Y = (9.0, . , 7.0, . , 5.0, . , -3.0, . , 1.0) \]

Output:
\[ \text{DOTT} = (45.0 + 28.0 - 15.0 - 6.0 + 1.0) = 53.0 \]

**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 \( \text{DOTT} \) is \( x_1(y_1 + ... + y_n) \).

Function Reference and Input:
\[ N \times \text{INCX} \times \text{INCY} \]
\[ \text{DOTT} = \text{SDOT}(5, X, 0, Y, 1) \]
\[ X = (1.0, . , . , . , .) \]
\[ Y = (9.0, 8.0, 7.0, -6.0, 5.0) \]

Output:
\[ \text{DOTT} = (1.0) \times (9.0 + 8.0 + 7.0 - 6.0 + 5.0) = 23.0 \]

**Example 5**

This example shows how to compute the dot product of two vectors, \( x \) and \( y \), with strides of 0. The result in \( \text{DOTT} \) is \( nx_1y_1 \).

Function Reference and Input:
\[
N \quad X \quad \text{INCX} \quad Y \quad \text{INCY}
\]
\[
\begin{align*}
\text{DOTT} &= \text{SDOT} (5, X, 0, Y, 0) \\
X &= (1.0, \ldots, 9.0, \ldots) \\
Y &= (9.0, \ldots) \\
\text{Output:} \\
\text{DOTT} &= (5) \times (1.0) \times (9.0) = 45.0
\end{align*}
\]

**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:
\[
\begin{align*}
N \quad X \quad \text{INCX} \quad Y \quad \text{INCY}
\end{align*}
\]
\[
\begin{align*}
\text{DOTT} &= \text{CDOTU} (3, X, 1, Y, 2) \\
X &= ((1.0, 2.0), (3.0, -4.0), (-5.0, 6.0)) \\
Y &= ((10.0, 9.0), \ldots) \\
\text{Output:} \\
\text{DOTT} &= ((10.0 - 18.0 - 10.0) - (18.0 - 20.0 + 6.0), \\
(9.0 + 15.0 - 5.0) + (20.0 + 24.0 + 12.0)) \\
&= (-22.0, 75.0)
\end{align*}
\]

**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:
\[
\begin{align*}
N \quad X \quad \text{INCX} \quad Y \quad \text{INCY}
\end{align*}
\]
\[
\begin{align*}
\text{DOTT} &= \text{CDOTC} (3, X, 1, Y, 2) \\
X &= ((1.0, 2.0), (3.0, -4.0), (-5.0, 6.0)) \\
Y &= ((10.0, 9.0), \ldots) \\
\text{Output:} \\
\text{DOTT} &= ((10.0 - 18.0 - 10.0) + (18.0 - 20.0 + 6.0), \\
(9.0 + 15.0 - 5.0) - (20.0 + 24.0 + 12.0)) \\
&= (-14.0, -37.0)
\end{align*}
\]
SNAXPY and DNAXPY (Compute SAXPY or DAXPY N Times)

Purpose

These subprograms compute SAXPY or DAXPY, respectively, \( n \) times:

\[
y_i \leftarrow y_i + \alpha x_i \quad \text{for } i = 1, n
\]

where each \( \alpha \) is a scalar value, contained in the vector \( a \), and each \( x_i \) and \( y_i \) are vectors, contained in vectors (or matrices) \( x \) and \( y \), 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 248.

Table 80. Data Types

<table>
<thead>
<tr>
<th>Subprogram</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SNAXPY</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DNAXPY</td>
</tr>
</tbody>
</table>

Syntax

Fortran: \( \text{CALL SNAXPY \mid DNAXPY} (n, m, a, \text{inca}, x, \text{incxi}, \text{incxo}, y, \text{incyi}, \text{incyo}) \)

C and C++: \( \text{snaxpy \mid dnaxpy} (n, m, a, \text{inca}, x, \text{incxi}, \text{incxo}, y, \text{incyi}, \text{incyo}); \)

On Entry

\( n \) is the number of SAXPY or DAXPY computations to be performed and the number of elements in vector \( a \).

Specified as: an integer; \( n \geq 0 \).

\( m \) is the number of elements in vectors \( x_i \) and \( y_i \) for each SAXPY or DAXPY computation.

Specified as: an integer; \( m \geq 0 \).

\( a \) is the vector \( a \) of length \( n \), containing the scalar values \( \alpha \), used in each computation of \( y_i + \alpha x_i \).

Specified as: a one-dimensional array of (at least) length \( 1+(n-1)\mid \text{inca} \mid \), containing numbers of the data type indicated in Table 80.

\( \text{inca} \)

is the stride for vector \( a \).

Specified as: an integer. It can have any value.

\( x \) is the vector (or matrix) \( x \), containing the \( x_i \) vectors of length \( m \), used in the \( n \) computations of \( y_i + \alpha x_i \).

Specified as: a one- or two-dimensional array of (at least) length \( (1+(n-1)\mid \text{incxo} \mid) + (m-1)\mid \text{incxi} \mid \), containing numbers of the data type indicated in Table 80.

\( \text{incxi} \)

is the stride for \( x \) in the inner loop—that is, the stride identifying the elements in each vector \( x_i \).

Specified as: an integer. It can have any value.
\textit{incxo}

is the stride for $x$ in the outer loop—that is, the stride identifying each vector $x_i$ in $x$.

Specified as: an integer; \textit{incxo} \geq 0.

\textit{y} is the vector (or matrix) $y$, containing the $y_i$ vectors of length $m$, used in the $n$ computations of $y_i + \alpha_i x_i$. Specified as: a one- or two-dimensional array of (at least) length $(1+(n-1)(\textit{incy})) + (m-1)|\textit{incy}|$, containing numbers of the data type indicated in Table 80 on page 258.

\textit{incy} is the stride for $y$ in the inner loop—that is, the stride identifying the elements in each vector $y_i$ in $y$. Specified as: an integer; \textit{incy} > 0 or \textit{incy} < 0.

\textit{incyo} is the stride for $y$ in the outer loop—that is, the stride identifying each vector $y_i$ in $y$.

Specified as: an integer; \textit{incyo} \geq 0.

\textbf{On Return}

\textit{y} is the vector (or matrix) $y$, containing the $y_i$ vectors of length $m$, which contain the results of the $n$ SAXPY or DAXPY computations, $y_i + \alpha_i x_i$ for $i = 1, n$.

Returned as: a one- or two-dimensional array, containing numbers of the data type indicated in Table 80 on page 258.

\textbf{Notes}

Vector $y$ must have no common elements with vector $a$ or vector $x$; otherwise, results are unpredictable. See \textbf{“Concepts” on page 75}.

\textbf{Function}

The SAXPY or DAXPY computations:

\[ y + y + \alpha x \]

are performed $n$ times. This is expressed as follows:

\[ y_i \leftarrow y_i + \alpha_i x_i \quad \text{for } i = 1, n \]

where each $\alpha_i$ is a scalar value, contained in the vector $a$, and each $x_i$ and $y_i$ are vectors, contained in vectors (or matrices) $x$ and $y$, respectively.

Each computation of SAXPY or DAXPY (see \textbf{“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 248}) uses the length of the $x$, and $y_i$ vectors, $m$, for its input argument, $n$. It also uses the strides for the inner loop, \textit{inxi} and \textit{incy}, for its parameters \textit{inx} and \textit{incy}, respectively. See \textbf{“Function” on page 249} for a description of how the computation is done.

The outer loop of the SNAXPY or DNAXPY computation uses the strides of $\textit{inca}$, \textit{incxo}, and \textit{incyo} to locate the elements in $a$ and vectors in $x$ and $y$ for each $i$-th computation. These are:

For $i = 1, n$:
\[ \alpha_{(i-1)\text{inc}a+1} \quad \text{for } \text{inc}a \geq 0 \]
\[ \alpha_{(i-n)\text{inc}a+1} \quad \text{for } \text{inc}a < 0 \]
\[ X_{(i-1)\text{inc}x0+1} \]
\[ y_{(i-1)\text{inc}y0+1} \]

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. \( \text{inc}x0 < 0 \)
4. \( \text{inc}yi = 0 \)
5. \( \text{inc}yo < 0 \)

**Examples**

**Example 1**

This example shows vectors, contained in matrices, with the stride of the inner loops \( \text{inc}xi \) and \( \text{inc}yi \) 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 )
```

\( A = (3.0, 2.0, 4.0) \)

\[
\begin{bmatrix}
1.0 & 4.0 & 3.0 \\
2.0 & 3.0 & 4.0 \\
3.0 & 2.0 & 2.0 \\
4.0 & 1.0 & 1.0
\end{bmatrix}
\]

\( X = \cdot \cdot \cdot \)

\[
\begin{bmatrix}
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot
\end{bmatrix}
\]

\( Y = \)

\[
\begin{bmatrix}
4.0 & 1.0 & 3.0 \\
3.0 & 2.0 & 4.0 \\
2.0 & 3.0 & 2.0 \\
1.0 & 4.0 & 1.0
\end{bmatrix}
\]

Output:

\( Y = \)

\[
\begin{bmatrix}
7.0 & 9.0 & 15.0 \\
9.0 & 8.0 & 20.0 \\
11.0 & 7.0 & 10.0 \\
13.0 & 6.0 & 5.0
\end{bmatrix}
\]

**Example 2**
This example shows vectors, contained in matrices, with a stride of the inner loop \textit{incxi} greater than 1.

**Call Statement and Input:**

\[
\text{CALL SNA\textsc{xpy}(} 3 \ , \ 4 \ , \ A \ , \ 1 \ , \ X \ , \ 2 \ , \ 10 \ , \ Y \ , \ 1 \ , \ 5 \ )
\]

\[
A = (3.0, 2.0, 4.0)
\]

\[
\begin{bmatrix}
1.0 & 4.0 & 3.0 \\
2.0 & 3.0 & 4.0 \\
3.0 & 2.0 & 2.0 \\
4.0 & 1.0 & 1.0 \\
\end{bmatrix}
\]

\[
X = 
\begin{bmatrix}
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\end{bmatrix}
\]

\[
Y = 
\begin{bmatrix}
4.0 & 1.0 & 3.0 \\
3.0 & 2.0 & 4.0 \\
2.0 & 3.0 & 2.0 \\
1.0 & 4.0 & 1.0 \\
\end{bmatrix}
\]

**Output:**

\[
Y = \text{\textit{\textit{(same as output Y in Example 1)}}}
\]

**Example 3**

This example shows vectors, contained in matrices, with a negative stride, \textit{incy}, for the inner loop.

**Call Statement and Input:**

\[
\text{CALL SNA\textsc{xpy}(} 3 \ , \ 4 \ , \ A \ , \ 1 \ , \ X \ , \ 1 \ , \ 10 \ , \ Y \ , \ -1 \ , \ 5 \ )
\]

\[
A = (3.0, 2.0, 4.0)
\]

\[
\begin{bmatrix}
1.0 & 4.0 & 3.0 \\
2.0 & 3.0 & 4.0 \\
3.0 & 2.0 & 2.0 \\
4.0 & 1.0 & 1.0 \\
\end{bmatrix}
\]

\[
X = 
\begin{bmatrix}
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\end{bmatrix}
\]

\[
Y = 
\begin{bmatrix}
1.0 & 4.0 & 1.0 \\
2.0 & 3.0 & 2.0 \\
3.0 & 2.0 & 4.0 \\
4.0 & 1.0 & 3.0 \\
\end{bmatrix}
\]

**Output:**
Example 4

This example shows vectors, contained in matrices, with a negative stride, \textit{inca},
for vector \textit{a}. For vector \textit{a}, processing begins at element \textit{A}(5), which is 3.0.

Call Statement and Input:

\begin{verbatim}
CALL SNAXPY( 3, 4, A, -2, X, 1, 10, Y, 1, 5 )
\end{verbatim}

\begin{align*}
A &= (4.0, \ldots, 2.0, \ldots, 3.0) \\
X &= \begin{bmatrix}
1.0 & 4.0 & 3.0 \\
2.0 & 3.0 & 4.0 \\
3.0 & 2.0 & 2.0 \\
4.0 & 1.0 & 1.0 \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
\end{bmatrix} \\
Y &= \begin{bmatrix}
4.0 & 1.0 & 3.0 \\
3.0 & 2.0 & 4.0 \\
2.0 & 3.0 & 2.0 \\
1.0 & 4.0 & 1.0 \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
\end{bmatrix}
\end{align*}

Output:

\begin{align*}
Y &= \text{(same as output } Y \text{ in Example 1)}
\end{align*}
SNDOT and DNDOT (Compute Special Dot Products N Times)

Purpose

These subprograms compute one of the following special dot products n times:

\[ s_i \times x_i \times y_i \]

- Store positive dot product
- Store negative dot product
- Accumulate positive dot product
- Accumulate negative dot product

for \( i = 1, n \)

where each \( s_i \) is an element in vector \( s \), and each \( x_i \) and \( y_i \) are vectors contained in vectors (or matrices) \( x \) and \( y \), respectively.

<table>
<thead>
<tr>
<th>Table 81. Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s, x, y )</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
</tbody>
</table>

Syntax

**Fortran**

```fortran
CALL SNDOT | DNDOT (n, m, s, incs, isw, x, incxi, incxo, y, incyi, incyo)
```

**C and C++**

```c
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 \( s \).
  - Specified as: an integer; \( n \geq 0 \).
- \( m \) is the number of elements in vectors \( x_i \) and \( y_i \) for each dot product computation.
  - Specified as: an integer; \( m \geq 0 \).
- \( s \) is the vector \( s \), 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 \( 1+(n-1) \times \text{incs} \), containing numbers of the data type indicated in [Table 81](#).
- \( incs \)
  - is the stride for vector \( s \).
  - Specified as: an integer; \( incs > 0 \) or \( incs < 0 \).
- \( isw \)
  - indicates the type of computation to perform, depending on the value specified:
    - If \( isw = 1 \), \( s_i \times x_i \times y_i \)
    - If \( isw = 2 \), \( s_i \times -x_i \times y_i \)
    - If \( isw = 3 \), \( s_i \times s_i + x_i \times y_i \)
If $isw = 4$, 
\[ s_i \leftarrow s_i - x_i \cdot y_i \]

where $i = 1, n$

Specified as: an integer. Its value must be 1, 2, 3, or 4.

$x$ is the vector (or matrix) $x$, containing the $x_i$ 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 81 on page 263.

$incxi$ is the stride for $x$ in the inner loop—that is, the stride identifying the elements in each vector $x_i$.

Specified as: an integer. It can have any value.

$incxo$ is the stride for $x$ in the outer loop—that is, the stride identifying each vector $x_i$ in $x$.

Specified as: an integer; $incxo \geq 0$.

$y$ is the vector (or matrix) $y$, containing the $y_i$ 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 81 on page 263.

$incyi$ is the stride for $y$ in the inner loop—that is, the stride identifying the elements in each vector $y_i$.

Specified as: an integer. It can have any value.

$incyo$ is the stride for $y$ in the outer loop—that is, the stride identifying each vector $y_i$ in $y$.

Specified as: an integer; $incyo \geq 0$.

On Return $s$ is the vector $s$ 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 $isw = 1$, 
\[ s_i \leftarrow x_i \cdot y_i \]
If $isw = 2$, 
\[ s_i \leftarrow -x_i \cdot y_i \]
If $isw = 3$, 
\[ s_i \leftarrow s_i + x_i \cdot y_i \]
If $isw = 4$, 
\[ s_i \leftarrow s_i - x_i \cdot y_i \]

where $i = 1, n$

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 81 on page 263.

Function

The four possible computations that can be performed by these subprograms are:

- $s_i \leftarrow x_i \cdot y_i$: Store positive dot product
- $s_i \leftarrow -x_i \cdot y_i$: Store negative dot product
$$s_i \leftarrow s_i + x_i \cdot y_i$$  
Accumulate positive dot product

$$s_i \leftarrow s_i - x_i \cdot y_i$$  
Accumulate negative dot product

for $i = 1, n$

where each $s_i$ is a scalar element in the vector $s$ of length $n$, and each of the $n$ $x_i$ and $y_i$ vectors of length $m$ are contained in vectors (or matrices) $x$ and $y$, respectively. Each computation uses the dot product, which is expressed:

$$x_i \cdot y_i = u_1 v_1 + u_2 v_2 + \ldots + u_m v_m$$

where $u_i$ and $v_i$ are elements of $x_i$ and $y_i$, respectively. To find the elements for the computation, it uses:

- The strides for the inner loops, $\text{incxi}$ and $\text{incyi}$, to locate the elements in vectors $x_i$ and $y_i$, respectively.
- The strides for the outer loops, $\text{incs}$, $\text{incxo}$, and $\text{incyo}$, to locate the element $s_i$ in vector $s$ and the vectors $x_i$ and $y_i$ in vectors (or matrices) $x$ and $y$, respectively.

If $m$ or $n$ is 0, no computation is performed. For SNDOT, intermediate results are accumulated in long precision when the Altivec or VSX unit is not used.

**Error conditions**

**Computational Errors**

None

**Input-Argument Errors**

1. $n < 0$
2. $m < 0$
3. $\text{incs} = 0$
4. $\text{isw} < 1$ or $\text{isw} > 4$
5. $\text{incxo} < 0$
6. $\text{incyo} < 0$

**Examples**

**Example 1**

This example shows a store positive dot product computation using vectors with positive strides.

Call Statement and Input:

```plaintext
CALL SNDOT( 3 , 4 , 5 , 1 , 1 , X , 1 , 4 , Y , 1 , 4 )
```

$$X = \begin{bmatrix}
1.0 & 2.0 & 3.0 \\
2.0 & 3.0 & 4.0 \\
3.0 & 4.0 & 5.0 \\
4.0 & 5.0 & 6.0 \\
4.0 & 3.0 & 2.0 \\
\end{bmatrix}$$
Example 2
This example shows a store negative dot product computation using vectors with positive and negative strides.

Call Statement and Input:

```
CALL SNDOT(3, 4, S, -1, 2, X, 2, 10, Y, -1, 6)
```

Output:
```
S = (20.0, 36.0, 48.0)
```

Example 3
This example shows an accumulative positive dot product using vectors with positive and negative strides.

Call Statement and Input:

```
CALL SNDOT(3, 4, S, 1, 3, X, -2, 10, Y, 2, 10)
```

Output:
```
S = (2.0, 5.0, 8.0)
```
\[
\begin{bmatrix}
4.0 & 3.0 & 2.0 \\
& & \\
3.0 & 2.0 & 1.0 \\
& & \\
Y &= 2.0 & 1.0 & 4.0 \\
& & \\
1.0 & 4.0 & 3.0 \\
& & \\
& & \\
& & \\
\end{bmatrix}
\]

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:

```plaintext
CALL SNDOT( 3 , 4 , S , -1 , 4 , X , 1 , 6 , Y , 2 , 10 )
```

\[
S = (3.0, 6.0, 9.0)
\]

\[
\begin{bmatrix}
1.0 & 2.0 & 3.0 \\
2.0 & 3.0 & 4.0 \\
X &= 3.0 & 4.0 & 5.0 \\
4.0 & 5.0 & 6.0 \\
& & \\
& & \\
& & \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
4.0 & 3.0 & 2.0 \\
& & \\
3.0 & 2.0 & 1.0 \\
& & \\
Y &= 2.0 & 1.0 & 4.0 \\
& & \\
1.0 & 4.0 & 3.0 \\
& & \\
& & \\
& & \\
\end{bmatrix}
\]

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)

Purpose

These subprograms compute the Euclidean length ($l_2$ norm) of vector $x$, with scaling of input to avoid destructive underflow and overflow.

Table 82. Data Types

<table>
<thead>
<tr>
<th>$x$</th>
<th>Result</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SNRM2</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DNRM2</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>SCNRM2</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>DZNRM2</td>
</tr>
</tbody>
</table>

Note:

1. 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 271 explains how to estimate whether your data will cause an overflow or underflow.

2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

Fortran:  SNRM2 | DNRM2 | SCNRM2 | DZNRM2 ($n, x, incx$)
C and C++:  snrm2 | dnrm2 | scnrn2 | dznrm2 ($n, x, incx$);
CBLAS:  cblas_snrm2 | cblas_dnrm2 | cblas_scnrn2 | cblas_dznrm2 ($n, x, incx$);

On Entry

$n$

is the number of elements in vector $x$. Specified as: an 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 82.

$incx$

is the stride for vector $x$.

Specified as: an 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 82.
Notes

Declare this function in your program as returning a value of the data type indicated in Table 82 on page 268.

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 [93 on page 1368]. 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

$n < 0$

Examples

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: $3.37\times10^{-38}$ to $3.37\times10^{38}$
- For long-precision: $1.67\times10^{-308}$ to $1.67\times10^{308}$

Example 1

This example shows a vector, $x$, whose elements must be scaled to prevent overflow.

```
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 = 0.14690D+201
```

Example 2

This example shows a vector, $x$, whose elements must be scaled to prevent destructive underflow.

Function Reference and Input:
Example 3
This example shows a vector, \( x \), with a stride of 0. The result in \( \text{SNORM} \) is:

\[
\sqrt{\sum_{i=1}^{n} x_i^2}
\]

Function Reference and Input:

\[
\begin{align*}
N & \quad X & \quad \text{INCX} \\
\text{SNORM} & = & \text{SNRM2}(4, X, 0) \\
X & = & (4.0)
\end{align*}
\]

Output:

\[
\text{SNORM} = 8.0
\]

Example 4
This example shows a vector, \( x \), containing complex numbers, and whose elements must be scaled to prevent overflow.

Function Reference and Input:

\[
\begin{align*}
N & \quad X & \quad \text{INCX} \\
\text{DZNORM} & = & \text{DZNRM2}(3, X, 1) \\
X & = & ((0.68056D+200, 0.25521D+200), (0.34028D+200, 0.85071D+200), (0.25521D+200, 0.85071D+200))
\end{align*}
\]

Output:

\[
\text{DZNORM} = 0.1469D+201
\]

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:

\[
\begin{align*}
N & \quad X & \quad \text{INCX} \\
\text{DZNORM} & = & \text{DZNRM2}(2, X, 2) \\
X & = & ((0.10795D-200, 0.10795D-200), \ldots, (0.10795D-200, 0.10795D-200))
\end{align*}
\]

Output:

\[
\text{DZNORM} = 0.2159D-200
\]
SNORM2, DNORM2, CNORM2, and ZNORM2 (Euclidean Length of a Vector with No Scaling of Input)

Purpose

These subprograms compute the euclidean length ($l_2$ norm) of vector $x$ with no scaling of input.

Table 83. Data Types

<table>
<thead>
<tr>
<th>$x$</th>
<th>Result</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SNORM2</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DNORM2</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>CNORM2</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>ZNORM2</td>
</tr>
</tbody>
</table>

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

Fortran

`SNORM2 | DNORM2 | CNORM2 | ZNORM2 (n, x, incx)`

C and C++

`snorm2 | dnorm2 | cnorm2 | znorm2 (n, x, incx);`

On Entry

$n$ is the number of elements in vector $x$. Specified as: an 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 83.

$incx$ is the stride for vector $x$.

Specified as: an 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 83.

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 83 on page 271.

**Function**

The euclidean length \((l_2 \text{ norm})\) 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 [93 on page 1368]. 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 271.

**Error conditions**

**Computational Errors**

None

**Input-Argument Errors**

\(n < 0\)

**Examples**

**Example 1**

This example shows a vector, \(x\), with a stride of 1.

Function Reference and Input:

\[
\begin{array}{c|c|c|c}
N & X & \text{INCX} \\
\hline
6 & X & 1 \\
\end{array}
\]

\[
\text{SNORM} = \text{SNORM2}(6, X, 1)
\]

\(X = (3.0, 4.0, 1.0, 8.0, 1.0, 3.0)\)

Output:

\(\text{SNORM} = 10.0\)

**Example 2**

This example shows a vector, \(x\), with a stride greater than 1.

Function Reference and Input:

\[
\begin{array}{c|c|c|c}
N & X & \text{INCX} \\
\hline
6 & X & 2 \\
\end{array}
\]

\[
\text{SNORM} = \text{SNORM2}(6, X, 2)
\]

\(X = (3.0, ., 4.0, ., 1.0, ., 8.0, ., 1.0, ., 3.0)\)

Output:

\(\text{SNORM} = 10.0\)

**Example 3**

This example shows a vector, \(x\), with a stride of 0. The result in \(\text{SNORM}\) is:
\[ \sqrt{\sum_{i=1}^{n} x_i^2} \]

Function Reference and Input:

\[
\begin{align*}
N & \quad X & \quad \text{INCX} \\
\text{SNORM} & = \text{SNORM2}( 4, X, 0 ) \\
X & = (4.0) \\
\text{Output:} \\
\text{SNORM} & = 8.0
\end{align*}
\]

**Example 4**

This example shows a vector, \( x \), containing complex numbers and having a stride of 1.

Function Reference and Input:

\[
\begin{align*}
N & \quad X & \quad \text{INCX} \\
\text{CNORM} & = \text{CNORM2}( 3, X, 1 ) \\
X & = ((3.0, 4.0), (1.0, 8.0), (-1.0, 3.0)) \\
\text{Output:} \\
\text{CNORM} & = 10.0
\end{align*}
\]
SROTG, DROTG, CROTG, and ZROTG (Construct a Givens Plane Rotation)

Purpose

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 84. Data Types

<table>
<thead>
<tr>
<th>( a, b, r, s )</th>
<th>( c )</th>
<th>( z )</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SROTG</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DROTG</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>(No value returned)</td>
<td>CROTG</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>(No value returned)</td>
<td>ZROTG</td>
</tr>
</tbody>
</table>

Syntax

Fortran

CALL SROTG | DROTG | CROTG | ZROTG (\( a, b, c, s \))

C and C++

srotg | drotg | crotg | zrotg (\( a, b, c, s \));

CBLAS

cblas_srotg | cblas_drotg | cblas_crotg | cblas_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 84

\( b \) is the rotational elimination parameter \( b \).

Specified as: a number of the data type indicated in Table 84

\( c \) 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:

\[
\sigma = \text{SIGN}(a) \text{ if } |a| > |b| \\
\sigma = \text{SIGN}(b) \text{ if } |a| \leq |b|
\]
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:

\[ \psi = a / |a| \]

Returned as: a number of the data type indicated in Table 84 on page 274.

\( b \) is the value computed for \( z \).

For SROTG and DROTG:

\[ z = s \quad \text{if } |a| > |b| \]

\[ z = 1/c \quad \text{if } |a| \leq |b| \text{ and } c \neq 0 \text{ and } r \neq 0 \]

\[ z = 1 \quad \text{if } |a| \leq |b| \text{ and } c = 0 \text{ and } r \neq 0 \]

\[ z = 0 \quad \text{if } r = 0 \]

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 84 on page 274.

\( c \) is the cosine \( c \) of the angle of (Givens) rotation. For SROTG and DROTG:

\[ c = a/r \quad \text{if } r \neq 0 \]

\[ c = 1 \quad \text{if } r = 0 \]

For CROTG and ZROTG:

\[ c = |a| \sqrt{|a|^2 + |b|^2} / |a|^2 + |b|^2 \quad \text{if } |a| \neq 0 \]

\[ c = 0 \quad \text{if } |a| = 0 \]

 Returned as: a number of the data type indicated in Table 84 on page 274.

\( s \) is the sine \( s \) of the angle of (Givens) rotation.

For SROTG and DROTG:

\[ s = b/r \quad \text{if } r \neq 0 \]

\[ s = 0 \quad \text{if } r = 0 \]

For CROTG and ZROTG:
\[ s = \frac{\psi \bar{b}}{\sqrt{|a|^2 + |b|^2}} \quad \text{if} \quad |a| \neq 0 \]

\[ s = (10, 0, 0) \quad \text{if} \quad |a| = 0 \]

where \( \psi = a / |a| \)

Returned as: a number of the data type indicated in [Table 84 on page 274](#).

**Notes**

1. In your C program, arguments \( a, b, c, \) and \( s \) must be passed by reference.
2. In your C++ program, for cblas_srotg and cblas_drotg, arguments \( a, b, c, \) and \( s \) must be passed by pointer.

**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:

\[ \sigma = \text{SIGN}(a) \quad \text{if} \quad |a| > |b| \]
\[ \sigma = \text{SIGN}(b) \quad \text{if} \quad |a| \leq |b| \]
\[ c = a/r \quad \text{if} \quad r \neq 0 \]
\[ c = 1 \quad \text{if} \quad r = 0 \]
\[ s = b/r \quad \text{if} \quad r \neq 0 \]
\[ s = 0 \quad \text{if} \quad r = 0 \]
\[ z = z \quad \text{if} \quad |a| > |b| \]
\[ z = 1/c \quad \text{if} \quad |a| \leq |b| \quad \text{and} \quad c \neq 0 \quad \text{and} \quad r \neq 0 \]
\[ z = 1 \quad \text{if} \quad |a| \leq |b| \quad \text{and} \quad c = 0 \quad \text{and} \quad r \neq 0 \]
\[ z = 0 \quad \text{if} \quad r = 0 \]

See reference [93 on page 1368](#).

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:
3. $\sigma$ 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 \[110 on page 1369\]. $c$ and $s$ are reconstructed from $z$ as follows:

For $z = 1$, $c = 0$ and $s = 1$

For $|z| < 1$, $c = \sqrt{1-z^2}$ and $s = z$

For $|z| > 1$, $c = 1/z$ and $s = \sqrt{1-c^2}$

A complex Givens plane rotation is constructed for values $a$ and $b$ by computing values for $r$, $c$, and $s$, where:

\[
r = \sigma (|a| + |b|) \sqrt{\left(\frac{a}{|a| + |b|}\right)^2 + \left(\frac{b}{|a| + |b|}\right)^2}
\]

3. $\sigma$ 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 \[110 on page 1369\]. $c$ and $s$ are reconstructed from $z$ as follows:

For $z = 1$, $c = 0$ and $s = 1$

For $|z| < 1$, $c = \sqrt{1-z^2}$ and $s = z$

For $|z| > 1$, $c = 1/z$ and $s = \sqrt{1-c^2}$

A complex Givens plane rotation is constructed for values $a$ and $b$ by computing values for $r$, $c$, and $s$, where:

\[
r = \psi \sqrt{|a|^2 + |b|^2} \quad \text{if } |a| \neq 0
\]

\[
r = b \quad \text{if } |a| = 0
\]

where:

\[
\psi = a/|a|
\]

\[
c = \frac{|a|}{\sqrt{|a|^2 + |b|^2}} \quad \text{if } |a| \neq 0
\]

\[
c = 0 \quad \text{if } |a| = 0
\]

\[
s = \frac{\psi \bar{b}}{\sqrt{|a|^2 + |b|^2}} \quad \text{if } |a| \neq 0
\]

\[
s = (1,0,0) \quad \text{if } |a| = 0
\]

See reference \[93 on page 1368\].

Following are some important points about the computation:

1. The numbers for $c$, $s$, and $r$ satisfy:
\[
\begin{bmatrix}
  c & s \\
-\bar{s} & c \\
\end{bmatrix}
\begin{bmatrix}
  a \\
  b \\
\end{bmatrix} = \begin{bmatrix}
  r \\
  0 \\
\end{bmatrix}
\]

2. Where necessary, scaling is used to avoid overflow and destructive underflow in the computation of \( r \), which is expressed as follows:

\[
r = \psi \left( |a| + |b| \right) \sqrt{ \frac{a}{|a| + |b|}^2 + \frac{b}{|a| + |b|}^2 }
\]

**Error conditions**

**Computational Errors**
None

**Input-Argument Errors**
None

**Examples**

**Example 1**
This example shows the construction of a real Givens plane rotation, where \( r \) is 0.

Call Statement and Input:

\[
\begin{array}{|c|c|c|c|}
\hline
A & B & C & S \\
\hline
\end{array}
\]

CALL SROTG ( 0.0 , 0.0 , C , S )

Output:
\[
\begin{array}{c}
A = 0.0 \\
B = 0.0 \\
C = 1.0 \\
S = 0.0 \\
\end{array}
\]

**Example 2**
This example shows the construction of a real Givens plane rotation, where \( c \) is 0.

Call Statement and Input:

\[
\begin{array}{|c|c|c|c|}
\hline
A & B & C & S \\
\hline
\end{array}
\]

CALL SROTG ( 0.0 , 2.0 , C , S )

Output:
\[
\begin{array}{c}
A = 2.0 \\
B = 1.0 \\
C = 0.0 \\
S = 1.0 \\
\end{array}
\]

**Example 3**
This example shows the construction of a real Givens plane rotation, where \( |b| > |a| \).

Call Statement and Input:

\[
\begin{array}{|c|c|c|c|}
\hline
A & B & C & S \\
\hline
\end{array}
\]

CALL SROTG ( 6.0 , -8.0 , C , S )

278  ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
Output:
\[
\begin{align*}
A & = -10.0 \\
B & = -1.666 \\
C & = -0.6 \\
S & = 0.8 \\
\end{align*}
\]

Example 4
This example shows the construction of a real Givens plane rotation, where \(|a| > |b|\).

Call Statement and Input:
\[
\begin{align*}
A & \quad B & \quad C & \quad S \\
\left| \begin{array}{cccc}
8.0 & 6.0 & C & S \\
\end{array} \right| \\
\end{align*}
\]
CALL SROTG( 8.0, 6.0, C, S )

Output:
\[
\begin{align*}
A & = 10.0 \\
B & = 0.6 \\
C & = 0.8 \\
S & = 0.6 \\
\end{align*}
\]

Example 5
This example shows the construction of a complex Givens plane rotation, where \(|a| = 0\).

Call Statement and Input:
\[
\begin{align*}
A & \quad B & \quad C & \quad S \\
\left| \begin{array}{cccc}
A & B & C & S \\
\end{array} \right| \\
\end{align*}
\]
CALL CROTG( A, B, C, S )

Output:
\[
\begin{align*}
A & = (0.0, 0.0) \\
B & = (1.0, 0.0) \\
\end{align*}
\]

Example 6
This example shows the construction of a complex Givens plane rotation, where \(|a| \neq 0\).

Call Statement and Input:
\[
\begin{align*}
A & \quad B & \quad C & \quad S \\
\left| \begin{array}{cccc}
3.0 & 4.0 & C & S \\
\end{array} \right| \\
\end{align*}
\]
CALL CROTG( A, B, C, S )

Output:
\[
\begin{align*}
A & = (5.26, 7.02) \\
C & = 0.57 \\
S & = (0.82, -0.05) \\
\end{align*}
\]
SROT, DROT, CROT, ZROT, CSROT, and ZDROT (Apply a Plane Rotation)

**Purpose**

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 85. Data Types

<table>
<thead>
<tr>
<th>( x, y )</th>
<th>( c )</th>
<th>( s )</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SROT</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DROT</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>Short-precision complex</td>
<td>CROT</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>Long-precision complex</td>
<td>ZROT</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>CSROT</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>ZDROT</td>
</tr>
</tbody>
</table>

**Note:** On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

**Syntax**

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C and C++</th>
<th>CBLAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL SROT</td>
<td>DROT</td>
<td>CROT</td>
</tr>
</tbody>
</table>

**On Entry**

\( n \) is the number of points to be rotated—that is, the number of elements in vectors \( x \) and \( y \).

Specified as: an integer; \( n \geq 0 \).

\( x \) is the vector \( x \) of length \( n \), containing the \( x \) coordinates of the points to be rotated.

Specified as: a one-dimensional array of (at least) length \( 1+(n-1) |\text{incx}| \), containing numbers of the data type indicated in Table 85.

\( \text{incx} \)

is the stride for vector \( x \).

Specified as: an integer. It can have any value.

\( y \) is the vector \( y \) of length \( n \), containing the \( y \) coordinates of the points to be rotated.

Specified as: a one-dimensional array of (at least) length \( 1+(n-1) |\text{incy}| \), containing numbers of the data type indicated in Table 85.
inc\(y\)

is the stride for vector \(y\).

Specified as: an 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 85 on page 280.

\(s\)

the sine, \(s\), of the angle of rotation.

Specified as: a number of the data type indicated in Table 85 on page 280.

**On Return**

\(x\) is the vector \(x\) of length \(n\), containing the rotated \(x\_i\) coordinates, where:

\[
x_i \leftarrow cx_i + sy_i \quad \text{for } i = 1,
\]

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 85 on page 280.

\(y\) is the vector \(y\) of length \(n\), containing the rotated \(y\_i\) coordinates, where:

For SROT, DROT, CSROT, and ZDROT:

\[
y_i \leftarrow -sx_i + cy_i \quad \text{for } i = 1, n
\]

For CROT and ZROT:

\[
y_i \leftarrow -sy_i + cx_i \quad \text{for } i = 1, n
\]

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 85 on page 280.

**Notes**

The vectors \(x\) and \(y\) must have no common elements; otherwise, results are unpredictable. See “Concepts” on page 75.

**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:

\[
\begin{bmatrix}
x_i \\
y_i
\end{bmatrix} \leftarrow \begin{bmatrix}
c & s \\
-s & c
\end{bmatrix} \begin{bmatrix}
x_i \\
y_i
\end{bmatrix} \quad \text{for } i = 1, n
\]

See references [68 on page 1367] and [93 on page 1368]. 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 when the Altivector or VSX unit is not used.

**Error conditions**

**Computational Errors**

None

**Input-Argument Errors**

\( n < 0 \)

**Examples**

**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:

\[
\begin{array}{l}
N \quad X \quad \text{INCX} \quad Y \quad \text{INCY} \quad C \quad S \\
\text{CALL SROT( 5 , X , 1 , Y , 2 , 0.5 , S )}
\end{array}
\]

\[
\begin{align*}
X & = (1.0, 2.0, 3.0, 4.0, 5.0) \\
Y & = (-1.0, . , -2.0, . , -3.0, . , -4.0, . , -5.0)
\end{align*}
\]

\[
S = \frac{\sqrt{3.0}}{2.0}
\]

Output:

\[
\begin{align*}
X & = (-0.366, -0.732, -1.098, -1.464, -1.830) \\
Y & = (-1.366, -2.732, -4.098, -5.464, -6.830)
\end{align*}
\]

**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:

\[
\begin{array}{l}
N \quad X \quad \text{INCX} \quad Y \quad \text{INCY} \quad C \quad S \\
\text{CALL SROT( 5 , X , 1 , Y , -1 , 0.5 , S )}
\end{array}
\]

\[
\begin{align*}
X & = (1.0, 2.0, 3.0, 4.0, 5.0) \\
Y & = (-5.0, -4.0, -3.0, -2.0, -1.0)
\end{align*}
\]

\[
S = \frac{\sqrt{3.0}}{2.0}
\]

Output:

\[
\begin{align*}
X & = \text{(same as output } X \text{ in Example 1)} \\
Y & = (-6.830, -5.464, -4.098, -2.732, -1.366)
\end{align*}
\]

**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:

\[
\begin{align*}
N & \quad X \quad \text{INCX} \quad Y \quad \text{INCY} \quad C \quad S \\
\end{align*}
\]

\[\text{CALL SROT( 1 , X , 0 , Y , 0 , 0.5 , S )}\]

\[
\begin{align*}
X &= (1.0) \\
Y &= (-1.0) \\
\end{align*}
\]

\[S = \frac{\sqrt{3.0}}{2.0}\]

Output:

\[
\begin{align*}
X &= (-0.366) \\
Y &= (-1.366) \\
\end{align*}
\]

**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:

\[
\begin{align*}
N & \quad X \quad \text{INCX} \quad Y \quad \text{INCY} \quad C \quad S \\
\end{align*}
\]

\[\text{CALL CROT( 3 , X , 1 , Y , 2 , 0.5 , S )}\]

\[
\begin{align*}
X &= ((1.0, 2.0), (2.0, 3.0), (3.0, 4.0)) \\
Y &= ((-1.0, 5.0), . , (-2.0, 4.0), . , (-3.0, 3.0)) \\
S &= (0.75, 0.50) \\
\end{align*}
\]

Output:

\[
\begin{align*}
X &= ((-2.750, 4.250), (-2.500, 3.500), (-2.250, 2.750)) \\
Y &= ((-2.250, 1.500), . , (-4.000, 0.750), . , (-5.750, 0.000)) \\
\end{align*}
\]

**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:

\[
\begin{align*}
N & \quad X \quad \text{INCX} \quad Y \quad \text{INCY} \quad C \quad S \\
\end{align*}
\]

\[\text{CALL CSROT( 3 , X , 1 , Y , 2 , 0.5 , S )}\]

\[
\begin{align*}
X &= ((1.0, 2.0), (2.0, 3.0), (3.0, 4.0)) \\
Y &= ((-1.0, 5.0), . , (-2.0, 4.0), . , (-3.0, 3.0)) \\
\end{align*}
\]

\[S = \frac{\sqrt{3.0}}{2.0}\]

Output:

\[
\begin{align*}
X &= ((-0.366, 5.330), (-0.732, 4.964), (-1.098, 4.598)) \\
Y &= ((-1.366, 0.768), . , (-2.732, -0.598), . , (-4.098, -1.964)) \\
\end{align*}
\]
SROTMG and DROTMG (Construct a modified Givens Transformation)

Purpose

SROTMG and DROTMG construct the modified Givens transformation matrix $H$, which zeros the second component of the 2-vector:

$H$ has one of the following forms:

$$
\begin{bmatrix}
\sqrt{d} x_1 \\
\sqrt{d} x_2
\end{bmatrix}
$$

Figure 11. 2-Vector

$H$ has one of the following forms:

<table>
<thead>
<tr>
<th>PARAM</th>
<th>$H$ = $\begin{bmatrix} h_{11} &amp; h_{12} \ h_{21} &amp; h_{22} \end{bmatrix}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.0</td>
<td>$H$ = $\begin{bmatrix} h_{11} &amp; h_{12} \ h_{21} &amp; h_{22} \end{bmatrix}$</td>
</tr>
<tr>
<td>0.0</td>
<td>$H$ = $\begin{bmatrix} 1.0 &amp; h_{12} \ h_{21} &amp; 1.0 \end{bmatrix}$</td>
</tr>
<tr>
<td>1.0</td>
<td>$H$ = $\begin{bmatrix} h_{11} &amp; 1.0 \ -1.0 &amp; h_{22} \end{bmatrix}$</td>
</tr>
<tr>
<td>-2.0</td>
<td>$H$ = $\begin{bmatrix} 1.0 &amp; 0.0 \ 0.0 &amp; 1.0 \end{bmatrix}$</td>
</tr>
</tbody>
</table>

Figure 12. The form of matrix $H$

SROTM and DROTM uses the information in PARAM to apply the modified Givens transformation.

Table 86. Data Types

<table>
<thead>
<tr>
<th>$d1$, $d2$, $x1$, $x2$</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SROTMG</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DROTMG</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SROTMG</th>
<th>DROTMG</th>
<th>$(d1, d2, x1, x2, param)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>srotmg</td>
<td>drotmg</td>
<td>$(d1, d2, x1, x2, param)$;</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_srotmg</td>
<td>cblas_drotmg</td>
<td>$(d1, d2, x1, x2, param)$;</td>
</tr>
</tbody>
</table>

On Entry

- $d1$ is the value $d_1$ shown in Figure 11
- Specified as: a number of the data type indicated in Table 86
- $d2$ is the value $d_2$ shown in Figure 11
- Specified as: a number of the data type indicated in Table 86
- $x1$ is the value $x_1$ shown in Figure 11
- Specified as: a number of the data type indicated in Table 86
\( x_2 \) is the value \( x_2 \) shown in Figure 11 on page 284. Specified as: a number of the data type indicated in Table 86 on page 284.

On Return

\( d_1 \)  \( d_1 \) is overwritten; that is, the original input is not preserved.

Returned as: a number of the data type indicated in Table 86 on page 284

\( d_2 \)  \( d_2 \) is overwritten; that is, the original input is not preserved.

Returned as: a number of the data type indicated in Table 86 on page 284

\( x_1 \)  \( x_1 \) is overwritten; that is, the original input is not preserved.

Returned as: a number of the data type indicated in Table 86 on page 284

\( x_2 \)  \( x_2 \) is overwritten; that is, the original input is not preserved.

Returned as: a number of the data type indicated in Table 86 on page 284

\( \text{param} \)

\( \text{param} \) is a vector of length 5 defining the form and element values of the modified Givens transformation matrix \( H \), where:

\[
\begin{align*}
\text{param}_1 & = \text{Form of } H, \text{ either -1.0, 0.0, 1.0, or -2.0, as shown in Figure 12 on page 284} \\
\text{param}_2 & = H_{11} \\
\text{param}_3 & = H_{21} \\
\text{param}_4 & = H_{12} \\
\text{param}_5 & = H_{22}
\end{align*}
\]

The matrix \( H \) values of 1.0, -1.0, 0.0 implied when \( \text{param}_1 = 0, 1, -2 \) are assumed and are not stored in \( \text{param}_{2:5} \).

SROTM and DROTM use the information in \( \text{param} \) to apply the modified Givens transformation.

Returned as: a vector of length 5.

Notes
1. In your C program, arguments \( d_1, d_2, x_1, \) and \( x_2 \) must be passed by reference.
2. In your C++ program, for cblas_srotmg and cblas_drotmg, arguments \( d_1, d_2, x_1, \) and \( x_2 \) must be passed by pointer.

Function

SROTMG and DROTMG construct the modified Givens transformation matrix \( H \), which zeros the second component of the 2-vector:

\[
\begin{bmatrix}
\sqrt{d} x_1 \\
\sqrt{d} x_2
\end{bmatrix}
\]

\( H \) has one of the following forms:
SROTM and DROTM uses the information in PARAM to apply the modified Givens transformation.

For more information, see references [93 on page 1368].

**Error conditions**

**Computational Errors**

None

**Input-Argument Errors**

None

**Examples**

See the combined DROTMG and DROTM example shown in “Example 1” on page 289.
SROTM and DROTM (Apply a modified Givens Transformation)

Purpose

SROTM and DROTM apply a modified GIVENS matrix transformation. For \( I=1,n \):

\[
\begin{bmatrix}
X_i \\
Y_i
\end{bmatrix}
\leftarrow H
\begin{bmatrix}
X_i \\
Y_i
\end{bmatrix}
\]

Where \( H \) is a modified Givens transformation returned by a preceding call to SROTMG or DROTMG respectively.

For \( I = 1,n \):

\[
x_i = h_{11} x_i + h_{12} y_i
\]

\[
y_i = h_{21} x_i + h_{22} y_i
\]

Table 87. Data Types

<table>
<thead>
<tr>
<th>x, y</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SROTM</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DROTM</td>
</tr>
</tbody>
</table>

Syntax

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fortran</strong></td>
<td>CALL SROTM</td>
</tr>
<tr>
<td><strong>C and C++</strong></td>
<td>srotm</td>
</tr>
<tr>
<td><strong>CBLAS</strong></td>
<td>cblas_srotm</td>
</tr>
</tbody>
</table>

On Entry

- \( n \) is the number of elements in vectors \( x \) and \( y \).
  Specified as: an integer; \( n \geq 0 \).
- \( x \) is the vector \( x \) of length \( n \).
  Specified as: a one-dimensional array of (at least) length \( 1+|n-1| \cdot |incx| \), containing numbers of the data type indicated in Table 87.
- \( incx \) is the stride for vector \( x \).
  Specified as: an integer. It can have any value.
- \( y \) is the vector \( y \) of length \( n \).
  Specified as: a one-dimensional array of (at least) length \( 1+|n-1| \cdot |incy| \), containing numbers of the data type indicated in Table 87.
- \( incy \) is the stride for vector \( y \).
  Specified as: an integer. It can have any value.
- \( param \) is a vector of length 5 as returned by a preceding call to SROTMG or DROTMG defining the form and element values of the modified Givens transformation matrix \( H \), where:
param_i = Form of H, either -1.0, 0.0, 1.0, or -2.0, as shown in Figure 12 on page 284.

param_2 = H_{11}
param_3 = H_{21}
param_4 = H_{12}
param_5 = H_{22}

The matrix H values of 1.0, -1.0, 0.0 implied when param_1 = 0, 1, -2 are assumed and do not need to be stored in param_2:5.

Returned as: a vector of length 5.

On Return
x is the vector x of length n, where:

\[ x_i = h_{11}x_i + h_{12}y_i \]

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 87 on page 287.

y is the vector y of length n, where:

\[ y_i = h_{21}x_i + h_{22}y_i \]

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 87 on page 287.

Notes
- The vectors x and y must have no common elements; otherwise, results are unpredictable. See “Concepts” on page 75.
- If negative stride is specified for a vector, then the position of the first element accessed is \((-n+1)inc + 1\), where n = vector count and inc = vector stride, except when stride of both vectors is -1, in which case both vectors are accessed from position 1. For all other strides, the position of the first element accessed is 1.

Function

SROTM and DROTM apply a modified GIVENS matrix transformation. For I=1,n:

\[
\begin{bmatrix}
X_i \\
Y_i
\end{bmatrix}
\leftarrow -H
\begin{bmatrix}
X_j \\
Y_j
\end{bmatrix}
\]

Where H is a modified Givens transformation matrix returned by a preceding call to SROTMG or DROTMG respectively.

For I = 1,n:

\[ x_i = h_{11}x_i + h_{12}y_i \]
\[ y_i = h_{21}x_i + h_{22}y_i \]

For more information, see references [93 on page 1368].

If n = 0 or if H is the identity matrix (param_1 = -2), the subroutine returns immediately.
Error conditions

Computational Errors
None

Input-Argument Errors

\( n < 0 \)

Examples

Example 1

This example illustrates the construction and application of a modified Givens transformation.

Call Statement and Input:

\[
\begin{array}{cccc|c}
D1 & D2 & X1 & X2 & PARAM \\
\hline
\end{array}
\]

\[
\text{CALL DROTMG( D1, D2, X1, X2, PARAM)}
\]

\[
D1 = 1000000000.0 \\
D2 = 1.0 \\
X1 = -2.0 \\
X2 = 4.0
\]

Output:

\[
\text{PARAM} = (-1.0, 4096.0, 2.0, -0.8192D-5, 1.0)
\]

On output, D1, D2, X1, and X2 are overwritten.

Call Statement and Input:

\[
\begin{array}{cccc|c}
N & X & INCX & Y & INCY & PARAM \\
\hline
\end{array}
\]

\[
\text{CALL DROTM( 5, X, INCX, Y, INCY, PARAM)}
\]

\[
X = (1.0, 2.0, 3.0, 4.0, 5.0) \\
Y = (-1.0, -2.0, -3.0, -4.0, -5.0) \\
\text{PARAM} = \text{output from the call to DROTMG}
\]

Output:

\[
X = (4096.0, 8192.0, 12288.0, 16384.0, 20480.0) \\
Y = (1.0, 2.0, 3.0, 4.0, 5.0)
\]
SSCAL, DSCAL, CSCAL, ZSCAL, CSSCAL, and ZDSCAL (Multiply a Vector X by a Scalar and Store in the Vector X)

Purpose

These subprograms perform the following computation, using the scalar \( \alpha \) and the vector \( x \):

\[ x \leftarrow \alpha x \]

<table>
<thead>
<tr>
<th>( \alpha )</th>
<th>( x )</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SSCAL</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DSCAL</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision complex</td>
<td>CSCAL</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision complex</td>
<td>ZSCAL</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>Short-precision complex</td>
<td>CSSCAL</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision complex</td>
<td>ZDSCAL</td>
</tr>
</tbody>
</table>

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

**Fortran**

```fortran
CALL SSCAL | DSCAL | CSCAL | ZSCAL | CSSCAL | ZDSCAL (n, alpha, x, incx)
```

**C and C++**

```c
sscal | dscal | cscal | zscal | csscal | zdscal (n, alpha, x, incx);
```

**CBLAS**

```c

cblas_sscal | cblas_dscal | cblas_cscal | cblas_zscal | cblas_csscal | cblas_zdscal (n, alpha, x, incx);
```

On Entry

- \( n \) is the number of elements in vector \( x \). Specified as: an integer; \( n \geq 0 \).
- \( \alpha \) is the scalar \( \alpha \).
  Specified as: a number of the data type indicated in [Table 88](#).
- \( x \) is the vector \( x \) of length \( n \). Specified as: a one-dimensional array of (at least) length \( 1+(n-1)|\text{incx}| \), containing numbers of the data type indicated in [Table 88](#).
- \( \text{incx} \) is the stride for vector \( x \).
  Specified as: an integer. It can have any value.

On Return

- \( x \) is the vector \( x \) of length \( n \), containing the result of the computation \( \alpha x \).
  Returned as: a one-dimensional array, containing numbers of the data type indicated in [Table 88](#).
Notes

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 [93 on page 1368]. If \( n \) is 0, no computation is performed. For CSCAL, intermediate results are accumulated in long precision when the AltiVec or VSX unit is not used.

Error conditions

Computational Errors

None

Input-Argument Errors

\( n < 0 \)

Examples

Example 1

This example shows a vector, \( x \), with a stride of 1.

Call Statement and Input:

\[
\begin{align*}
N & \quad \text{ALPHA} & \quad X & \quad \text{INCX} \\
\mid & \quad \mid & \quad \mid & \quad \\
\text{CALL SSCAL} & (5, 2.0, X, 1) \\
\end{align*}
\]

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

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:

\[
\begin{align*}
N & \quad \text{ALPHA} & \quad X & \quad \text{INCX} \\
\mid & \quad \mid & \quad \mid & \quad \\
\text{CALL SSCAL} & (5, 2.0, X, 2) \\
\end{align*}
\]

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

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:

```
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:

```
CALL SSCAL( 1, 2.0, X, 0 )
```

$x$ = (1.0)

Output:

$x$ = (2.0)

**Example 5**

This example shows a scalar, $\alpha$, and a vector, $x$, containing complex numbers, where vector $x$ has a stride of 1.

Call Statement and Input:

```
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, $\alpha$, containing a real number, and a vector, $x$, containing complex numbers, where vector $x$ has a stride of 1.

Call Statement and Input:

```
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)

Purpose

These subprograms interchange the elements of vectors $x$ and $y$:

$$y \leftrightarrow x$$

Table 89. Data Types

<table>
<thead>
<tr>
<th>$x, y$</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SSWAP</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DSWAP</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CSWAP</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZSWAP</td>
</tr>
</tbody>
</table>

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SSWAP</th>
<th>DSWAP</th>
<th>CSWAP</th>
<th>ZSWAP $(n, x, \text{incx}, y, \text{incy})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>sswap</td>
<td>dswap</td>
<td>cswap</td>
<td>zswap $(n, x, \text{incx}, y, \text{incy})$</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_sswap</td>
<td>cblas_dswap</td>
<td>cblas_cswap</td>
<td>cblas_zswap $(n, x, \text{incx}, y, \text{incy})$</td>
</tr>
</tbody>
</table>

On Entry

$n$ is the number of elements in vectors $x$ and $y$.

Specified as: an integer; $n \geq 0$.

$x$ is the vector $x$ of length $n$. Specified as: a one-dimensional array of (at least) length $1+(n-1)|\text{incx}|$, containing numbers of the data type indicated in Table 89.

$\text{incx}$

is the stride for vector $x$.

Specified as: an integer. It can have any value.

$y$ is the vector $y$ of length $n$. Specified as: a one-dimensional array of (at least) length $1+(n-1)|\text{incy}|$, containing numbers of the data type indicated in Table 89.

$\text{incy}$

is the stride for vector $y$.

Specified as: an 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 89.
$y$ 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 89 on page 293.

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 75.

Function
The elements of vectors $x$ and $y$ are interchanged as follows:

\[
\begin{bmatrix}
y_1 \\
\vdots \\
y_n \\
\end{bmatrix}
\leftrightarrow
\begin{bmatrix}
x_1 \\
\vdots \\
x_n \\
\end{bmatrix}
\]

See reference 93 on page 1368. If $n$ is 0, no elements are interchanged.

Error conditions
Computational Errors
None
Input-Argument Errors
$n < 0$

Examples
Example 1
This example shows vectors $x$ and $y$ with positive strides.

Call Statement and Input:

```
N X INCX Y INCY
5 X 1 Y 2
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
5 X 1 Y -1
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:

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:

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 \( z \))

**Purpose**

These subprograms perform the following computation, using vectors \( x, y, \) and \( z \):

\[
z + x + y
\]

**Table 90. Data Types**

<table>
<thead>
<tr>
<th>Type</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SVEA</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DVEA</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CVEA</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZVEA</td>
</tr>
</tbody>
</table>

**Note:** On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

**Syntax**

**Fortran**

\[
\text{CALL SVEA | DVEA | CVEA | ZVEA (n, x, incx, y, incy, z, incz)}
\]

**C and C++**

\[
svea | dvea | cvea | zvea (n, x, incx, y, incy, z, incz);
\]

**CBLAS**

\[
cblas_svea | cblas_dvea | cblas_cvea | cblas_zvea (n, x, incx, y, incy, z, incz);
\]

**On Entry**

\( n \) is the number of elements in vectors \( x, y, \) and \( z \).

Specified as: an integer; \( n \geq 0 \).

\( x \) is the vector \( x \) of length \( n \). Specified as: a one-dimensional array of (at least) length \( 1+(n-1) \mid incx \mid \), containing numbers of the data type indicated in [Table 90](#).

\( incx \)

is the stride for vector \( x \).

Specified as: an integer. It can have any value.

\( y \) is the vector \( y \) of length \( n \). Specified as: a one-dimensional array of (at least) length \( 1+(n-1) \mid incy \mid \), containing numbers of the data type indicated in [Table 90](#).

\( incy \)

is the stride for vector \( y \).

Specified as: an integer. It can have any value.

\( z \) See [On Return](#).

\( incz \)

is the stride for vector \( z \).

Specified as: an integer. It can have any value.

**On Return**
**Notes**

1. If you specify the same vector for \( x \) and \( z \), then \( \text{incx} \) and \( \text{incz} \) 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 75.

**Function**

The computation is expressed as follows:

\[
\begin{bmatrix}
  z_1 \\
  \vdots \\
  z_n \\
\end{bmatrix} \leftarrow \begin{bmatrix}
  x_1 \\
  \vdots \\
  x_n \\
\end{bmatrix} + \begin{bmatrix}
  y_1 \\
  \vdots \\
  y_n \\
\end{bmatrix}
\]

If \( n \) is 0, no computation is performed.

**Error conditions**

**Computational Errors**

None

**Input-Argument Errors**

\( n < 0 \)

**Examples**

**Example 1**

This example shows vectors \( x, y, \) and \( z \), with positive strides.

Call Statement and Input:

```plaintext
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:
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:

\[
\begin{array}{c|cccc|cccc}
N & X & INCX & Y & INCY & Z & INCZ \\
\hline
5 & X & 1 & Y & -1 & Z & 2 \\
\end{array}
\]

\[
\begin{align*}
X &= (1.0, 2.0, 3.0, 4.0, 5.0) \\
Y &= (5.0, 4.0, 3.0, 2.0, 1.0)
\end{align*}
\]

Output:

\[
Z = (2.0, 4.0, 6.0, 8.0, 10.0)
\]

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:

\[
\begin{array}{c|cccc|cccc}
N & X & INCX & Y & INCY & Z & INCZ \\
\hline
5 & X & 0 & Y & 1 & Z & 1 \\
\end{array}
\]

\[
\begin{align*}
X &= (1.0) \\
Y &= (5.0, 4.0, 3.0, 2.0, 1.0)
\end{align*}
\]

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:

\[
\begin{array}{c|cccc|cccc}
N & X & INCX & Y & INCY & Z & INCZ \\
\hline
5 & X & 1 & Y & 0 & Z & 0 \\
\end{array}
\]

\[
\begin{align*}
X &= (1.0, 2.0, 3.0, 4.0, 5.0) \\
Y &= (5.0)
\end{align*}
\]

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.
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:

\[
\text{CALL SVEA(} \, 5 \, , \, x \, , \, 0 \, , \, y \, , \, -1 \, , \, z \, , \, 0 \, )
\]

\[
\begin{align*}
x & = (1.0) \\
y & = (5.0, 4.0, 3.0, 2.0, 1.0)
\end{align*}
\]

Output:

\[
\begin{align*}
z & = (6.0)
\end{align*}
\]

Example 8

This example shows vectors \( x \) and \( y \), containing complex numbers and having positive strides.

Call Statement and Input:

\[
\text{CALL CVEA(} \, 3 \, , \, x \, , \, 1 \, , \, y \, , \, 2 \, , \, z \, , \, 1 \, )
\]

\[
\begin{align*}
x & = ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0)) \\
y & = ((7.0, 8.0), \ldots, (9.0, 10.0), \ldots, (11.0, 12.0))
\end{align*}
\]

Output:

\[
\begin{align*}
z & = ((8.0, 10.0), (12.0, 14.0), (16.0, 18.0))
\end{align*}
\]
SVES, DVES, CVES, and ZVES (Subtract a Vector Y from a Vector X and Store in a Vector Z)

Purpose

These subprograms perform the following computation, using vectors \( x, y, \) and \( z \):

\[ z \leftarrow x - y \]

Table 91. Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SVES</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DVES</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CVES</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZVES</td>
</tr>
</tbody>
</table>

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SVES</td>
</tr>
<tr>
<td>C and C++</td>
<td>sves</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_sves</td>
</tr>
</tbody>
</table>

On Entry

\( n \) is the number of elements in vectors \( x, y, \) and \( z \).

Specified as: an integer; \( n \geq 0 \).

\( x \) is the vector \( x \) of length \( n \). Specified as: a one-dimensional array of (at least) length \( 1+(n-1)\lvert \text{incx} \rvert \), containing numbers of the data type indicated in Table 91.

\( \text{incx} \)

is the stride for vector \( x \).

Specified as: an integer. It can have any value.

\( y \) is the vector \( y \) of length \( n \). Specified as: a one-dimensional array of (at least) length \( 1+(n-1)\lvert \text{incy} \rvert \), containing numbers of the data type indicated in Table 91.

\( \text{incy} \)

is the stride for vector \( y \).

Specified as: an integer. It can have any value.

\( z \) See On Return

\( \text{incz} \)

is the stride for vector \( z \).

Specified as: an integer. It can have any value.

On Return
**z** is the vector of length \( n \), containing the result of the computation. Returned as: a one-dimensional array of \((1+(n-1)|incz|)\), containing numbers of the data type indicated in Table 91 on page 300.

**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 75.

**Function**

The computation is expressed as follows:

\[
\begin{bmatrix}
z_1 \\
\vdots \\
z_n
\end{bmatrix} \leftarrow
\begin{bmatrix}
x_1 \\
\vdots \\
x_n
\end{bmatrix} - 
\begin{bmatrix}
y_1 \\
\vdots \\
y_n
\end{bmatrix}
\]

If \( n \) is 0, no computation is performed.

**Error conditions**

**Computational Errors**
None

**Input-Argument Errors**
\( n < 0 \)

**Examples**

**Example 1**

This example shows vectors \( x \), \( y \), and \( z \), with positive strides.

**Call Statement and Input:**

```fortran
CALL SVES( 5 , X , 1 , Y , Z , Z , 1 )

X = (1.0, 2.0, 3.0, 4.0, 5.0)
Y = (1.0, 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:**
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:
\[ Z = (0.0, 0.0, 0.0, 0.0, 0.0, 0.0) \]

**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:
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:
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:
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.
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:
\[
\text{CALL SVES( 5, X, 0, Y, -1, Z, 0 )}
\]
\[
X = (1.0)
\]
\[
Y = (5.0, 4.0, 3.0, 2.0, 1.0)
\]

Output:
\[
Z = (-4.0)
\]

Example 8
This example shows vectors \( x \) and \( y \), containing complex numbers and having positive strides.

Call Statement and Input:
\[
\text{CALL CVES( 3, X, 1, Y, 2, Z, 1 )}
\]
\[
X = ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
\]
\[
Y = ((7.0, 8.0), .., (9.0, 10.0), .., (11.0, 12.0))
\]

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)

Purpose

These subprograms perform the following computation, using vectors x, y, and z:

\[ z \leftarrow x^T y \]

Table 92. Data Types

<table>
<thead>
<tr>
<th>x, y, z</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SVEM</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DVEM</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CVEM</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZVEM</td>
</tr>
</tbody>
</table>

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SVEM</th>
<th>DVEM</th>
<th>CVEM</th>
<th>ZVEM (n, x, incx, y, incy, z, incz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>svem</td>
<td>dvem</td>
<td>cvem</td>
<td>zvem (n, x, incx, y, incy, z, incz);</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_svem</td>
<td>cblas_dvem</td>
<td>cblas_cvem</td>
<td>cblas_zvem (n, x, incx, y, incy, z, incz);</td>
</tr>
</tbody>
</table>

On Entry

n
is the number of elements in vectors x, y, and z.
Specified as: an integer; \( n \geq 0 \).

x
is the vector x of length n. Specified as: a one-dimensional array of (at least) length \( 1+\left|n-1\right|\) \( \text{incx} \), containing numbers of the data type indicated in Table 92.

incx
is the stride for vector x.
Specified as: an integer. It can have any value.

y
is the vector y of length n. Specified as: a one-dimensional array of (at least) length \( 1+\left|n-1\right|\) \( \text{incy} \), containing numbers of the data type indicated in Table 92.

incy
is the stride for vector y.
Specified as: an integer. It can have any value.

z
See On Return

incz
is the stride for vector z.
Specified as: an integer. It can have any value.

On Return
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 92 on page 304.

**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 75.

**Function**

The computation is expressed as follows:

\[ z_i = x_i y_i \quad \text{for} \quad i = 1, n \]

If \( n \) is 0, no computation is performed. For CVEM, intermediate results are accumulated in long precision when the AltIVec or VSX unit is not used.

**Error conditions**

**Computational Errors**

None

**Input-Argument Errors**

\( n < 0 \)

**Examples**

**Example 1**

This example shows vectors \( x \), \( y \), and \( z \), with positive strides.

Call Statement and Input:

\[
\begin{align*}
N & \quad X \quad INCX \quad Y \quad INCY \quad Z \quad INCZ \\
& \quad | \quad | \quad | \quad | \quad | \quad |
\end{align*}
\]

CALL SVEM( 5, X, 1, Y, 2, Z, 1 )

\[
\begin{align*}
X & = (1.0, 2.0, 3.0, 4.0, 5.0) \\
Y & = (1.0, 1.0, 1.0, 1.0, 1.0)
\end{align*}
\]

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:

\[
\begin{align*}
N & \quad X \quad INCX \quad Y \quad INCY \quad Z \quad INCZ \\
& \quad | \quad | \quad | \quad | \quad | \quad |
\end{align*}
\]

CALL SVEM( 5, X, 1, Y, -1, Z, 2 )

\[
\begin{align*}
X & = (1.0, 2.0, 3.0, 4.0, 5.0) \\
Y & = (5.0, 4.0, 3.0, 2.0, 1.0)
\end{align*}
\]

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:

```
<table>
<thead>
<tr>
<th>N</th>
<th>X INCX</th>
<th>Y INCY</th>
<th>Z INCZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>X</td>
<td>0</td>
<td>Y</td>
</tr>
</tbody>
</table>
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:

```
<table>
<thead>
<tr>
<th>N</th>
<th>X INCX</th>
<th>Y INCY</th>
<th>Z INCZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>X</td>
<td>1</td>
<td>Y</td>
</tr>
</tbody>
</table>
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:

```
<table>
<thead>
<tr>
<th>N</th>
<th>X INCX</th>
<th>Y INCY</th>
<th>Z INCZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>X</td>
<td>1</td>
<td>Y</td>
</tr>
</tbody>
</table>
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:

```
<table>
<thead>
<tr>
<th>N</th>
<th>X INCX</th>
<th>Y INCY</th>
<th>Z INCZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>X</td>
<td>0</td>
<td>Y</td>
</tr>
</tbody>
</table>
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:

```
CALL SVEM( 1, X, 0, Y, 0, Z, 0 )
```

\[
\begin{align*}
X &= (1.0) \\
Y &= (5.0) \\
\end{align*}
\]

Output:

\[
\begin{align*}
Z &= (5.0) \\
\end{align*}
\]

Example 8
This example shows vectors \( x \) and \( y \), containing complex numbers and having positive strides.

Call Statement and Input:

```
CALL CVEM( 3, X, 1, Y, 2, Z, 1 )
```

\[
\begin{align*}
X &= ( (1.0, 2.0), (3.0, 4.0), (5.0, 6.0) ) \\
Y &= ( (7.0, 8.0), \ldots, (9.0, 10.0), \ldots, (11.0, 12.0) ) \\
\end{align*}
\]

Output:

\[
\begin{align*}
Z &= ( (-9.0, 22.0), (-13.0, 66.0), (-17.0, 126.0) ) \\
\end{align*}
\]
SYAX, DYAX, CYAX, ZYAX, CSYAX, and ZDYAX (Multiply a Vector X by a Scalar and Store in a Vector Y)

Purpose

These subprograms perform the following computation, using the scalar \( \alpha \) and vectors \( x \) and \( y \):

\[
y \leftarrow \alpha x
\]

Table 93. Data Types

<table>
<thead>
<tr>
<th>( \alpha )</th>
<th>( x, y )</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SYAX</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DYAX</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision complex</td>
<td>CYAX</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision complex</td>
<td>ZYAX</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>Short-precision complex</td>
<td>CSYAX</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision complex</td>
<td>ZDYAX</td>
</tr>
</tbody>
</table>

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C and C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL SYAX</td>
<td>DYAX</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_syax</td>
</tr>
</tbody>
</table>

On Entry

\( n \) is the number of elements in vector \( x \) and \( y \).

Specified as: an integer; \( n \geq 0 \).

\( \alpha \) is the scalar \( \alpha \).

Specified as: a number of the data type indicated in Table 93.

\( x \) is the vector \( x \) of length \( n \). Specified as: a one-dimensional array of (at least) length \( 1+(n-1)\mid \text{incx} \), containing numbers of the data type indicated in Table 93.

\( \text{incx} \) is the stride for vector \( x \).

Specified as: an integer. It can have any value.

\( y \) See On Return

\( \text{incy} \) is the stride for vector \( y \).

Specified as: an integer. It can have any value.
On Return

\( y \) is the vector of length \( n \), containing the result of the computation \( \alpha x \).

Returned as: a one-dimensional array of (at least) length \( 1+(n-1)|\text{incy}| \), containing numbers of the data type indicated in Table 93 on page 308.

Notes

1. If you specify the same vector for \( x \) and \( y \), then \text{incx} and \text{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 75.

Function

The computation is expressed as follows:

\[
\begin{bmatrix}
  y_1 \\
  \vdots \\
  y_n
\end{bmatrix}
\leftarrow \alpha
\begin{bmatrix}
  x_1 \\
  \vdots \\
  x_n
\end{bmatrix}
\]

See reference [93 on page 1368]. If \( n \) is 0, no computation is performed. For CYAX, intermediate results are accumulated in long precision when the AltIVec or VSX unit is not used.

Error conditions

Computational Errors

None

Input-Argument Errors

\( n < 0 \)

Examples

Example 1

This example shows vectors \( x \) and \( y \) with positive strides.

Call Statement and Input:

\[
\begin{align*}
\text{N ALPHA X INCX Y INCY} \\
\text{CALL SYAX( 5 , 2.0 , X , 1 , Y , 2 )}
\end{align*}
\]

\( x = (1.0, 2.0, 3.0, 4.0, 5.0) \)

Output:

\( y = (2.0, \ldots, 4.0, \ldots, 6.0, \ldots, 8.0, \ldots, 10.0) \)

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:
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:

\[
\text{CALL SYAX( 5, 2.0, X, 0, Y, 1 )}
\]

\[
X = (1.0)
\]

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:

\[
\text{CALL SYAX( 1, 2.0, X, 0, Y, 0 )}
\]

\[
X = (1.0)
\]

Output:
\[
Y = (2.0)
\]

**Example 5**
This example shows a scalar, \( \alpha \), and vectors \( x \) and \( y \), containing complex numbers, where both vectors have a stride of 1.

Call Statement and Input:

\[
\text{CALL CYAX( 3, \alpha, X, 1, Y, 1 )}
\]

\[
\alpha = (2.0, 3.0)
\]

\[
X = ((1.0, 2.0), (2.0, 0.0), (3.0, 5.0))
\]

Output:
\[
Y = ((-4.0, 7.0), (4.0, 6.0), (-9.0, 19.0))
\]

**Example 6**
This example shows a scalar, \( \alpha \), containing a real number, and vectors \( x \) and \( y \), containing complex numbers, where both vectors have a stride of 1.

Call Statement and Input:

\[
\text{CALL CSYAX( 3, \alpha, X, 1, Y, 1 )}
\]

\[
\alpha = (2.0, 3.0)
\]

\[
X = ((1.0, 2.0), (2.0, 0.0), (3.0, 5.0))
\]

Output:
\[ Y = ((2.0, 4.0), (4.0, 0.0), (6.0, 10.0)) \]
SZAXPY, DZAXPY, CZAXPY, and ZZAXPY (Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in a Vector Z)

Purpose

These subprograms perform the following computation, using the scalar \( \alpha \) and vectors \( x, y, \) and \( z \):

\[
z \leftarrow y + \alpha x
\]

Table 94. Data Types

<table>
<thead>
<tr>
<th>( \alpha, x, y, z )</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SZAXPY</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DZAXPY</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CZAXPY</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZZAXPY</td>
</tr>
</tbody>
</table>

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 32.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SZAXPY</th>
<th>DZAXPY</th>
<th>CZAXPY</th>
<th>ZZAXPY ((n, \alpha, x, \text{incx}, y, \text{incy}, z, \text{incz}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>szaxpy</td>
<td>dzaxpy</td>
<td>czaxpy</td>
<td>zzaxpy ((n, \alpha, x, \text{incx}, y, \text{incy}, z, \text{incz}));</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_szaxpy</td>
<td>cblas_dzaxpy</td>
<td>cblas_czaxpy</td>
<td>cblas_zzaxpy ((n, \alpha, x, \text{incx}, y, \text{incy}, z, \text{incz}));</td>
</tr>
</tbody>
</table>

On Entry

\( n \) is the number of elements in vectors \( x, y, \) and \( z \).

Specified as: an integer; \( n \geq 0 \).

\( \alpha \) is the scalar \( \alpha \).

Specified as: a number of the data type indicated in Table 94.

\( x \) is the vector \( x \) of length \( n \). Specified as: a one-dimensional array of (at least) length \( 1+(n-1)|\text{incx}| \), containing numbers of the data type indicated in Table 94.

\( \text{incx} \) is the stride for vector \( x \).

Specified as: an integer. It can have any value.

\( y \) is the vector \( y \) of length \( n \). Specified as: a one-dimensional array of (at least) length \( 1+(n-1)|\text{incy}| \), containing numbers of the data type indicated in Table 94.

\( \text{incy} \) is the stride for vector \( y \).

Specified as: an integer. It can have any value.

\( z \) See On Return
incz

is the stride for vector z.

Specified as: an 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 94 on page 312.

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 75.

Function

The computation is expressed as follows:

\[
\begin{bmatrix}
  z_1 \\
  \vdots \\
  z_n \\
\end{bmatrix}
\leftarrow
\begin{bmatrix}
  y_1 \\
  \vdots \\
  y_n \\
\end{bmatrix}
+ \alpha
\begin{bmatrix}
  x_1 \\
  \vdots \\
  x_n \\
\end{bmatrix}
\]

See reference [93 on page 1368]. If \( n \) is 0, no computation is performed. For CZAXPY, intermediate results are accumulated in long precision when the Altivec or VSX unit is not used.

Error conditions

Computational Errors

None

Input-Argument Errors

\( n < 0 \)

Examples

Example 1

This example shows vectors \( x \) and \( y \) with positive strides.

Call Statement and Input:

\[
\begin{array}{cccccccc}
N & ALPHA & X & INCX & Y & INCY & Z & INCZ \\
\hline
5 & 2.0 & X & 1 & Y & 2 & Z & 1 \\
\end{array}
\]

\[
\text{CALL SZAXPY(} \quad 5 \quad , \quad 2.0 \quad , \quad X \quad , \quad 1 \quad , \quad Y \quad , \quad 2 \quad , \quad Z \quad , \quad 1 \quad )
\]

\[
X = (1.0, 2.0, 3.0, 4.0, 5.0)
\]

\[
Y = (1.0, \ldots, 1.0, \ldots, 1.0, \ldots, 1.0, \ldots, 1.0)
\]

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:

\[
\begin{align*}
N & \quad \text{ALPHA} & \quad X & \quad \text{INCX} & \quad Y & \quad \text{INCY} & \quad Z & \quad \text{INCZ} \\
\text{CALL} & \quad \text{SZAXPY}( 5, 2.0, X, 1, Y, -1, Z, 2 )
\end{align*}
\]

\[
\begin{align*}
X & = (1.0, 2.0, 3.0, 4.0, 5.0) \\
Y & = (5.0, 4.0, 3.0, 2.0, 1.0)
\end{align*}
\]

Output:

\[
\begin{align*}
Z & = (3.0, 6.0, 9.0, 12.0, 15.0)
\end{align*}
\]

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:

\[
\begin{align*}
N & \quad \text{ALPHA} & \quad X & \quad \text{INCX} & \quad Y & \quad \text{INCY} & \quad Z & \quad \text{INCZ} \\
\text{CALL} & \quad \text{SZAXPY}( 5, 2.0, X, 0, Y, 1, Z, -1 )
\end{align*}
\]

\[
\begin{align*}
X & = (1.0) \\
Y & = (5.0, 4.0, 3.0, 2.0, 1.0) \\
Z & = (3.0, 4.0, 5.0, 6.0, 7.0)
\end{align*}
\]

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:

\[
\begin{align*}
N & \quad \text{ALPHA} & \quad X & \quad \text{INCX} & \quad Y & \quad \text{INCY} & \quad Z & \quad \text{INCZ} \\
\text{CALL} & \quad \text{SZAXPY}( 5, 2.0, X, 1, Y, 0, Z, 1 )
\end{align*}
\]

\[
\begin{align*}
X & = (1.0, 2.0, 3.0, 4.0, 5.0) \\
Y & = (5.0)
\end{align*}
\]

Output:

\[
\begin{align*}
Z & = (7.0, 9.0, 11.0, 13.0, 15.0)
\end{align*}
\]

Example 5
This example shows how \text{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:

\[
\begin{align*}
N & \quad \text{ALPHA} & \quad X & \quad \text{INCX} & \quad Y & \quad \text{INCY} & \quad Z & \quad \text{INCZ} \\
\text{CALL} & \quad \text{SZAXPY}( 1, 2.0, X, 0, Y, 0, Z, 0 )
\end{align*}
\]

\[
\begin{align*}
X & = (1.0) \\
Y & = (5.0)
\end{align*}
\]

Output:

\[
\begin{align*}
Z & = (7.0)
\end{align*}
\]

Example 6
This example shows vectors \( x \) and \( y \), containing complex numbers and having positive strides.
Call Statement and Input:

\[
\begin{array}{ccccccc}
N & \text{ALPHA} & X & \text{INX} & Y & \text{INCY} & Z & \text{INZ} \\
\end{array}
\]

\[
\text{CALL CZAXPY}(3, \text{ALPHA}, X, 1, Y, 2, Z, 1)
\]

\begin{align*}
\text{ALPHA} & = (2.0, 3.0) \\
X & = ((1.0, 2.0), (2.0, 0.0), (3.0, 5.0)) \\
Y & = ((1.0, 1.0), \ldots, (0.0, 2.0), \ldots, (5.0, 4.0))
\end{align*}

Output:

\[
Z = ((-3.0, 8.0), (4.0, 8.0), (-4.0, 23.0))
\]
Sparse Vector-Scalar Subprograms

This 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)

Purpose

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>
<thead>
<tr>
<th>Table 95. Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x, y)</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
<tr>
<td>Short-precision complex</td>
</tr>
<tr>
<td>Long-precision complex</td>
</tr>
</tbody>
</table>

Syntax

**Fortran**

```fortran
CALL SSCTR | DSCTR | CSCTR | ZSCTR (nz, x, indx, y)
```

**C and C++**

```c
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: an 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 95](#).
- \(indx\) 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 integers.

**On Return**

- \(y\) is the sparse vector \(y\), stored in full-vector storage mode, of (at least) length \(\max(INDX(i))\) for \(i = 1, nz\), 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 95](#).

**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 75.

3. For a description of how sparse vectors are stored, see “Sparse Vector” on page 80.

**Function**

The copy is expressed as follows:

$$y_{INDX(i)} = x_i \quad \text{for } i = 1, nz$$

where:

$x$ is a sparse vector, stored in compressed-vector storage mode.

$y$ is a sparse vector, stored in full-vector storage mode.

$INDX$ is the indices array for sparse vector $x$.

See reference [37 on page 1365]. If $nz$ is 0, no copy is performed.

**Error conditions**

**Computational Errors**

None

**Input-Argument Errors**

$nz < 0$

**Examples**

**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 \( \text{INDX} \) 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:**

```
CALL CSCTR( 3 , X , INDX , Y )
```

\[
X = ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
\]

\[
\text{INDX} = (4, 1, 3)
\]

**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)

**Purpose**

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>
<thead>
<tr>
<th>Table 96. Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x, y)</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
<tr>
<td>Short-precision complex</td>
</tr>
<tr>
<td>Long-precision complex</td>
</tr>
</tbody>
</table>

**Syntax**

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SGTHR</th>
<th>DGTHR</th>
<th>CGTHR</th>
<th>ZGTHR (nz, y, x, indx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>sgthr</td>
<td>dgthr</td>
<td>cgthr</td>
<td>zgthr (nz, y, x, indx);</td>
</tr>
</tbody>
</table>

**On Entry**

- \(nz\) is the number of elements in sparse vector \(x\), stored in compressed-vector storage mode. Specified as: an integer; \(nz \geq 0\).

- \(y\) is the sparse vector \(y\), stored in full-vector storage mode, of (at least) length \(\max(\text{INDX}(i))\) for \(i = 1, nz\), from which \(nz\) elements are copied from positions indicated by the indices array \(\text{INDX}\).

  Specified as: a one-dimensional array of (at least) length \(\max(\text{INDX}(i))\) for \(i = 1, nz\), containing numbers of the data type indicated in Table 96.

- \(x\) See On Return.

- \(\text{indx}\)

  is the array, referred to as \(\text{INDX}\), 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 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 \(\text{INDX}\).

  Returned as: a one-dimensional array of (at least) length \(nz\), containing numbers of the data type indicated in Table 96.

**Notes**

1. Vectors \(x\) and \(y\) must have no common elements; otherwise, results are unpredictable. See “Concepts” on page 75.
2. For a description of how sparse vectors are stored, see “Sparse Vector” on page 80.

Function

The copy is expressed as follows:

\[ x_i \leftarrow y_{INDX(i)} \quad \text{for } i = 1, nz \]

where:

- \( x \) is a sparse vector, stored in compressed-vector storage mode.
- \( y \) is a sparse vector, stored in full-vector storage mode.
- \( \text{INDX} \) is the indices array for sparse vector \( x \).

See reference [37 on page 1365]. If \( nz \) is 0, no copy is performed.

Error conditions

Computational Errors
None

Input-Argument Errors
\( nz < 0 \)

Examples

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 \( \text{INDX} \) are in ascending order.

Call Statement and Input:

\[
\begin{array}{|c|c|c|c|}
\hline
\text{NZ} & \text{Y} & \text{X} & \text{INDX} \\
\hline
\end{array}
\]

\[
\text{CALL} \quad \text{SGTHR}( 5, \text{Y}, \text{X}, \text{INDX} )
\]

\[
\begin{array}{c}
\text{Y} \\
\text{INDX}
\end{array} = \begin{array}{c}
(6.0, 2.0, 4.0, 7.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0) \\
(1, 3, 4, 7, 9)
\end{array}
\]

Output:

\[
\text{X} = (6.0, 4.0, 7.0, -2.0, 9.0)
\]

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 \( \text{INDX} \) 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:

\[
\begin{array}{|c|c|c|c|}
\hline
\text{NZ} & \text{Y} & \text{X} & \text{INDX} \\
\hline
\end{array}
\]

\[
\text{CALL} \quad \text{SGTHR}( 5, \text{Y}, \text{X}, \text{INDX} )
\]

\[
\begin{array}{c}
\text{Y} \\
\text{INDX}
\end{array} = \begin{array}{c}
(6.0, 2.0, 4.0, 7.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0) \\
(4, 3, 1, 10, 7)
\end{array}
\]

Output:

\[
\text{X} = (7.0, 4.0, 6.0, 0.0, -2.0)
\]
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)

**Purpose**

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 97. Data Types**

<table>
<thead>
<tr>
<th>Subprogram</th>
<th>( x ), ( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGTHRZ</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGTHRZ</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGTHRZ</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGTHRZ</td>
</tr>
</tbody>
</table>

**Syntax**

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SGTHRZ</th>
<th>DGTHRZ</th>
<th>CGTHRZ</th>
<th>ZGTHRZ (nz, y, x, indx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>sgthrz</td>
<td>dgthrz</td>
<td>cgthrz</td>
<td>zgthrz (nz, y, x, indx);</td>
</tr>
</tbody>
</table>

**On Entry**

\( nz \) is the number of elements in sparse vector \( x \), stored in compressed-vector storage mode. Specified as: an integer; \( nz \geq 0 \).

\( y \) is the sparse vector \( y \), stored in full-vector storage mode, of (at least) length \( \max(\text{INDX}(i)) \) for \( i = 1, nz \), from which \( nz \) elements are copied from positions indicated by the indices array \( \text{INDX} \).

Specified as: a one-dimensional array of (at least) length \( \max(\text{INDX}(i)) \) for \( i = 1, nz \), containing numbers of the data type indicated in Table 97.

\( x \) See **On Return**

\( \text{indx} \) is the array, referred to as \( \text{INDX} \), 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 integers.

**On Return**

\( y \) is the sparse vector \( y \), stored in full-vector storage mode, of (at least) length \( \max(\text{INDX}(i)) \) for \( i = 1, nz \), whose elements are set to zero at positions indicated by the indices array \( \text{INDX} \).

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 97.

\( x \) is the sparse vector \( x \), containing \( nz \) elements stored in compressed-vector storage mode in an array, referred to as \( \lambda \), into which are copied the elements of vector \( y \) from positions indicated by the indices array \( \text{INDX} \).
Returned as: a one-dimensional array of (at least) length \( nz \), containing numbers of the data type indicated in Table 97 on page 323.

Notes

1. Each value specified in array \( \text{INDX} \) 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 75.
3. For a description of how sparse vectors are stored, see “Sparse Vector” on page 80.

Function

The copy is expressed as follows:

\[
\begin{align*}
x_i & \leftarrow y_{\text{INDX}(i)} \\
y_{\text{INDX}(i)} & \leftarrow \begin{cases} 0.0 & \text{for } \text{SGTHRZ and } \text{DGTHRZ} \\ (0.0, 0.0) & \text{for } \text{CGTHRZ and } \text{ZGTHRZ} \end{cases} \\
\end{align*}
\]

where:

\( x \) is a sparse vector, stored in compressed-vector storage mode.
\( \text{INDX} \) is the indices array for sparse vector \( x \).
\( y \) is a sparse vector, stored in full-vector storage mode.

See reference [37 on page 1365]. If \( nz \) is 0, no computation is performed.

Error conditions

Computational Errors

None

Input-Argument Errors

\( nz < 0 \)

Examples

Example 1

This example shows how to use \text{SGTHRZ} to copy specified elements of a vector \( y \) into a sparse vector \( x \) of length 5, where the elements of array \( \text{INDX} \) are in ascending order.

Call Statement and Input:

\[
\begin{align*}
\text{NZ} & \quad \text{Y} & \quad \text{X} & \quad \text{INDX} \\
\text{CALL } \text{SGTHRZ}(5, \, \text{Y}, \, \text{X}, \, \text{INDX}) \\
\end{align*}
\]

\[
\begin{align*}
\text{Y} & \quad = (6.0, 2.0, 4.0, 7.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0) \\
\text{INDX} & \quad = (1, 3, 4, 7, 9) \\
\end{align*}
\]

Output:

\[
\begin{align*}
\text{Y} & \quad = (0.0, 2.0, 0.0, 0.0, 6.0, 10.0, 0.0, 8.0, 0.0, 0.0) \\
\text{X} & \quad = (6.0, 4.0, 7.0, -2.0, 9.0) \\
\end{align*}
\]

Example 2

This example shows how to use \text{SGTHRZ} to copy specified elements of a
vector \( y \) into a sparse vector \( x \) of length 5, where the elements of array \( \text{INDX} \) 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:

\[
\begin{align*}
\text{NZ} & \quad \text{Y} & \quad \text{X} & \quad \text{INDX} \\
\end{align*}
\]

\[
\text{CALL SGTHRZ( 5 , Y , X , INDX )}
\]

\[
\begin{align*}
\text{Y} & = (6.0, 2.0, 4.0, 7.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0) \\
\text{INDX} & = (4, 3, 1, 10, 7)
\end{align*}
\]

Output:

\[
\begin{align*}
\text{Y} & = (0.0, 2.0, 0.0, 0.0, 6.0, 10.0, 0.0, 8.0, 9.0, 0.0) \\
\text{X} & = (7.0, 4.0, 6.0, 0.0, -2.0)
\end{align*}
\]

**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 \( \text{INDX} \) are in random order.

Call Statement and Input:

\[
\begin{align*}
\text{NZ} & \quad \text{Y} & \quad \text{X} & \quad \text{INDX} \\
\end{align*}
\]

\[
\text{CALL CGTHRZ( 3 , Y , X , INDX )}
\]

\[
\begin{align*}
\text{Y} & = ((6.0, 5.0), (-2.0, 3.0), (15.0, 4.0), (9.0, 0.0)) \\
\text{INDX} & = (4, 1, 3)
\end{align*}
\]

Output:

\[
\begin{align*}
\text{Y} & = ((0.0, 0.0), (-2.0, 3.0), (0.0, 0.0), (0.0, 0.0)) \\
\text{X} & = ((9.0, 0.0), (6.0, 5.0), (15.0, 4.0))
\end{align*}
\]
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 \(y\))

Purpose

These subprograms multiply sparse vector \(x\), stored in compressed-vector storage mode, by scalar \(\alpha\), add it to sparse vector \(y\), stored in full-vector storage mode, and store the result in vector \(y\).

Table 98. Data Types

<table>
<thead>
<tr>
<th>(\alpha, x, y)</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SAXPYI</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DAXPYI</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CAXPYI</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZAXPYI</td>
</tr>
</tbody>
</table>

Syntax

Fortran          
CALL SAXPYI | DAXPYI | CAXPYI | ZAXPYI (nz, alpha, x, indx, y)

C and C++        
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: an integer; \(nz \geq 0\).

\(alpha\)

is the scalar \(\alpha\). Specified as: a number of the data type indicated in Table 98.

\(x\) 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 98.

\(indx\)

is the array, referred to as \(\text{INDX}\), 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 integers.

\(y\) is the sparse vector \(y\), stored in full-vector storage mode, of (at least) length \(\max(\text{INDEX}(i))\) for \(i = 1, nz\). Specified as: a one-dimensional array of (at least) length \(\max(\text{INDEX}(i))\) for \(i = 1, nz\), containing numbers of the data type indicated in Table 98.

On Return

\(y\) is the sparse vector \(y\), stored in full-vector storage mode, of (at least) length \(\max(\text{INDEX}(i))\) for \(i = 1, nz\) containing the results of the computation, stored at positions indicated by the indices array \(\text{INDX}\).

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 98.
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 75.
3. For a description of how sparse vectors are stored, see “Sparse Vector” on page 80.

Function

The computation is expressed as follows:

\[
y_{\text{INDX}(i)} \leftarrow y_{\text{INDX}(i)} + \alpha x_i \quad \text{for } i = 1, \text{nz}
\]

where:

- \( x \) is a sparse vector, stored in compressed-vector storage mode.
- \( \text{INDX} \) is the indices array for sparse vector \( x \).
- \( y \) is a sparse vector, stored in full-vector storage mode.

See reference [37 on page 1365]. If \( \alpha \) or \( \text{nz} \) is zero, no computation is performed. For \( \text{SAXPYI} \) and \( \text{CAXPYI} \), intermediate results are accumulated in long-precision.

Error conditions

Computational Errors
None

Input-Argument Errors
\( \text{nz} < 0 \)

Examples

Example 1
This example shows how to use \( \text{SAXPYI} \) to perform a computation using a sparse vector \( x \) of length 5, where the elements of array \( \text{INDX} \) are in ascending order.

Call Statement and Input:

```plaintext
NZ ALPHA X INDX Y
| | | | |
CALL SAXPYI( 5, 2.0, X, INDX, Y )
```

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

\( \text{INDX} = (1, 3, 4, 7, 10) \)

\( Y = (1.0, 5.0, 4.0, 3.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0) \)

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 \( \text{SAXPYI} \) to perform a computation using a sparse vector \( x \) of length 5, where the elements of array \( \text{INDX} \) are in random order.

Call Statement and Input:

```plaintext
NZ ALPHA X INDX Y
| | | | |
CALL SAXPYI( 5, 2.0, X, INDX, Y )
```
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 \( \text{INDX} \) are in random order.

Call Statement and Input:

\[
\begin{array}{cccc}
\text{NZ} & \text{ALPHA} & \text{X} & \text{INDX} & \text{Y} \\
\hline
\end{array}
\]

\[
\text{CALL CAXPYI(} 3 , \text{ALPHA}, \text{X}, \text{INDX}, \text{Y} \text{)}
\]

\[
\begin{array}{cccc}
\text{ALPHA} & \text{X} & \text{INDX} & \text{Y} \\
\hline
(2.0, 3.0) & ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0)) & (4, 1, 3) & ((6.0, 5.0), (-2.0, 3.0), (15.0, 4.0), (9.0, 0.0)) \\
\end{array}
\]

Output:

\[
\begin{array}{cccc}
\text{Y} \\
\hline
(0.0, 22.0), (-2.0, 3.0), (7.0, 31.0), (5.0, 7.0) \\
\end{array}
\]
**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)

**Purpose**

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 99. Data Types**

<table>
<thead>
<tr>
<th>( x, y, Result )</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SDOTI</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DDOTI</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CDOTUI</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZDOTUI</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CDOTCI</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZDOTCI</td>
</tr>
</tbody>
</table>

**Syntax**

<table>
<thead>
<tr>
<th>Fortran</th>
<th>SDOTI</th>
<th>DDOTI</th>
<th>CDOTUI</th>
<th>ZDOTUI</th>
<th>CDOTCI</th>
<th>ZDOTCI (nz, ( x, indx, y ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>sdoti</td>
<td>ddoti</td>
<td>cdotui</td>
<td>zdotui</td>
<td>cdotci</td>
<td>zdotci (nz, ( x, indx, y ))</td>
</tr>
</tbody>
</table>

**On Entry**

\( nz \) is the number of elements in sparse vector \( x \), stored in compressed-vector storage mode. Specified as: an 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 99.

\( indx \) 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 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 99.

**On Return**
Function value

is the result of the dot product computation.

Returned as: a number of the data type indicated in Table 99 on page 329

Notes

1. Declare this function in your program as returning a value of the data type indicated in Table 99 on page 329.

2. For a description of how sparse vectors are stored, see “Sparse Vector” on page 80.

Function

For SDOTI, DDOTI, CDOTUI, and ZDOTUI, the dot product computation is expressed as follows:

\[ \sum_{i=1}^{nz} x_i y_{INDX(i)} = x_1 y_{INDX(1)} + x_2 y_{INDX(2)} + \ldots + x_{nz} y_{INDX(nz)} \]

For CDOTCI and ZDOTCI, the dot product computation is expressed as follows:

\[ \sum_{i=1}^{nz} \bar{x}_i y_{INDX(i)} = \bar{x}_1 y_{INDX(1)} + \bar{x}_2 y_{INDX(2)} + \ldots + \bar{x}_{nz} y_{INDX(nz)} \]

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.
- \( y \) is a sparse vector, stored in full-vector storage mode.
- \( \text{INDX} \) is the indices array for sparse vector \( x \).

See reference [37 on page 1365]. 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
\( nz < 0 \)
Examples

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 $\text{INDX}$ are in ascending
order.

Function Reference and Input:

\[
\begin{align*}
\text{NZ} & \quad \text{X} \quad \text{INDX} \quad \text{Y} \\
\text{DOTT} & \quad \text{SDOTI}(5, \text{X}, \text{INDX}, \text{Y}) \\
X & \quad = (1.0, 2.0, 3.0, 4.0, 5.0) \\
\text{INDX} & \quad = (1, 3, 4, 7, 10) \\
Y & \quad = (1.0, 5.0, 4.0, 3.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0)
\end{align*}
\]

Output:
\[
\text{DOTT} = (1.0 + 8.0 + 9.0 - 8.0 + 0.0) = 10.0
\]

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 $\text{INDX}$ are in random
order.

Function Reference and Input:

\[
\begin{align*}
\text{NZ} & \quad \text{X} \quad \text{INDX} \quad \text{Y} \\
\text{DOTT} & \quad \text{SDOTI}(5, \text{X}, \text{INDX}, \text{Y}) \\
X & \quad = (1.0, 2.0, 3.0, 4.0, 5.0) \\
\text{INDX} & \quad = (4, 3, 1, 10, 7) \\
Y & \quad = (1.0, 5.0, 4.0, 3.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0)
\end{align*}
\]

Output:
\[
\text{DOTT} = (3.0 + 8.0 + 3.0 + 0.0 - 10.0) = 4.0
\]

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 $\text{INDX}$ are in ascending
order.

Function Reference and Input:

\[
\begin{align*}
\text{NZ} & \quad \text{X} \quad \text{INDX} \quad \text{Y} \\
\text{DOTT} & \quad \text{CDOTUI}(3, \text{X}, \text{INDX}, \text{Y}) \\
X & \quad = ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0)) \\
\text{INDX} & \quad = (1, 3, 4) \\
Y & \quad = ((6.0, 5.0), (-2.0, 3.0), (15.0, 4.0), (9.0, 0.0))
\end{align*}
\]

Output:
\[
\text{DOTT} = (70.0, 143.0)
\]

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 $\text{INDX}$ are in random order.

Function Reference and Input:

\[
\begin{align*}
\text{NZ} & \quad \text{X} \quad \text{INDX} \quad \text{Y} \\
\text{DOTT} & \quad \text{CDOTCI}(3, \text{X}, \text{INDX}, \text{Y})
\end{align*}
\]

Chapter 8. Linear Algebra Subprograms
\[ X = ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0)) \]
\[ \text{INDX} = (4, 1, 3) \]
\[ Y = ((6.0, 5.0), (-2.0, 3.0), (15.0, 4.0), (9.0, 0.0)) \]

Output:
\[ \text{DOTT} = (146.0, -97.0) \]
Matrix-Vector Subprograms

This 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)

Purpose

SGEMV and DGEMV compute the matrix-vector product for either a real general matrix or its transpose, using the scalars $\alpha$ and $\beta$, vectors $x$ and $y$, and matrix $A$ or its transpose:

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

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

CGEMV and ZGEMV compute the matrix-vector product for either a complex general matrix, its transpose, or its conjugate transpose, using the scalars $\alpha$ and $\beta$, vectors $x$ and $y$, and matrix $A$, its transpose, or its conjugate transpose:

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

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

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

SGEMX and DGEMX compute the matrix-vector product for a real general matrix, using the scalar $\alpha$, vectors $x$ and $y$, and matrix $A$:

$y \leftarrow y + \alpha Ax$

SGEMTX and DGEMTX compute the matrix-vector product for the transpose of a real general matrix, using the scalar $\alpha$, vectors $x$ and $y$, and the transpose of matrix $A$:

$y \leftarrow y + \alpha A^T x$

Table 100. Data Types

<table>
<thead>
<tr>
<th>$\alpha, \beta, x, y, A$</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGEMV, SGEMX, and SGEMTX</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGEMV, DGEMX, and DGEMTX</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGEMV</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGEMV</td>
</tr>
</tbody>
</table>

Note:

1. 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.

2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.
Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SGEMV</th>
<th>DGEMV</th>
<th>CGEMV</th>
<th>ZGEMV (transa, m, n, alpha, a, lda, x, incx, beta, y, incy)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>CALL SGEMX</td>
<td>DGEMX</td>
<td>SGEMTX</td>
<td>DGEMTX (m, n, alpha, a, lda, x, incx, y, incy)</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_sgemv</td>
<td>cblas_dgemv</td>
<td>cblas_cgemv</td>
<td>cblas_zgemv (cblas_layout, cblas_transa, m, n, alpha, a, lda, x, incx, beta, y, incy);</td>
</tr>
</tbody>
</table>

On Entry

- **cblas_layout**
  - indicates whether the input matrices are stored in row major or column major order, where:
    - If cblas_layout = CblasRowMajor, the matrices are stored in row major order.
    - If cblas_layout = CblasColMajor, the matrices are stored in column major order.
  - Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

- **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'.

- **cblas_transa**
  - indicates the form of matrix A to use in the computation, where:
    - If cblas_transa = CblasNoTrans, A is used in the computation.
    - If cblas_transa = CblasTrans, $A^T$ is used in the computation.
    - If cblas_transa = CblasConjTrans, $A^H$ is used in the computation.
  - Specified as: an object of enumerated type CBLAS_TRANSPOSE. It must be CblasNoTrans, CblasTrans, or CblasConjTrans.

- **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: an integer; $0 \leq m \leq lda$.

- **n**
  - is the number of columns in matrix A, and:
    - For SGEMV, DGEMV, CGEMV, and ZGEMV:
If \( \text{transa} = 'N' \), it is the length of vector \( x \).
If \( \text{transa} = 'T' \) 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: an integer; \( n \geq 0 \).

\textit{alpha}

is the scaling constant \( \alpha \).
Specified as: a number of the data type indicated in Table 100 on page 334.

\textit{a}

is the \( m \) by \( n \) matrix \( A \), where:
For SGEMV, DGEMV, CGEMV, and ZGEMV:

If \( \text{transa} = 'N' \), \( A \) is used in the computation.
If \( \text{transa} = 'T' \), \( A^T \) is used in the computation.
If \( \text{transa} = 'C' \), \( A^H \) is used in the computation.
For SGEMX and DGEMX, \( A \) is used in the computation.
For SGEMTX and DGEMTX, \( A^T \) is used in the computation.

\textbf{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: an \textit{lda} by (at least) \( n \) array, containing numbers of the data type indicated in Table 100 on page 334.

\textit{lda}

is the leading dimension of the array specified for \( a \).
Specified as: an integer; \( \textit{lda} > 0 \) and \( \textit{lda} \equiv m \).

\textit{x}

is the vector \( x \), where:
For SGEMV, DGEMV, CGEMV, and ZGEMV:

If \( \text{transa} = 'N' \), it has length \( n \).
If \( \text{transa} = 'T' \) 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 100 on page 334 where:
For SGEMV, DGEMV, CGEMV, and ZGEMV:

If \( \text{transa} = 'N' \), it must have at least \( 1+(n-1) \mid \text{incx} \mid \) elements.
If \( \text{transa} = 'T' \) or \'C'\), it must have at least \( 1+(m-1) \mid \text{incx} \mid \) elements.
For SGEMX and DGEMX, it must have at least \( 1+(n-1) \mid \text{incx} \mid \) elements.
For SGEMTX and DGEMTX, it must have at least \( 1+(m-1) \mid \text{incx} \mid \) elements.

\textit{beta}

is the scaling constant \( \beta \).
Specified as: a number of the data type indicated in Table 100 on page 334.

\textit{incx}

is the stride for vector \( x \).
Specified as: an integer; It can have any value.

\(y\) 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 100 on page 334, 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: an 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 100 on page 334.

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 [42 on page 1365] and [43 on page 1365].
4. Vector \(y\) must have no common elements with matrix \(A\) or vector \(x\); otherwise, results are unpredictable. See "Concepts" on page 75.

Function

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 when the AltiVec or VSX unit is not used. Occasionally, for performance reasons, these intermediate results are stored.

See references [42 on page 1365], [43 on page 1365], [46 on page 1366], [54 on page 1366], and [93 on page 1368]. No computation is performed if \( m \) or \( n \) is 0 or if \( \alpha \) is zero and \( \beta \) is one.

**General Matrix**

For SGEMV, DGEMV, CGEMV, and ZGEMV, the matrix-vector product for a general matrix:

\[
y = \beta y + \alpha Ax
\]

is expressed as follows:

\[
\begin{bmatrix}
y_1 \\
\vdots \\
y_m
\end{bmatrix}
\leftarrow \beta 
\begin{bmatrix}
y_1 \\
\vdots \\
y_m
\end{bmatrix}
+ \alpha 
\begin{bmatrix}
a_{11} & \cdots & a_{1n} \\
\vdots & \ddots & \vdots \\
a_{m1} & \cdots & a_{mn}
\end{bmatrix}
\begin{bmatrix}
x_1 \\
\vdots \\
x_n
\end{bmatrix}
\]

For SGEMX and DGEMX, the matrix-vector product for a real general matrix:

\[
y = \beta y + \alpha Ax
\]

is expressed as follows:

\[
\begin{bmatrix}
y_1 \\
\vdots \\
y_m
\end{bmatrix}
\leftarrow \beta 
\begin{bmatrix}
y_1 \\
\vdots \\
y_m
\end{bmatrix}
+ \alpha 
\begin{bmatrix}
a_{11} & \cdots & a_{1n} \\
\vdots & \ddots & \vdots \\
a_{m1} & \cdots & a_{mn}
\end{bmatrix}
\begin{bmatrix}
x_1 \\
\vdots \\
x_n
\end{bmatrix}
\]

In these expressions:

- \( y \) is a vector of length \( m \).
- \( \alpha \) is a scalar.
- \( \beta \) is a scalar.
- \( A \) is an \( m \) by \( n \) matrix.
- \( x \) 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:

\[
y = \beta y + \alpha A^T x
\]

is expressed as follows:
For SGEMTX and DGEMTX, the matrix-vector product for the transpose of a real general matrix:

\[ y \leftarrow y + \alpha A^T x \]

is expressed as follows:

\[
\begin{bmatrix}
  y_1 \\
  \vdots \\
  y_n
\end{bmatrix}
\leftarrow
\begin{bmatrix}
  \beta \\
  \vdots \\
  \beta
\end{bmatrix}
+ \alpha
\begin{bmatrix}
  a_{11} & \cdots & a_{m1} \\
  \vdots & \ddots & \vdots \\
  a_{1n} & \cdots & a_{mn}
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  \vdots \\
  x_m
\end{bmatrix}
\]

In these expressions:

- \( y \) is a vector of length \( n \).
- \( \alpha \) is a scalar.
- \( \beta \) 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:

\[
\begin{bmatrix}
  y_1 \\
  \vdots \\
  y_n
\end{bmatrix}
\leftarrow
\begin{bmatrix}
  \beta \\
  \vdots \\
  \beta
\end{bmatrix}
+ \alpha
\begin{bmatrix}
  \bar{a}_{11} & \cdots & \bar{a}_{m1} \\
  \vdots & \ddots & \vdots \\
  \bar{a}_{1n} & \cdots & \bar{a}_{mn}
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  \vdots \\
  x_m
\end{bmatrix}
\]

where:

- \( y \) is a vector of length \( n \).
- \( \alpha \) is a scalar.
- \( \beta \) 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.

Computational Errors
None

Input-Argument Errors
1.  
2.  
3.  
4.  
5.  
6.  
7.  
8.  

Examples

Example 1
This example shows the computation for \texttt{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 \( F2-F1+1 \geq 3 \). In this example, array \( A \) is declared as \( A(1:10,0:2) \).

Call Statement and Input:

\begin{verbatim}
CALL SGEMV( 'N', 4, 3, 1.0, A(1,0), 10, X, 1, 1.0, Y, 2 )
\end{verbatim}

\[
A = \begin{bmatrix}
1.0 & 2.0 & 3.0 \\
2.0 & 2.0 & 4.0 \\
3.0 & 2.0 & 2.0 \\
4.0 & 2.0 & 1.0 \\
\end{bmatrix}
\]

\[
X = (3.0, 2.0, 1.0)
Y = (4.0, , 5.0, , 2.0, , 3.0)
\]

Output:

\[
Y = (14.0, , 19.0, , 17.0, , 20.0)
\]

Example 2
This example shows the computation for \texttt{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:

\begin{verbatim}
CALL SGEMV( 'T', 4, 3, 1.0, A(-1,1), 10, X, 1, 2.0, Y, 2 )
\end{verbatim}

\[
A = \text{(same as input } A \text{ in Example 1)}
\]

\[
X = (3.0, 2.0, 1.0, 4.0)
Y = (1.0, , 2.0, , 3.0)
\]
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 )
```

$\begin{bmatrix}
(1.0, 2.0) & (3.0, 5.0) & (2.0, 0.0) \\
(2.0, 3.0) & (7.0, 9.0) & (4.0, 8.0) \\
(7.0, 4.0) & (1.0, 4.0) & (6.0, 0.0) \\
(8.0, 2.0) & (2.0, 5.0) & (8.0, 0.0)
\end{bmatrix}$

$A =
\begin{bmatrix}
(9.0, 1.0) & (3.0, 6.0) & (1.0, 0.0) \\
. & . & . \\
. & . & . \\
. & . & . \\
. & . & .
\end{bmatrix}$

$X = ((1.0, 2.0), (4.0, 0.0), (1.0, 1.0))$

$\begin{bmatrix}
(1.0, 0.0)
\end{bmatrix}$

$BETA = (0.0, 0.0)$

$Y = ((1.0, 2.0), (4.0, 0.0), (1.0, -1.0), (3.0, 4.0), (2.0, 0.0))$

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 $A$ is used in the computation. Because $\beta$ is zero, the result of the computation is $\alpha A^T x$

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 )
```

$\begin{bmatrix}
(1.0, 2.0) & (3.0, 5.0) & (2.0, 0.0) \\
(2.0, 3.0) & (7.0, 9.0) & (4.0, 8.0) \\
(7.0, 4.0) & (1.0, 4.0) & (6.0, 0.0) \\
(8.0, 2.0) & (2.0, 5.0) & (8.0, 0.0)
\end{bmatrix}$

$A =$

$\begin{bmatrix}
(1.0, 0.0)
\end{bmatrix}$

$X = ((1.0, 2.0), (4.0, 0.0), (1.0, 1.0))$

$\begin{bmatrix}
(0.0, 0.0)
\end{bmatrix}$

$BETA =$

$\begin{bmatrix}
(0.0, 0.0)
\end{bmatrix}$

$Y =$

$\begin{bmatrix}
(42.0, 67.0), (10.0, 87.0), (50.0, 74.0)
\end{bmatrix}$

Example 5
This example shows the computation for TRANSA equal to 'C', where the conjugate transpose of the complex general matrix $A$ is used in the computation.

Call Statement and Input:
TRANSMANALPHAALDAXINCBETAYINCY
CALLCGEMV( 'C', 5, 3, ALPHA, A, 10, X, 1, BETA, Y, 1)

\[
\begin{align*}
\text{ALPHA} & = (-1.0, 0.0) \\
A & = \text{(same as input } A \text{ in Example 3)} \\
X & = ((1.0, 2.0), (4.0, 0.0), (1.0, 1.0), (3.0, 4.0), (2.0, 0.0)) \\
\text{BETA} & = (1.0, 0.0) \\
Y & = ((1.0, 2.0), (4.0, 0.0), (1.0, -1.0))
\end{align*}
\]

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 \( F2-F1+1 \geq 3 \). For this example, array \( A \) is declared as \( A(1:10,0:2) \).

Call Statement and Input:
\[
\begin{align*}
\text{M} & \quad \text{N} \quad \text{ALPHA} \quad \text{A} \quad \text{LDA} \quad \text{X} \quad \text{INCX} \quad \text{Y} \quad \text{INCY} \\
\text{CALL SGER}(4, 3, 1.0, A(1,0), 10, X, 1, Y, 2)
\end{align*}
\]

\[
\begin{bmatrix}
1.0 & 2.0 & 3.0 \\
2.0 & 2.0 & 4.0 \\
3.0 & 2.0 & 2.0 \\
4.0 & 2.0 & 1.0
\end{bmatrix}
\]

\[
\begin{align*}
A & = \begin{bmatrix}
\vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots 
\end{bmatrix} \\
X & = (3.0, 2.0, 1.0) \\
Y & = (4.0, \ldots, 5.0, \ldots, 2.0, \ldots, 3.0)
\end{align*}
\]

Output:
\[
Y = (14.0, \ldots, 19.0, \ldots, 17.0, \ldots, 20.0)
\]

**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:
\[
\begin{align*}
\text{M} & \quad \text{N} \quad \text{ALPHA} \quad \text{A} \quad \text{LDA} \quad \text{X} \quad \text{INCX} \quad \text{Y} \quad \text{INCY} \\
\text{CALL SGER}(4, 3, 1.0, A(-1,1), 10, X, 1, Y, -2)
\end{align*}
\]

\[
A = \text{(same as input } A \text{ in Example 6)} \\
X = (3.0, 2.0, 1.0) \\
Y = (3.0, \ldots, 2.0, \ldots, 5.0, \ldots, 4.0)
\]

Output:
\[
Y = (20.0, \ldots, 17.0, \ldots, 19.0, \ldots, 14.0)
\]
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:

\[
\begin{array}{cccccccc}
M & N & ALPHA & A & LDA & X & INCX & Y & INCY \\
\end{array}
\]
\[
\text{CALL SGEMX( 4, 3, 1.0, A(5,1), 10, X, 1, Y, 1 )}
\]

\[
A = \begin{bmatrix}
1.0 & 2.0 & 3.0 \\
2.0 & 2.0 & 4.0 \\
3.0 & 2.0 & 2.0 \\
4.0 & 2.0 & 1.0 \\
\end{bmatrix}
\]

\[
X = (3.0, 2.0, 1.0) \\
Y = (4.0, 5.0, 2.0, 3.0)
\]

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 \( F2-F1+1 \geq 3 \). For this example, array \( A \) is declared as \( A(1:4,0:2) \).

Call Statement and Input:

\[
\begin{array}{cccccccc}
M & N & ALPHA & A & LDA & X & INCX & Y & INCY \\
\end{array}
\]
\[
\text{CALL SGEMX( 4, 3, 1.0, A(1,0), 4, X, 1, Y, 1 )}
\]

\[
A = \begin{bmatrix}
1.0 & 2.0 & 3.0 \\
2.0 & 2.0 & 4.0 \\
3.0 & 2.0 & 2.0 \\
4.0 & 2.0 & 1.0 \\
\end{bmatrix}
\]

\[
X = (3.0, 2.0, 1.0) \\
Y = (4.0, 5.0, 2.0, 3.0)
\]

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 \( F2-F1+1 \geq 3 \). For this example, array \( A \) is declared as \( A(1:4,0:2) \).

Call Statement and Input:

\[
\begin{array}{cccccccc}
M & N & ALPHA & A & LDA & X & INCX & Y & INCY \\
\end{array}
\]
\[
\text{CALL SGEMTX( 4, 3, 1.0, A(1,0), 4, X, 1, Y, 1 )}
\]
\[ A = \begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 2.0 & 2.0 & 4.0 \\ 3.0 & 2.0 & 2.0 \\ 4.0 & 2.0 & 1.0 \end{bmatrix} \]

\[
X = (3.0, 2.0, 1.0, 4.0) \\
Y = (1.0, 2.0, 3.0) \\
\]

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:

```
CALL SGEMTX( 4, 3, 2.0, A(-1,1), 4, X, 1, Y, 1 )
```

\[
A \text{ } = \text{(same as input } A \text{ in Example 10)} \\
X \text{ } = (3.0, 2.0, 1.0, 4.0) \\
Y \text{ } = (1.0, 2.0, 3.0) \\
\]

Output:
\[
Y \text{ } = (53.0, 42.0, 49.0) \\
\]
**SGER, DGER, CGERU, ZGERU, CGERC, and ZGERC (Rank-One Update of a General Matrix)**

**Purpose**

SGER, DGER, CGERU, and ZGERU compute the rank-one update of a general matrix, using the scalar $\alpha$, matrix $A$, vector $x$, and the transpose of vector $y$:

$$A \leftarrow A + \alpha xy^T$$

CGERC and ZGERC compute the rank-one update of a general matrix, using the scalar $\alpha$, matrix $A$, vector $x$, and the conjugate transpose of vector $y$:

$$A \leftarrow A + \alpha xy^H$$

**Table 101. Data Types**

<table>
<thead>
<tr>
<th>$\alpha$, $A$, $x$, $y$</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGER</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGER</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGERU and CGERC</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGERU and ZGERC</td>
</tr>
</tbody>
</table>

**Note:**
1. For compatibility with earlier releases of ESSL, you can use the names SGER1 and DGER1 for SGER and DGER, respectively.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 32.

**Syntax**

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SGER</th>
<th>DGER</th>
<th>CGERU</th>
<th>ZGERU</th>
<th>CGERC</th>
<th>ZGERC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$(m, n, \alpha, x, \text{incx}, y, \text{incy}, a, \text{lda})$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C and C++</th>
<th>sger</th>
<th>dger</th>
<th>cgeru</th>
<th>zgeru</th>
<th>cgerc</th>
<th>zgerc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$(m, n, \alpha, x, \text{incx}, y, \text{incy}, a, \text{lda})$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CBLAS</th>
<th>cblas_sger</th>
<th>cblas_dger</th>
<th>cblas_cgeru</th>
<th>cblas_zgeru</th>
<th>cblas_cgerc</th>
<th>cblas_zgerc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$(\text{cblas_layout}, m, n, \alpha, x, \text{incx}, y, \text{incy}, a, \text{lda})$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**On Entry**

$\text{cblas_layout}$ indicates whether the input and output matrices are stored in row major order or column major order, where:

- If $\text{cblas_layout} = \text{CblasRowMajor}$, the matrices are stored in row major order.
- If $\text{cblas_layout} = \text{CblasColMajor}$, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

$m$ is the number of rows in matrix $A$ and the number of elements in vector $x$.

Specified as: an integer; $0 \leq m \leq \text{lda}$.

$n$ is the number of columns in matrix $A$ and the number of elements in vector $y$. 

Chapter 8. Linear Algebra Subprograms 345
Specified as: an integer; \( n \geq 0 \).

\( \text{alpha} \)

is the scaling constant \( \alpha \).

Specified as: a number of the data type indicated in Table 101 on page 345.

\( x \)

is the vector \( x \) of length \( m \).

Specified as: a one-dimensional array of (at least) length \( 1+(m-1)|\text{incx}| \), containing numbers of the data type indicated in Table 101 on page 345.

\( \text{incx} \)

is the stride for vector \( x \).

Specified as: an integer. It can have any value.

\( y \)

is the vector \( y \) of length \( n \), whose transpose or conjugate transpose is used in the computation.

**Note:** No data should be moved to form \( y^T \) or \( y^H \); that is, the vector \( y \) should always be stored in its untransposed form.

Specified as: a one-dimensional array of (at least) length \( 1+(n-1)|\text{incy}| \), containing numbers of the data type indicated in Table 101 on page 345.

\( \text{incy} \)

is the stride for vector \( y \).

Specified as: an integer. It can have any value.

\( a \)

is the \( m \) by \( n \) matrix \( A \). Specified as: an \( \text{lda} \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 101 on page 345.

\( \text{lda} \)

is the size of the leading dimension of the array specified for \( a \).

Specified as: an integer; \( \text{lda} > 0 \) and \( \text{lda} \geq m \).

**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 101 on page 345.

**Notes**

1. In these subroutines, \( \text{incx} = 0 \) and \( \text{incy} = 0 \) are valid; however, the Level 2 BLAS standard considers \( \text{incx} = 0 \) and \( \text{incy} = 0 \) to be invalid. See references [42 on page 1365] and [43 on page 1365].

2. Matrix \( A \) can have no common elements with vectors \( x \) and \( y \); otherwise, results are unpredictable. See “Concepts” on page 75.

**Function**

SGER, DGER, CGERU, and ZGERU compute the rank-one update of a general matrix:

\[ A + \alpha xy^T \]

where:
A is an $m$ by $n$ matrix.
$\alpha$ is a scalar.
$x$ is a vector of length $m$.
$y^T$ is the transpose of vector $y$ of length $n$.

It is expressed as follows:

\[
\begin{bmatrix}
a_{11} & \cdots & a_{1n} \\
\vdots & \ddots & \vdots \\
a_{m1} & \cdots & a_{mn}
\end{bmatrix}
\leftarrow
\begin{bmatrix}
a_{11} & \cdots & a_{1n} \\
\vdots & \ddots & \vdots \\
a_{m1} & \cdots & a_{mn}
\end{bmatrix}
+ \alpha
\begin{bmatrix}
x_1 \\
\vdots \\
x_m
\end{bmatrix}
\begin{bmatrix}
y_1 \\
\vdots \\
y_n
\end{bmatrix}
\]

It can also be expressed as:

\[
\begin{bmatrix}
a_{11} & \cdots & a_{1n} \\
\vdots & \ddots & \vdots \\
a_{m1} & \cdots & a_{mn}
\end{bmatrix}
\leftarrow
\begin{bmatrix}
a_{11} + \alpha x_1 y_1^* & \cdots & a_{1n} + \alpha x_1 y_n^* \\
\vdots & \ddots & \vdots \\
a_{m1} + \alpha x_m y_1^* & \cdots & a_{mn} + \alpha x_m y_n^*
\end{bmatrix}
\]

CGERC and ZGERC compute a slightly different rank-one update of a general matrix:

\[A + \alpha xy^H\]

where:

$A$ is an $m$ by $n$ matrix.
$\alpha$ is a scalar.
$x$ is a vector of length $m$.
$y^H$ is the conjugate transpose of vector $y$ of length $n$.

It is expressed as follows:

\[
\begin{bmatrix}
a_{11} & \cdots & a_{1n} \\
\vdots & \ddots & \vdots \\
a_{m1} & \cdots & a_{mn}
\end{bmatrix}
\leftarrow
\begin{bmatrix}
a_{11} & \cdots & a_{1n} \\
\vdots & \ddots & \vdots \\
a_{m1} & \cdots & a_{mn}
\end{bmatrix}
+ \alpha
\begin{bmatrix}
x_1 \\
\vdots \\
x_m
\end{bmatrix}
\begin{bmatrix}
\bar{y}_1 \\
\vdots \\
\bar{y}_n
\end{bmatrix}
\]

It can also be expressed as:
See references [42 on page 1365], [43 on page 1365], and [43 on page 1365]. No computation is performed if \( m, n \), or \( \alpha \) is zero. For CGERU and CGERC, intermediate results are accumulated in long precision when the AltiVec or VSX unit is not used. For SGER, intermediate results are accumulated in long precision on some platforms when the AltiVec or VSX unit is not used.

Error conditions

Resource Errors:
Unable to allocate internal work area.

Computational Errors
None

Input-Argument Errors
1. \( \text{cblas_layout} \neq \text{CblasRowMajor or CblasColMajor} \)
2. \( m < 0 \)
3. \( n < 0 \)
4. \( lda \leq 0 \)
5. \( m > lda \)

Examples

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 \( F2-F1+1 \geq 3 \). For this example, array \( A \) is declared as \( A(1:10,0:2) \).

Call Statement and Input:

\[
\begin{array}{ccccccc}
M & N & ALPHA & X & INCX & Y & INCY & A & LDA \\
\hline
4 & 3 & 1.0 & X & 1 & Y & 2 & A(1,0) & 10 \\
\end{array}
\]

\[
\begin{align*}
X &= (3.0, 2.0, 1.0, 4.0) \\
Y &= (1.0, \ , 2.0, \ , 3.0)
\end{align*}
\]

\[
\begin{bmatrix}
1.0 & 2.0 & 3.0 \\
2.0 & 2.0 & 4.0 \\
3.0 & 2.0 & 2.0 \\
4.0 & 2.0 & 1.0 \\
\end{bmatrix}
\]

Output:
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:
```fortran
CALL SGER( 4, 3, 1.0, X, 1, Y, -2, A(-1,1), 10 )
```

\[
X = (3.0, 2.0, 1.0, 4.0) \\
Y = (3.0, . . , 2.0, . . , 1.0) \\
A = \text{(same as input } A \text{ in Example 1)}
\]

Output:
\[
A = \text{(same as input } A \text{ in Example 1)}
\]

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:
```fortran
CALL SGER( 4, 3, 1.0, X, 3, Y, 1, A(4,1), 10 )
```

\[
X = (3.0, . . , 2.0, . . , 1.0, . . , 4.0) \\
Y = (1.0, 2.0, 3.0) \\
A = \\
\begin{bmatrix}
  . & . & . \\
  . & . & . \\
  1.0 & 2.0 & 3.0 \\
  2.0 & 2.0 & 4.0 \\
  3.0 & 2.0 & 4.0 \\
  4.0 & 2.0 & 1.0 \\
  . & . & . \\
  . & . & . \\
  . & . & .
\end{bmatrix}
\]

Output:
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 \( F2-F1+1 \geq 3 \). For this example, array \( A \) is declared as \( A(1:4,0:2) \).

Call Statement and Input:

\[
\begin{array}{ccccccc}
\text{M} & \text{N} & \text{ALPHA} & \text{X INCX} & \text{Y INCY} & \text{A} & \text{LDA} \\
\end{array}
\]

\[
\text{CALL SGER}( 4, 3, 1.0, X, 1, Y, 1, A(1,0), 4 )
\]

\[
X = (3.0, 2.0, 1.0, 4.0) \]

\[
Y = (1.0, 2.0, 3.0)
\]

\[
A = \begin{bmatrix}
1.0 & 2.0 & 3.0 \\
2.0 & 2.0 & 4.0 \\
3.0 & 2.0 & 2.0 \\
4.0 & 2.0 & 1.0 \\
\end{bmatrix}
\]

Output:

\[
A = \begin{bmatrix}
4.0 & 8.0 & 12.0 \\
4.0 & 6.0 & 10.0 \\
4.0 & 4.0 & 5.0 \\
8.0 & 10.0 & 13.0 \\
\end{bmatrix}
\]

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:

\[
\begin{array}{ccccccc}
\text{M} & \text{N} & \text{ALPHA} & \text{X INCX} & \text{Y INCY} & \text{A} & \text{LDA} \\
\end{array}
\]

\[
\text{CALL SGER}( 4, 3, 2.0, X, 1, Y, 1, A(-1,1), 4 )
\]

\[
X = (3.0, 2.0, 1.0, 4.0) \]

\[
Y = (1.0, 2.0, 3.0)
\]

\[
A = (\text{same as input } A \text{ in Example 4})
\]

Output:

\[
A = \begin{bmatrix}
7.0 & 14.0 & 21.0 \\
6.0 & 10.0 & 16.0 \\
5.0 & 6.0 & 9.0 \\
12.0 & 18.0 & 25.0 \\
\end{bmatrix}
\]

Example 6
This example shows a rank-one update in which all data items contain complex numbers, and the transpose \( y^T \) is used in the computation. Matrix \( A \) is contained in a larger array, \( A \). The strides of vectors \( x \) and \( y \) 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:

```fortran
CALL CGERU(5, 3, ALPHA, X, 1, Y, 1, A(1,0), 10)
```

\[
\begin{align*}
\text{ALPHA} &= (1.0, 0.0) \\
X &= ((1.0, 2.0), (4.0, 0.0), (1.0, 1.0), (3.0, 4.0), (2.0, 0.0)) \\
Y &= ((1.0, 2.0), (4.0, 0.0), (1.0, -1.0))
\end{align*}
\]

\[
A = \begin{bmatrix}
(1.0, 2.0) & (3.0, 5.0) & (2.0, 0.0) \\
(2.0, 3.0) & (7.0, 9.0) & (4.0, 8.0) \\
(7.0, 4.0) & (1.0, 4.0) & (6.0, 0.0) \\
(8.0, 2.0) & (2.0, 5.0) & (8.0, 0.0) \\
(9.0, 1.0) & (3.0, 6.0) & (1.0, 0.0)
\end{bmatrix}
\]

Output:

\[
A = \begin{bmatrix}
(-2.0, 6.0) & (7.0, 13.0) & (5.0, 1.0) \\
(6.0, 11.0) & (23.0, 9.0) & (8.0, 4.0) \\
(6.0, 7.0) & (5.0, 8.0) & (8.0, 0.0) \\
(3.0, 12.0) & (14.0, 21.0) & (15.0, 1.0) \\
(11.0, 5.0) & (11.0, 6.0) & (3.0, -2.0)
\end{bmatrix}
\]

Example 7

This example shows a rank-one update in which all data items contain complex numbers, and the conjugate transpose $y^H$ is used in the computation. Matrix A is contained in a larger array A. The strides of vectors x and 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:

```fortran
CALL CGERC(5, 3, ALPHA, X, 1, Y, 1, A(1,0), 10)
```

\[
\begin{align*}
\text{ALPHA} &= (1.0, 0.0) \\
X &= ((1.0, 2.0), (4.0, 0.0), (1.0, 1.0), (3.0, 4.0), (2.0, 0.0)) \\
Y &= ((1.0, 2.0), (4.0, 0.0), (1.0, -1.0)) \\
A &= \text{(same as input A in Example 6)}
\end{align*}
\]

Output:

\[
\begin{bmatrix}
(6.0, 2.0) & (7.0, 13.0) & (1.0, 3.0) \\
(6.0, -5.0) & (23.0, 9.0) & (8.0, 12.0) \\
(10.0, 3.0) & (5.0, 8.0) & (6.0, 2.0)
\end{bmatrix}
\]
\[
A = \begin{bmatrix}
(19.0, 0.0) & (14.0, 21.0) & (7.0, 7.0) \\
(11.0, -3.0) & (11.0, 6.0) & (3.0, 2.0) \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot 
\end{bmatrix}
\]
SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV, SSLMX, and DSLMX (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Matrix)

Purpose

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 \( \alpha \) and \( \beta \), matrix \( A \), and vectors \( x \) and \( y \):

\[
y \leftarrow \beta y + \alpha Ax
\]

SSLMX and DSLMX compute the matrix-vector product for a real symmetric matrix, using the scalar \( \alpha \), matrix \( A \), and vectors \( x \) and \( y \):

\[
y \leftarrow \alpha Ax
\]

The following storage modes are used:

- For SSPMV, DSPMV, CHPMV, and ZHPMV, matrix \( A \) is stored in upper- or lower-packed storage mode.
- For SSYMV, DSYMV, CHEMV, and ZHEMV, matrix \( A \) is stored in upper or lower storage mode.
- For SSLMX and DSLMX, matrix \( A \) is stored in lower-packed storage mode.

Table 102. Data Types

<table>
<thead>
<tr>
<th>( \alpha, \beta, A, x, y )</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SSPMV, SSYMV, and SSLMX</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DSPMV, DSYMV, and DSLMX</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CHPMV and CHEMV</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZHPMV and ZHEMV</td>
</tr>
</tbody>
</table>

Note:

1. 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.

2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SSPMV</th>
<th>DSPMV</th>
<th>CHPMV</th>
<th>ZHPMV (uplo, n, alpha, ap, x, incx, beta, y, incy)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CALL SSYMV</td>
<td>DSYMV</td>
<td>CHEMV</td>
<td>ZHEMV (uplo, n, alpha, a, lda, x, incx, beta, y, incy)</td>
</tr>
<tr>
<td></td>
<td>CALL SSLMX</td>
<td>DSLMX</td>
<td>(n, alpha, ap, x, incx, y, incy)</td>
<td></td>
</tr>
<tr>
<td>C and C++</td>
<td>sspmv</td>
<td>dspmv</td>
<td>chpmv</td>
<td>zhpmv (uplo, n, alpha, ap, x, incx, beta, y, incy);</td>
</tr>
<tr>
<td></td>
<td>ssymv</td>
<td>dsymv</td>
<td>chemv</td>
<td>zhemv (uplo, n, alpha, a, lda, x, incx, beta, y, incy);</td>
</tr>
<tr>
<td></td>
<td>sslmx</td>
<td>dslmx</td>
<td>(n, alpha, ap, x, incx, y, incy);</td>
<td></td>
</tr>
</tbody>
</table>
CBLAS | cblas_sspmv | cblas_dspmv | cblas_chpmv | cblas_zhpmv (cblas_layout, cblas_uplo, n, alpha, ap, x, incx, beta, y, incy);
CBLAS | cblas_ssymv | cblas_dsymv | cblas_chemv | cblas_zhemv (cblas_layout, cblas_uplo, n, alpha, a, lda, x, incx, beta, y, incy);

### On Entry

**cblas_layout** indicates whether the input matrices are stored in row major order or column major order, where:
- If `cblas_layout` = CblasRowMajor, the matrices are stored in row major order.
- If `cblas_layout` = CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

**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'.

**cblas_uplo** indicates the storage mode used for matrix A, where:
- If `cblas_uplo` = CblasUpper, A is stored in upper-packed or upper storage mode.
- If `cblas_uplo` = CblasLower, A is stored in lower-packed or lower storage mode.

Specified as: an object of enumerated type CBLAS_UPLO. It must be CblasUpper or CblasLower.

**n** is the number of elements in vectors x and y and the order of matrix A.

Specified as: an integer; \( n \geq 0 \).

**alpha** is the scaling constant \( \alpha \).

Specified as: a number of the data type indicated in Table 102 on page 353.

**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 102 on page 353.

**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 102 on page 353

\( lda \)

is the leading dimension of the array specified for \( a \).

Specified as: an integer; \( lda > 0 \) and \( lda \geq n \).

\( x \)

is the vector \( x \) of length \( n \).

Specified as: a one-dimensional array of (at least) \( 1+(n-1)|incx| \), containing numbers of the data type indicated in Table 102 on page 353

\( incx \)

is the stride for vector \( x \).

Specified as: an 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 \( \beta \).

Specified as: a number of the data type indicated in Table 102 on page 353

\( y \)

is the vector \( y \) of length \( n \).

Specified as: a one-dimensional array of (at least) \( 1+(n-1)|incy| \), containing numbers of the data type indicated in Table 102 on page 353

\( incy \)

is the stride for vector \( y \).

Specified as: an 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 102 on page 353

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 75.
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 85. 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 90.
Function

These subroutines perform the calculations described. See references \[42\] on page 1365, \[43\] on page 1365, and \[93\] on page 1368.

For **SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, and ZHEMV**

If \( n \) is zero or if \( \alpha \) is zero and \( \beta \) is one, no computation is performed.

For **SSLMX and DSLMX**

If \( n \) or \( \alpha \) is zero, no computation is performed.

For **SSLMX, SSPMV, SSYMV, CHPMV, and CHEMV**

Intermediate results are accumulated in long precision when the AltiVec or VSX unit is not used. 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:

\[
y + \beta y + \alpha Ax
\]

where:

- \( y \) is a vector of length \( n \).
- \( \alpha \) is a scalar.
- \( \beta \) 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:

\[
\begin{bmatrix}
y_1 \\
\vdots \\
y_n \\
\end{bmatrix}
\leftarrow \beta 
\begin{bmatrix}
y_1 \\
\vdots \\
y_n \\
\end{bmatrix}
+ \alpha 
\begin{bmatrix}
a_{11} & \cdots & a_{1n} \\
\vdots & \ddots & \vdots \\
a_{n1} & \cdots & a_{nn} \\
\end{bmatrix}
\begin{bmatrix}
x_1 \\
\vdots \\
x_n \\
\end{bmatrix}
\]

For **SSLMX and DSLMX**

These subroutines compute the matrix-vector product for a real symmetric matrix stored in lower-packed storage mode:

\[
y + y + \alpha Ax
\]

where:

- \( y \) is a vector of length \( n \).
- \( \alpha \) is a scalar.
- \( A \) is a real symmetric matrix of order \( n \).
- \( x \) is a vector of length \( n \).

It is expressed as follows:
Error conditions

Resource Errors
Unable to allocate internal work area.

Computational Errors
None

Input-Argument Errors
1. cblas_layout ≠ CblasRowMajor or CblasColMajor
2. uplo ≠ 'L' or 'U'
3. cblas_uplo ≠ CblasLower or CblasUpper
4. n < 0
5. lda < n
6. lda ≤ 0
7. incx = 0
8. incy = 0

Examples

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:

\[
\begin{bmatrix}
  8.0 & 4.0 & 2.0 \\
  4.0 & 6.0 & 7.0 \\
  2.0 & 7.0 & 3.0 \\
\end{bmatrix}
\]

Call Statement and Input:

\[
\text{CALL SSPMV('L', 3, 1.0, AP, X, 1, 1.0, Y, 2)}
\]

\[
\begin{align*}
\text{AP} &= (8.0, 4.0, 2.0, 6.0, 7.0, 3.0) \\
\text{X} &= (3.0, 2.0, 1.0) \\
\text{Y} &= (5.0, . , 3.0, . , 2.0)
\end{align*}
\]

Output:
\[
\begin{align*}
\text{Y} &= (39.0, . , 34.0, . , 25.0)
\end{align*}
\]

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:

\[
\text{CALL SSPMV('U', 3, 1.0, AP, X, -2, 2.0, Y, 1)}
\]
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:

\[
\begin{pmatrix}
(1.0, 0.0) & (3.0, 5.0) & (2.0, -3.0) \\
(3.0, -5.0) & (7.0, 0.0) & (4.0, -8.0) \\
(2.0, 3.0) & (4.0, 8.0) & (6.0, 0.0)
\end{pmatrix}
\]

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:

```fortran
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:

```fortran
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:
```
CALL SSYMV( 'L', 3, 1.0, A, 3, X, 1, 1.0, Y, 2 )
```

\[
A = \begin{bmatrix}
8.0 & . & . \\
4.0 & 6.0 & . \\
2.0 & 7.0 & 3.0
\end{bmatrix}
\]

$X = (3.0, 2.0, 1.0)$

$Y = (5.0, . , 3.0, . , 2.0)$

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:
```
CALL SSYMV( 'U', 3, 1.0, A, 4, X, -2, 2.0, Y, 1 )
```

\[
A = \begin{bmatrix}
8.0 & 4.0 & 2.0 \\
. & 6.0 & 7.0 \\
. & . & 3.0 \\
. & . & .
\end{bmatrix}
\]

$X = (4.0, . , 2.0, . , 1.0)$

$Y = (6.0, 5.0, 4.0)$

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:
```
CALL CHEMV( 'L', 3, ALPHA, A, 3, X, 1, BETA, Y, 2 )
```

\[
\text{ALPHA} = (1.0, 0.0)
\]
\[ A = \begin{bmatrix} (1.0, .) & . & . \\ (3.0, -5.0) & (7.0, .) & . \\ (2.0, 3.0) & (4.0, 8.0) & (6.0, .) \end{bmatrix} \]

\[ X = ((1.0, 2.0), (4.0, 0.0), (3.0, 4.0)) \]
\[ \text{BETA} = (1.0, 0.0) \]
\[ Y = ((1.0, 0.0), ., (2.0, -1.0), ., (2.0, 1.0)) \]

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:
\[
\begin{array}{cccccccc}
\text{UPLO} & N & \text{ALPHA} & A & \text{LDA} & X & \text{INCX} & \text{BETA} & \text{Y} & \text{INCY} \\
\end{array}
\]
\text{CALL CHEMV('U', 3, ALPHA, A, 3, X, -2, BETA, Y, 2 )}
\[
\text{ALPHA} = (1.0, 0.0)
\]
\[
A = \begin{bmatrix}
(1.0, .) & (3.0, 5.0) & (2.0, -3.0) \\
. & (7.0, .) & (4.0, -8.0) \\
. & . & (6.0, .) \\
\end{bmatrix}
\]
\[
X = ((3.0, 4.0), ., (4.0, 0.0), ., (1.0, 2.0))
\]
\[ \text{BETA} = (0.0, 0.0) \]
\[ Y = \text{(not relevant)} \]

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:
\[
\begin{bmatrix}
8.0 & 4.0 & 2.0 \\
4.0 & 6.0 & 7.0 \\
2.0 & 7.0 & 3.0 \\
\end{bmatrix}
\]

Call Statement and Input:
\[
\begin{array}{cccccccc}
N & \text{ALPHA} & AP & X & \text{INCX} & Y & \text{INCY} \\
\end{array}
\]
\text{CALL SSLMX(3, 1.0, AP, X, 1, Y, 2 )}
\[
\text{AP} = (8.0, 4.0, 2.0, 6.0, 7.0, 3.0) \\
X = (3.0, 2.0, 1.0) \\
Y = (5.0, ., 3.0, ., 2.0) \\
\]

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)

Purpose

SSPR, DSPR, SSYR, DSYR, SSLR1, and DSLR1 compute the rank-one update of a real symmetric matrix, using the scalar \( \alpha \), matrix \( A \), vector \( x \), and its transpose \( x^T \):

\[
A \leftarrow A + \alpha xx^T
\]

CHPR, ZHPR, CHER, and ZHER compute the rank-one update of a complex Hermitian matrix, using the scalar \( \alpha \), matrix \( A \), vector \( x \), and its conjugate transpose \( x^H \):

\[
A \leftarrow A + \alpha xx^H
\]

The following storage modes are used:

- For SSPR, DSPR, CHPR, and ZHPR, matrix \( A \) is stored in upper- or lower-packed storage mode.
- For SSYR, DSYR, CHER, and ZHER, matrix \( A \) is stored in upper or lower storage mode.
- For SSLR1 and DSLR1, matrix \( A \) is stored in lower-packed storage mode.

<table>
<thead>
<tr>
<th>Table 103. Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A, x )</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
<tr>
<td>Short-precision complex</td>
</tr>
<tr>
<td>Long-precision complex</td>
</tr>
</tbody>
</table>

**Note:**

1. 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.

2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

**Fortran**

```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++**

```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);
```
On Entry

cblas_layout
indicates whether the input and output matrices are stored in row major order or column major order, where:

- If cblas_layout = CblasRowMajor, the matrices are stored in row major order.
- If cblas_layout = CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

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'.

cblas_uplo
indicates the storage mode used for matrix A, where:

If cblas_uplo = CblasUpper, A is stored in upper-packed or upper storage mode.
If cblas_uplo = CblasLower, A is stored in lower-packed or lower storage mode.
Specified as: an object of enumerated type CBLAS_UPLO. It must be CblasUpper or CblasLower.

n
is the number of elements in vector x and the order of matrix A.
Specified as: an integer; \( n \geq 0 \).

alpha
is the scaling constant \( \alpha \).
Specified as: a number of the data type indicated in Table 103 on page 362

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 103 on page 362

incx
is the stride for vector x.
Specified as: an 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 103 on page 362.

\( a \) 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 103 on page 362.

\( lda \) is the leading dimension of the array specified for \( a \).

Specified as: an 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 103 on page 362.

\( a \) 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 103 on page 362.

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 75.
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 85. 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 90.

Function

These subroutines perform the computations described. See references [42 on page 1365], [43 on page 1365], and [93 on page 1368].

Note: If \( n \) or \( \alpha \) is 0, no computation is performed.

For CHPR and CHER

Intermediate results are accumulated in long precision when the AltiVec or VSX unit is not used.
For SSPR, SSYR, and SSLR1
Intermediate results are accumulated in long precision on some platforms when the AltiVec or VSX unit is not used.

For SSPR, DSPR, SSYR, DSYR, SSLR1, and DSLR1
These subroutines compute the rank-one update of a real symmetric matrix:

\[ A + \alpha xx^T \]

where:

- \( A \) is a real symmetric matrix of order \( n \).
- \( \alpha \) is a scalar.
- \( x \) is a vector of length \( n \).
- \( x^T \) is the transpose of vector \( x \).

It is expressed as follows:

\[
\begin{bmatrix}
    a_{11} & \cdots & a_{1n} \\
    \vdots & \ddots & \vdots \\
    a_{n1} & \cdots & a_{nn}
\end{bmatrix}
\begin{bmatrix}
    \alpha x_1 \\
    \vdots \\
    a_n x_n
\end{bmatrix}
\]

For CHPR, ZHPR, CHER, and ZHER
These subroutines compute the rank-one update of a complex Hermitian matrix:

\[ A + \alpha xx^H \]

where:

- \( A \) is a complex Hermitian matrix of order \( n \).
- \( \alpha \) is a scalar.
- \( x \) is a vector of length \( n \).
- \( x^H \) is the conjugate transpose of vector \( x \).

It is expressed as follows:

\[
\begin{bmatrix}
    a_{11} & \cdots & a_{1n} \\
    \vdots & \ddots & \vdots \\
    a_{n1} & \cdots & a_{nn}
\end{bmatrix}
\begin{bmatrix}
    \alpha x_1 \\
    \vdots \\
    a_n x_n
\end{bmatrix}
\]

Error conditions

Computational Errors
None

Input-Argument Errors
1. \( cblas_layout \neq \text{CblasRowMajor} \text{ or } \text{CblasColMajor} \)
2. \( uplo \neq \text{`L'} \text{ or } \text{`U'} \)
3. \( cblas_uplo \neq \text{CblasLower} \text{ or } \text{CblasUpper} \)
4. \( n < 0 \)
5. \( \text{incx} = 0 \)
6. \( \text{lda} \leq 0 \)
7. \( \text{lda} < n \)

Examples

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:

\[
\begin{bmatrix}
  8.0 & 4.0 & 2.0 \\
  4.0 & 6.0 & 7.0 \\
  2.0 & 7.0 & 3.0 \\
\end{bmatrix}
\]

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 )
```
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:

```fortran
CALL CHPR('U', 3, 1.0, X, -2, AP)
```

$X = ((3.0, 4.0), \ldots, (4.0, 0.0), \ldots, (1.0, 2.0))$

$AP = ((1.0, \ldots), (3.0, -5.0), (2.0, 3.0), (7.0, \ldots), (4.0, 8.0), (6.0, \ldots))$

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 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:

```fortran
CALL SSYR('L', 3, 1.0, X, 1, A, 3)
```

$X = (3.0, 2.0, 1.0)$

$A = \begin{bmatrix}
8.0 & \ldots \\
4.0 & 6.0 \\
2.0 & 7.0 & 3.0
\end{bmatrix}$

Output:

$A = \begin{bmatrix}
17.0 & \ldots \\
10.0 & 10.0 \\
5.0 & 9.0 & 4.0
\end{bmatrix}$

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:
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:
```
CALL SSYR('U', 3, 1.0, X, -2, A, 4)
```

\[ X = (1.0, 2.0, 3.0) \]

\[ A = \begin{bmatrix}
8.0 & 4.0 & 2.0 \\
6.0 & 7.0 & \\
. & 3.0 & \\
\end{bmatrix} \]

**Output:**

```
A = \begin{bmatrix}
17.0 & 10.0 & 5.0 \\
10.0 & 9.0 & \\
. & 4.0 & \\
\end{bmatrix}
```

Example 7
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:
```
CALL CHER('L', 3, 1.0, X, 1, A, 3)
```

\[ X = ((1.0, 2.0), (4.0, 0.0), (3.0, 4.0)) \]

\[ A = \begin{bmatrix}
(1.0, ) & & \\
(3.0, -5.0) & (7.0, ) & \\
(2.0, 3.0) & (4.0, 8.0) & (6.0, )
\end{bmatrix} \]

**Output:**

```
A = \begin{bmatrix}
(6.0, 0.0) & & \\
(7.0, -13.0) & (23.0, 0.0) & \\
(13.0, 1.0) & (16.0, 24.0) & (31.0, 0.0)
\end{bmatrix}
```

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:
```
CALL CHER('U', 3, 1.0, X, -2, A, 3)
```

\[ X = ((3.0, 4.0), (4.0, 0.0), (1.0, 2.0)) \]
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:

\[
\begin{align*}
\text{CALL } & \text{ SSLR1( 3, 1.0, X, 1, AP )} \\
X & = (3.0, 2.0, 1.0) \\
AP & = (8.0, 4.0, 2.0, 6.0, 7.0, 3.0)
\end{align*}
\]

Output:

\[
\begin{align*}
AP & = (17.0, 10.0, 5.0, 10.0, 9.0, 4.0)
\end{align*}
\]
SSPR2, DSPR2, CHPR2, ZHPR2, SSYR2, DSYR2, CHER2, ZHER2, SSLR2, and DSLR2 (Rank-Two Update of a Real Symmetric or Complex Hermitian Matrix)

**Purpose**

SSPR2, DSPR2, SSYR2, DSYR2, SSLR2, and DSLR2 compute the rank-two update of a real symmetric matrix, using the scalar $\alpha$, matrix $A$, vectors $x$ and $y$, and their transposes $x^T$ and $y^T$:

$$ A \leftarrow A + \alpha xy^T + \alpha yx^T $$

CHPR2, ZHPR2, CHER2, and ZHER2, compute the rank-two update of a complex Hermitian matrix, using the scalar $\alpha$, matrix $A$, vectors $x$ and $y$, and their conjugate transposes $x^H$ and $y^H$:

$$ A \leftarrow A + \alpha xy^H + \alpha yx^H $$

The following storage modes are used:

- For SSPR2, DSPR2, CHPR2, and ZHPR2, matrix $A$ is stored in upper- or lower-packed storage mode.
- For SSYR2, DSYR2, CHER2, and ZHER2, matrix $A$ is stored in upper or lower storage mode.
- For SSLR2 and DSLR2, matrix $A$ is stored in lower-packed storage mode.

**Table 104. Data Types**

<table>
<thead>
<tr>
<th>$\alpha$, A, x, y</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SSPR2, SSYR2, and SSLR2</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DSPR2, DSYR2, and DSLR2</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CHPR2 and CHER2</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZHPR2 and ZHER2</td>
</tr>
</tbody>
</table>

**Note:**

1. 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.

2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "[Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL](#)

**Syntax**

<table>
<thead>
<tr>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL SSPR2</td>
</tr>
<tr>
<td>CALL SSYR2</td>
</tr>
<tr>
<td>CALL SSLR2</td>
</tr>
<tr>
<td>C and C++</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CBLAS</th>
<th>cblas_sspr2</th>
<th>cblas_dspr2</th>
<th>cblas_chpr2</th>
<th>cblas_zhpr2 (cblas_layout, cblas_uplo, n, alpha, x, incx, y, icy, ap);</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cblas_ssy2r</td>
<td>cblas_dsyr2</td>
<td>cblas_cher2</td>
<td>cblas_zher2 (cblas_layout, cblas_uplo, n, alpha, x, incx, y, icy, a, lda);</td>
</tr>
</tbody>
</table>

**On Entry**

**cblas_layout**
indicates whether the input and output matrices are stored in row major order or column major order, where:

- If `cblas_layout` = CblasRowMajor, the matrices are stored in row major order.
- If `cblas_layout` = CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

**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'.

**cblas_uplo**
indicates the storage mode used for matrix A, where:

- If `cblas_uplo` = CblasUpper, A is stored in upper-packed or upper storage mode.
- If `cblas_uplo` = CblasLower, A is stored in lower-packed or lower storage mode.

Specified as: an object of enumerated type CBLAS_UPLO. It must be CblasUpper or CblasLower.

**n**
is the number of elements in vectors x and y and the order of matrix A.

Specified as: an integer; \( n \geq 0 \).

**alpha**
is the scaling constant \( \alpha \).

Specified as: a number of the data type indicated in Table 104 on page 370.

**x**
is the vector x of length n.

Specified as: a one-dimensional array of (at least) length \( 1+(n-1) | \text{incx} | \), containing numbers of the data type indicated in Table 104 on page 370.

**incx**
is the stride for vector x.

Specified as: an 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) \( \text{length } 1+(n-1)|\text{incy}| \), containing numbers of the data type indicated in Table 104 on page 370

**incy**

is the stride for vector \( y \).

Specified as: an integer, where:

For SSPR2, DSPR2, CHPR2, ZHPR2, SSYR2, DSYR2, CHER2, and ZHER2, \( \text{incy} < 0 \) or \( \text{incy} > 0 \).

For SSLR2 and DSLR2, \( \text{incy} \) 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) \( \text{length } n(n+1)/2 \), containing numbers of the data type indicated in Table 104 on page 370

**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 104 on page 370

**lda**

is the leading dimension of the array specified for \( a \).

Specified as: an integer; \( lda > 0 \) and \( lda \equiv n \).

**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 104 on page 370

\( a \) 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 104 on page 370

**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 75.
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 \( \alpha \neq \text{zero} \), 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 85. 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 90.

Function

These subroutines perform the computation described. See references [42 on page 1365], [43 on page 1365], and [93 on page 1368]. If \( n \) or \( \alpha \) is zero, no computation is performed.

For SSPR2, SSYR2, SSLR2, CHPR2, and CHER2, intermediate results are accumulated in long precision when the AltiVec or VSX unit is not used.

**SSPR2, DSPR2, SSYR2, DSYR2, SSLR2, and DSLR2**

These subroutines compute the rank-two update of a real symmetric matrix:

\[
A \leftarrow A + \alpha xy^T + \alpha yx^T
\]

where:

- \( A \) is a real symmetric matrix of order \( n \).
- \( \alpha \) is a scalar.
- \( x \) is a vector of length \( n \).
- \( x^T \) is the transpose of vector \( x \).
- \( y \) is a vector of length \( n \).
- \( y^T \) is the transpose of vector \( y \).

It is expressed as follows:

\[
\begin{bmatrix}
ad_{11} & \ldots & a_{1n} \\
\vdots & \ddots & \vdots \\
ad_{n1} & \ldots & a_{nn}
\end{bmatrix}
\leftarrow
\begin{bmatrix}
ad_{11} & \ldots & a_{1n} \\
\vdots & \ddots & \vdots \\
ad_{n1} & \ldots & a_{nn}
\end{bmatrix}
+ \alpha
\begin{bmatrix}
x_1 \\
\vdots \\
x_n
\end{bmatrix}
\times
\begin{bmatrix}
y_1 \\
\vdots \\
y_n
\end{bmatrix}
\]

**CHPR2, ZHPR2, CHER2, and ZHER2**

These subroutines compute the rank-two update of a complex Hermitian matrix:

\[
A \leftarrow A + \alpha xy^H + \overline{\alpha y}x^H
\]

where:
$A$ is a complex Hermitian matrix of order $n$.
$\alpha$ is a scalar.
$x$ is a vector of length $n$.
$x^H$ is the conjugate transpose of vector $x$.
$y$ is a vector of length $n$.
$y^H$ is the conjugate transpose of vector $y$.

It is expressed as follows:

\[
\begin{bmatrix}
    a_{11} & \cdots & a_{1n} \\
    \vdots & \ddots & \vdots \\
    a_{n1} & \cdots & a_{nn}
\end{bmatrix}
\rightarrow
\begin{bmatrix}
    a_{11} & \cdots & a_{1n} \\
    \vdots & \ddots & \vdots \\
    a_{n1} & \cdots & a_{nn}
\end{bmatrix} + \alpha
\begin{bmatrix}
    x_1 \\
    \vdots \\
    x_n
\end{bmatrix}
\begin{bmatrix}
    y_1 \\
    \vdots \\
    y_n
\end{bmatrix}
\]

\[
+ \bar{\alpha}
\begin{bmatrix}
    \bar{x}_1 \\
    \vdots \\
    \bar{x}_n
\end{bmatrix}
\]

**Error conditions**

**Resource Errors**

Unable to allocate internal work area.

**Computational Errors**

None

**Input-Argument Errors**

1. $cblas_layout \neq$ CblasRowMajor or CblasColMajor
2. $uplo \neq$ 'L' or 'U'
3. $cblas_uplo \neq$ CblasLower or CblasUpper
4. $n < 0$
5. $incx = 0$
6. $incy = 0$
7. $lda \leq 0$
8. $lda < n$

**Examples**

**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:

\[
\begin{bmatrix}
    8.0 & 4.0 & 2.0 \\
    4.0 & 6.0 & 7.0 \\
    2.0 & 7.0 & 3.0
\end{bmatrix}
\]

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:
\[
\text{CALL SSPR2('U', 3, 1.0, X, -2, Y, 2, AP)}
\]
\[
X = (1.0, 2.0, 3.0) \\
Y = (5.0, 3.0, 2.0) \\
AP = (8.0, 4.0, 6.0, 2.0, 7.0, 3.0)
\]

Output:
\[
AP = (38.0, 23.0, 13.0, 18.0, 14.0, 7.0)
\]

**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:
\[
\begin{pmatrix}
(1.0, 0.0) & (3.0, 5.0) & (2.0, -3.0) \\
(3.0, -5.0) & (7.0, 0.0) & (4.0, -8.0) \\
(2.0, 3.0) & (4.0, 8.0) & (6.0, 0.0)
\end{pmatrix}
\]

**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 \( \alpha \neq 0 \), the imaginary parts of the diagonal elements are set to zero.

Call Statement and Input:
\[
\text{CALL CHPR2('L', 3, ALPHA, X, 1, Y, 2, AP)}
\]
\[
ALPHA = (1.0, 0.0) \\
X = ((1.0, 2.0), (4.0, 0.0), (3.0, 4.0)) \\
Y = ((1.0, 0.0), (2.0, -1.0), (2.0, 1.0)) \\
AP = ((1.0, 0.0), (3.0, -5.0), (2.0, 3.0), (7.0, 0.0), (4.0, 8.0), (6.0, 0.0))
\]

Output:
\[
AP = ((3.0, 0.0), (7.0, -10.0), (9.0, 0.0), (23.0, 0.0), (14.0, 23.0), (26.0, 0.0))
\]

**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 \( \alpha \neq 0 \), 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 )
```

**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 )
```

**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 )
```

**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 )
```

Output:

\[
\begin{bmatrix}
8.0 & . & . \\
4.0 & 6.0 & . \\
2.0 & 7.0 & 3.0
\end{bmatrix}
\]

**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.
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 \( \alpha \neq 0 \), the imaginary parts of the diagonal elements are set to zero.

Call Statement and Input:

```fortran
CALL CHER2( 'L', 3, ALPHA, X, 1, Y, 2, A, 3 )
```

\( \text{ALPHA} = (1.0, 0.0) \)

\( \text{X} = ((1.0, 2.0), (4.0, 0.0), (3.0, 4.0)) \)

\( \text{Y} = ((1.0, 0.0), \ldots, (2.0, -1.0), \ldots, (2.0, 1.0)) \)

\( A = \begin{pmatrix}
    (1.0, \ldots) & \ldots & \\
    (3.0, -5.0) & (7.0, \ldots) & \\
    (2.0, 3.0) & (4.0, 8.0) & (6.0, \ldots)
\end{pmatrix} \)

Output:

\( A = \begin{pmatrix}
    (3.0, 0.0) & \ldots & \\
    (7.0, -10.0) & (23.0, 0.0) & \\
    (9.0, 4.0) & (14.0, 23.0) & (26.0, 0.0)
\end{pmatrix} \)

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 \( \alpha \neq 0 \), the imaginary parts of the diagonal elements are set to zero.

Call Statement and Input:

```fortran
CALL CHER2( 'U', 3, ALPHA, X, -2, Y, 2, A, 3 )
```

\( \text{ALPHA} = (1.0, 0.0) \)

\( \text{X} = ((3.0, 4.0), \ldots, (4.8, 0.0), \ldots, (1.0, 2.0)) \)

\( \text{Y} = ((1.0, 0.0), \ldots, (2.0, -1.0), \ldots, (2.0, 1.0)) \)
A = \[
\begin{pmatrix}
(1.0, & .) & (3.0, & 5.0) & (2.0, & -3.0) \\

& . & (7.0, & .) & (4.0, & -8.0) \\
& & . & . & (6.0, & .) \\
\end{pmatrix}
\]

Output:

A = \[
\begin{pmatrix}
(3.0, & 0.0) & (7.0, & 10.0) & (9.0, & -4.0) \\

& . & (23.0, & 0.0) & (14.0, & -23.0) \\
& & . & . & (26.0, & 0.0) \\
\end{pmatrix}
\]

**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:**

```
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)

Purpose

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 $\alpha$ and $\beta$, vectors $x$ and $y$, and general band matrix $A$ or its transpose:

\[ y \leftarrow \beta y + \alpha Ax \]
\[ y \leftarrow \beta y + \alpha A^T x \]

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 $\alpha$ and $\beta$, vectors $x$ and $y$, and general band matrix $A$, its transpose, or its conjugate transpose:

\[ y \leftarrow \beta y + \alpha Ax \]
\[ y \leftarrow \beta y + \alpha A^T x \]
\[ y \leftarrow \beta y + \alpha A^H x \]

Table 105. Data Types

<table>
<thead>
<tr>
<th>$\alpha, \beta, x, y, A$</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGBMV</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGBMV</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGBMV</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGBMV</td>
</tr>
</tbody>
</table>

Syntax

For Fortran:

```fortran
CALL SGBMV | DGBMV | CGBMV | ZGBMV (transa, m, n, ml, mu, alpha, a, lda, x, incx, beta, y, incy)
```

For C and C++:

```c
sgbmv | dgbmv | cgbmv | zgbmv (transa, m, n, ml, mu, alpha, a, lda, x, incx, beta, y, incy);
```

For CBLAS:

```c
?gbmv (transa, m, n, ml, mu, alpha, a, lda, x, incx, beta, y, incy);
```

On Entry

- `cblas_layout` indicates whether the input matrices are stored in row major order or column major order, where:
  - If `cblas_layout` = CblasRowMajor, the matrices are stored in row major order.
  - If `cblas_layout` = CblasColMajor, the matrices are stored in column major order.

  Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

- `transa` indicates the form of matrix $A$ to use in the computation, where:
If \( \text{transa} = 'N' \), \( A \) is used in the computation.
If \( \text{transa} = 'T' \), \( A^T \) is used in the computation.
If \( \text{transa} = 'C' \), \( A^H \) is used in the computation.
Specified as: a single character. It must be \( 'N' \), \( 'T' \), or \( 'C' \).

\texttt{cblas\_transa}

indicates the form of matrix \( A \) to use in the computation, where:
If \( \text{cblas\_transa} = \text{CblasNoTrans} \), \( A \) is used in the computation.
If \( \text{cblas\_transa} = \text{CblasTrans} \), \( A^T \) is used in the computation.
If \( \text{cblas\_transa} = \text{CblasConjTrans} \), \( A^H \) is used in the computation.
Specified as: an object of enumerated type CBLAS\_TRANSPOSE. It must be CblasNoTrans, CblasTrans, or CblasConjTrans.

\( m \) is the number of rows in matrix \( A \), and:
If \( \text{transa} = 'N' \), it is the length of vector \( y \).
If \( \text{transa} = 'T' \) or \( 'C' \), it is the length of vector \( x \).
Specified as: an integer; \( m \geq 0 \).

\( n \) is the number of columns in matrix \( A \), and:
If \( \text{transa} = 'N' \), it is the length of vector \( x \).
If \( \text{transa} = 'T' \) or \( 'C' \), it is the length of vector \( y \).
Specified as: an integer; \( n \geq 0 \).

\( ml \) is the lower band width \( ml \) of the matrix \( A \).
Specified as: an integer; \( ml \geq 0 \).

\( mu \) is the upper band width \( mu \) of the matrix \( A \).
Specified as: an integer; \( mu \geq 0 \).

\texttt{alpha}

is the scaling constant \( \alpha \).
Specified as: a number of the data type indicated in Table 105 on page 379.

\( a \) 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 \( \text{transa} = 'N' \), \( A \) is used in the computation.
If \( \text{transa} = 'T' \), \( A^T \) is used in the computation.
If \( \text{transa} = 'C' \), \( A^H \) is used in the computation.

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 in BLAS-general-band storage mode.
Specified as: an \( lda \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 105 on page 379 where \( lda \geq ml+mu+1 \).

\( lda \)

is the leading dimension of the array specified for \( a \).
Specified as: an 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 105 on page 379 where:
If \( transa = 'N' \), it must have at least \( 1+(n-1)\cdot|incx| \) elements.
If \( transa = 'T' \) or \( 'C' \), it must have at least \( 1+(m-1)\cdot|incx| \) elements.

\( incx \)

is the stride for vector \( x \).
Specified as: an integer; \( incx > 0 \) or \( incx < 0 \).

\( beta \)

is the scaling constant \( \beta \).
Specified as: a number of the data type indicated in Table 105 on page 379.

\( y \)

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 105 on page 379 where:
If \( transa = 'N' \), it must have at least \( 1+(m-1)\cdot|incy| \) elements.
If \( transa = 'T' \) or \( 'C' \), it must have at least \( 1+(n-1)\cdot|incy| \) elements.

\( incy \)

is the stride for vector \( y \).
Specified as: an 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 105 on page 379.

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 75.
4. To achieve optimal performance, use \( lda = mu+ml+1 \).
5. For general band matrices, if you specify \( ml \geq m \) or \( mu \geq n \), 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 100. For an illustration of this technique, see Example 4.
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 100.
Function

The possible computations that can be performed by these subroutines are described. 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 \[42 on page 1365\], \[43 on page 1365\], \[46 on page 1366\], \[54 on page 1366\], and \[93 on page 1368\]. No computation is performed if \( m \) or \( n \) is 0 or if \( \alpha \) is zero and \( \beta \) is one.

General Band Matrix

For SGBMV, DGBMV, CGBMV, and ZGBMV, the matrix-vector product for a general band matrix is expressed as follows:

\[ y \leftarrow \beta y + \alpha Ax \]

where:

\( x \) is a vector of length \( n \).

\( y \) is a vector of length \( m \).

\( \alpha \) is a scalar.

\( \beta \) is a scalar.

\( A \) 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:

\[ y \leftarrow \beta y + \alpha A^T x \]

where:

\( x \) is a vector of length \( m \).

\( y \) is a vector of length \( n \).

\( \alpha \) is a scalar.

\( \beta \) is a scalar.

\( A^T \) is the transpose of an \( m \) by \( n \) general band matrix \( A \), 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:

\[ y \leftarrow \beta y + \alpha A^H x \]
where:

- $x$ is a vector of length $m$.
- $y$ is a vector of length $n$.
- $\alpha$ is a scalar.
- $\beta$ is a scalar.

$A^H$ is the conjugate transpose of an $m$ by $n$ general band matrix $A$ of order $n$, having a lower band width of $ml$ and an upper band width of $mu$.

**Error conditions**

**Resource Errors**

Unable to allocate internal work area

**Computational Errors**

None

**Input-Argument Errors**

1. $\text{cblas_layout} \neq \text{CblasRowMajor or CblasColMajor}$
2. $\text{transa} \neq \text{'N', 'T', or 'C'}$
3. $\text{cblas_transa} \neq \text{CblasNoTrans, CblasTrans, or CblasConjTrans}$
4. $m < 0$
5. $n < 0$
6. $ml < 0$
7. $mu < 0$
8. $lda \leq 0$
9. $lda < ml + mu + 1$
10. $\text{incx} = 0$
11. $\text{incy} = 0$

**Examples**

**Example 1**

This example shows how to use SGBMV to perform the computation $y + \beta y + \alpha Ax$, where $\text{TRANSA}$ is equal to ‘N’, and the following real general band matrix $A$ is used in the computation. Matrix $A$ is:

\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 0.0 \\
2.0 & 2.0 & 2.0 & 2.0 \\
3.0 & 3.0 & 3.0 & 3.0 \\
4.0 & 4.0 & 4.0 & 4.0 \\
0.0 & 5.0 & 5.0 & 5.0 \\
\end{bmatrix}
\]

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)
```

\[
A = \begin{bmatrix}
. & . & 1.0 & 2.0 \\
. & 1.0 & 2.0 & 3.0 \\
1.0 & 2.0 & 3.0 & 4.0 \\
2.0 & 3.0 & 4.0 & 5.0 \\
3.0 & 4.0 & 5.0 & . \\
4.0 & 5.0 & . & . \\
. & . & . & . \\
. & . & . & . \\
\end{bmatrix}
\]
Example 2
This example shows how to use SGBMV to perform the computation \( y = \beta y + \alpha A^T x \), where \( \text{TRANSA} \) 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:
```
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 
\( y = \beta y + \alpha A^H x \), where \( \text{TRANSA} \) is equal to 'C', and the complex conjugate of the following general band matrix \( A \) is used in the computation. Matrix \( A \) is:
```
\[
\begin{bmatrix}
(1.0, 1.0) & (1.0, 1.0) & (1.0, 1.0) & (0.0, 0.0) \\
(2.0, 2.0) & (2.0, 2.0) & (2.0, 2.0) & (2.0, 2.0) \\
(3.0, 3.0) & (3.0, 3.0) & (3.0, 3.0) & (3.0, 3.0) \\
(4.0, 4.0) & (4.0, 4.0) & (4.0, 4.0) & (4.0, 4.0) \\
(0.0, 0.0) & (5.0, 5.0) & (5.0, 5.0) & (0.0, 0.0)
\end{bmatrix}
\]
```

Call Statement and Input:
```
CALL CGBMV( 'C', 5, 4, 3, 2, ALPHAR, A, 8, X, 1, BETA, Y, 2 )
```
Output:
\[
A = \begin{bmatrix}
\cdot & \cdot & (1.0, 1.0) & (2.0, 2.0) \\
\cdot & (1.0, 1.0) & (2.0, 2.0) & (3.0, 3.0) \\
(1.0, 1.0) & (2.0, 2.0) & (3.0, 3.0) & (4.0, 4.0) \\
(2.0, 2.0) & (3.0, 3.0) & (4.0, 4.0) & (5.0, 5.0) \\
(3.0, 3.0) & (4.0, 4.0) & (5.0, 5.0) & . \\
(4.0, 4.0) & (5.0, 5.0) & . & . \\
\cdot & \cdot & \cdot & . \\
\cdot & \cdot & \cdot & . \\
\end{bmatrix}
\]

Example 4
This example shows how to use SGBMV to perform the computation 
\( y = \beta y + \alpha A x \), where \( ml \geq m \) and \( mu \geq n \), \( \text{TRANSA} \) is equal to 'N', and the following real general band matrix \( A \) is used in the computation. Matrix \( A \) is:
```
\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 \\
2.0 & 2.0 & 2.0 & 2.0
\end{bmatrix}
\]
```
Call Statement and Input:

\[
\begin{bmatrix}
3.0 & 3.0 & 3.0 & 3.0 & 3.0 \\
4.0 & 4.0 & 4.0 & 4.0 & 4.0
\end{bmatrix}
\]

\[
\begin{array}{cccccccc}
\ldots & \ldots & \ldots & \ldots & 1.0 \\
\ldots & \ldots & 1.0 & 2.0 \\
\ldots & 1.0 & 2.0 & 3.0 \\
\ldots & 1.0 & 2.0 & 3.0 & 4.0
\end{array}
\]

\begin{align*}
\mathbf{X} &= (1.0, 2.0, 3.0, 4.0, 5.0) \\
\mathbf{Y} &= (1.0, \ldots, 2.0, \ldots, 3.0, \ldots, 4.0, \ldots)
\end{align*}

Output:

\[
\begin{array}{cccccccc}
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots
\end{array}
\]

\[
\mathbf{Y} = (40.0, \ldots, 80.0, \ldots, 120.0, \ldots, 160.0, \ldots)
\]
SSBMV, DSBMV, CHBMV, and ZHBMV (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Band Matrix)

Purpose

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 $\alpha$ and $\beta$, vectors $x$ and $y$, and band matrix $A$:

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

Table 106. Data Types

<table>
<thead>
<tr>
<th>$\alpha$, $\beta$, $x$, $y$, $A$</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SSBMV</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DSBMV</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CHBMV</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZHBMV</td>
</tr>
</tbody>
</table>

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);
```

CBLAS

```
cblas/ssbmv | cblas/dsbmv | cblas/chbmv | cblas/zhbmv (cblas_layout, cblas_uplo, n, k, alpha, a, lda, x, incx, beta, y, incy);
```

On Entry

$cblas_layout$

indicates whether the input matrices are stored in row major order or column major order, where:

- If $cblas_layout = $CblasRowMajor, the matrices are stored in row major order.
- If $cblas_layout = $CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be $CblasRowMajor$ or $CblasColMajor$.

$uplo$

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'.

$cblas_uplo$

indicates the storage mode used for matrix $A$, where:

If $cblas_uplo = $CblasUpper, $A$ is stored in upper-band-packed storage mode.
If $cblas_uplo = $CblasLower, $A$ is stored in lower-band-packed storage mode.
Specified as: an object of enumerated type CBLAS_UPLO. It must be CblasUpper or CblasLower.

\( n \) is the order of matrix \( A \) and the number of elements in vectors \( x \) and \( y \).

Specified as: an integer; \( n \geq 0 \).

\( k \) is the half band width \( k \) of the matrix \( A \).

Specified as: an integer; \( k \geq 0 \).

\( \alpha \) is the scaling constant \( \alpha \).

Specified as: a number of the data type indicated in Table 106 on page 386.

\( a \) is the real symmetric or complex Hermitian band matrix \( A \) of order \( n \), having a half band width of \( k \), where:

If \( \text{uplo} = \text{'U'} \), \( A \) is stored in upper-band-packed storage mode.

If \( \text{uplo} = \text{'L'} \), \( A \) is stored in lower-band-packed storage mode.

Specified as: an \( \text{lda} \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 106 on page 386 where \( \text{lda} \geq k+1 \).

\( \text{lda} \) is the leading dimension of the array specified for \( a \).

Specified as: an integer; \( \text{lda} > 0 \) and \( \text{lda} \geq k+1 \).

\( x \) is the vector \( x \) of length \( n \).

Specified as: a one-dimensional array of (at least) length \( 1+(n-1)|\text{incx}| \), containing numbers of the data type indicated in Table 106 on page 386.

\( \text{incx} \) is the stride for vector \( x \).

Specified as: an integer; \( \text{incx} > 0 \) or \( \text{incx} < 0 \).

\( \beta \) is the scaling constant \( \beta \).

Specified as: a number of the data type indicated in Table 106 on page 386.

\( y \) 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 106 on page 386.

\( \text{incy} \) is the stride for vector \( y \).

Specified as: an integer; \( \text{incy} > 0 \) or \( \text{incy} < 0 \).

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 106 on page 386.

Notes

1. All subroutines accept lowercase letters for the \text{uplo} argument.
2. Vector \( y \) must have no common elements with matrix \( A \) or vector \( x \); otherwise, results are unpredictable. See "Concepts" on page 75.
3. To achieve optimal performance in these subroutines, use \( \text{lda} = k+1 \).
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.

6. For a description of how a real symmetric band matrix is stored, see “Upper-Band-Packed Storage Mode” on page 106 or “Lower-Band-Packed Storage Mode” on page 107. For a description of how a complex Hermitian band matrix is stored, see “Complex Hermitian Matrix” on page 90.

**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:

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

where:

- $x$ and $y$ are vectors of length $n$.
- $\alpha$ and $\beta$ are scalars.
- $A$ is an real symmetric or complex Hermitian band matrix of order $n$, having a half bandwidth of $k$.

For SSBMV and CHBMV, intermediate results are accumulated in long precision when the Altivector or VSX unit is not used. Occasionally, for performance reasons, these intermediate results are truncated to short precision and stored.

See references [42 on page 1365], [46 on page 1366], [54 on page 1366], and [93 on page 1368]. No computation is performed if $n$ is 0 or if $\alpha$ is zero and $\beta$ is one.

**Error conditions**

**Resource Errors**

Unable to allocate internal work area.

**Computational Errors**

None

**Input-Argument Errors**

1. $\text{cblas_layout} \neq \text{CblasRowMajor or CblasColMajor}$
2. $\text{uplo} \neq \text{'U' or 'L'}$
3. $\text{cblas_uplo} \neq \text{CblasLower or CblasUpper}$
4. $n < 0$
5. $k < 0$
6. $\text{lda} \leq 0$
7. $\text{lda} < k+1$
8. $\text{incx} = 0$
9. $\text{incy} = 0$
Examples

Example 1

This example shows how to use SSBMV to perform the matrix-vector product, where the real symmetric band matrix \( A \) of order 7 and half band width of 3 is stored in upper-band-packed storage mode. Matrix \( A \) is:

\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 2.0 & 2.0 & 2.0 & 2.0 & 0.0 & 0.0 \\
1.0 & 2.0 & 3.0 & 3.0 & 3.0 & 3.0 & 0.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 4.0 & 4.0 & 4.0 \\
0.0 & 2.0 & 3.0 & 4.0 & 5.0 & 5.0 & 5.0 \\
0.0 & 0.0 & 3.0 & 4.0 & 5.0 & 6.0 & 6.0 \\
0.0 & 0.0 & 0.0 & 4.0 & 5.0 & 6.0 & 7.0
\end{bmatrix}
\]

Call Statement and Input:

\[
\text{UPLO N K ALPHA A LDA X INCX BETA Y INCY}
\]

\[
\text{CALL SSBMV( 'U', 7, 3, 2.0, A, 5, X, 1, 10.0, Y, 2 )}
\]

\[
A = \begin{bmatrix}
. & . & . & 1.0 & 2.0 & 3.0 & 4.0 \\
. & . & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\
\end{bmatrix}
\]

\[
X = (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0)
\]

\[
Y = (1.0, , 2.0, , 3.0, , 4.0, , 5.0, , 6.0, , 7.0)
\]

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:

\[
\text{UPLO N K ALPHA A LDA X INCX BETA Y INCY}
\]

\[
\text{CALL CHBMV( 'L', 7, 3, ALPHA, A, 5, X, 1, BETA, Y, 2 )}
\]

\[
\text{ALPHA} = (2.0, 0.0)
\]

\[
\text{BETA} = (10.0, 0.0)
\]
Example 3

This example shows how to use SSBMV to perform the matrix-vector product, where \( n \geq k \). Matrix \( A \) is a real 5 by 5 symmetric band matrix with a half band width of 5, stored in upper-band-packed storage mode. Matrix \( A \) is:

\[
A = \begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 2.0 & 2.0 & 2.0 & 2.0 \\
1.0 & 2.0 & 3.0 & 3.0 & 3.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 4.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0
\end{bmatrix}
\]

Call Statement and Input:

\[
\text{UPLO N K ALPHA A LDA X INCX BETA Y INCY}
\]

\[
\text{CALL SSBMV( 'U', 5, 5, 2.0, A, 7, X, 1, 10.0, Y, 2 )}
\]

\[
X = (1.0, 2.0, 3.0, 4.0, 5.0)
\]

\[
Y = (1.0, 2.0, 3.0, 4.0, 5.0)
\]

Output:

\[
Y = (40.0, 78.0, 112.0, 140.0, 160.0)
\]
STRMV, DTRMV, CTRMV, ZTRMV, STPMV, DTPMV, CTPMV, and ZTPMV
(Matrix-Vector Product for a Triangular Matrix, Its Transpose, or Its
Conjugate Transpose)

Purpose

STRMV, DTRMV, STPMV, and DTPMV compute one of the following matrix-vector
products, using the vector \( x \) and triangular matrix \( A \) or its transpose:

\[
x + Ax\nonumber
\]
\[
x + A^T x\nonumber
\]

CTRMV, ZTRMV, CTPMV, and ZTPMV compute one of the following
matrix-vector products, using the vector \( x \) and triangular matrix \( A \), its transpose, or
its conjugate transpose:

\[
x + Ax\nonumber
\]
\[
x + A^T x\nonumber
\]
\[
x + A^{H} x\nonumber
\]

Matrix \( A \) 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 107. Data Types

<table>
<thead>
<tr>
<th>A, X</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>STRMV and STPMV</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DTRMV and DTPMV</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CTRMV and CTPMV</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZTRMV and ZTPMV</td>
</tr>
</tbody>
</table>

Note: On certain processors, SIMD algorithms may be used if alignment
requirements are met. For further details, see “Use of SIMD Algorithms by Some
Subroutines in the Libraries Provided by ESSL” on page 32.

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);
```

CBLAS

```
cblas_strmv | cblas_dtrmv | cblas_ctrmv | cblas_ztrmv (cblas_layout, cblas_uplo,
cblas_transa, cblas_diag, n, a, lda, x, incx);
cblas_stpmv | cblas_dtpmv | cblas_ctpmv | cblas_ztpmv (cblas_layout, cblas_uplo,
cblas_transa, cblas_diag, n, ap, x, incx);
```

On Entry
**cblas_layout**
indicates whether the input matrices are stored in row major order or column major order, where:
- If `cblas_layout` = CblasRowMajor, the matrices are stored in row major order.
- If `cblas_layout` = CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

**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'.

**cblas_uplo**
indicates whether matrix \( A \) is an upper or lower triangular matrix, where:
- If `cblas_uplo` = CblasUpper, \( A \) is an upper triangular matrix.
- If `cblas_uplo` = CblasLower, \( A \) is a lower triangular matrix.

Specified as: an object of enumerated type CBLAS_UPLO. It must be CblasUpper or CblasLower.

**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'.

**cblas_transa**
indicates the form of matrix \( A \) to use in the computation, where:
- If `cblas_transa` = CblasNoTrans, \( A \) is used in the computation.
- If `cblas_transa` = CblasTrans, \( A^T \) is used in the computation.
- If `cblas_transa` = CblasConjTrans, \( A^H \) is used in the computation.

Specified as: an object of enumerated type CBLAS_TRANSPOSE. It must be CblasNoTrans, CblasTrans, or CblasConjTrans.

**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'.

**cblas_diag**
indicates the characteristics of the diagonal of matrix \( A \), where:
- If `diag` = CblasUnit, \( A \) is a unit triangular matrix.
- If `diag` = CblasNonUnit, \( A \) is not a unit triangular matrix.
Specified as: an object of enumerated type CBLAS_DIAG. It must be CblasNonUnit or CblasUnit.

\[ n \]  is the order of triangular matrix \( A \).

Specified as: an integer; 0 ≤ \( n \) ≤ \( lda \).

\( a \)  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 \( A^T \) or \( A^H \); 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 107 on page 391

\( lda \)  is the leading dimension of the array specified for \( a \).

Specified as: an 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 107 on page 391

\( x \)  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 107 on page 391

\( incx \)  is the stride for vector \( x \).

Specified as: an 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 107 on page 391

**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 93.
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:

\[
x^\v A x \\
x^\v A^T x \\
x^\v A^H x \quad \text{(for CTRMV, ZTRMV, CTPMV, and ZTPMV only)}
\]

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 [40 on page 1365] and [46 on page 1366]. If $n$ is 0, no computation is performed.

Error conditions

Computational Errors
None

Input-Argument Errors

1. $\text{cblas\_layout} \neq \text{CblasRowMajor or CblasColMajor}$
2. $\text{uplo} \neq \text{'L'}$ or $\text{'U'}$
3. $\text{cblas\_uplo} \neq \text{CblasLower or CblasUpper}$
4. $\text{transa} \neq \text{'T'}, \text{'N'},$ or $\text{'C'}$
5. $\text{cblas\_transa} \neq \text{CblasNoT\, Trans, CblasTrans, or CblasConjT\, Trans}$
6. $\text{diag} \neq \text{'N'}$ or $\text{'U'}$
7. $\text{cblas\_diag} \neq \text{CblasNonUnit or CblasUnit}$
8. $n < 0$
9. $\text{lda} \leq 0$
10. $\text{lda} < n$
11. $\text{incx} = 0$

Examples

Example 1

This example shows the computation $x^\v A x$. 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:

\[
\begin{bmatrix}
1.0 & . & . & . \\
1.0 & 1.0 & . & . \\
2.0 & 3.0 & 1.0 & . \\
3.0 & 4.0 & 3.0 & 1.0
\end{bmatrix}
\]

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:
Example 2

This example shows the computation $x^T A x$. Matrix $A$ is a real 4 by 4 upper triangular matrix that is unit triangular, stored in upper-triangular storage mode. Vector $x$ is a vector of length 4. Matrix $A$ is:

\[
A = \begin{bmatrix}
1.0 & . & . & . \\
2.0 & 3.0 & . & . \\
3.0 & 4.0 & 3.0 & . \\
\end{bmatrix}
\]

\[X = (1.0, 2.0, 3.0, 4.0)\]

Output:
\[X = (1.0, 3.0, 11.0, 24.0)\]

Example 3

This example shows the computation $x^H A^H x$. Matrix $A$ is a complex 4 by 4 upper triangular matrix that is unit triangular, stored in upper-triangular storage mode. Vector $x$ is a vector of length 4. Matrix $A$ is:

\[
A = \begin{bmatrix}
1.0 & 2.0 & 3.0 & 2.0 \\
. & 1.0 & 2.0 & 5.0 \\
. & . & 1.0 & 3.0 \\
. & . & . & 1.0 \\
\end{bmatrix}
\]

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:
Example 4

This example shows the computation $x \leftarrow Ax$. 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:

$$
\begin{bmatrix}
1.0 & . & . & . \\
1.0 & 1.0 & . & . \\
2.0 & 3.0 & 1.0 & . \\
3.0 & 4.0 & 3.0 & 1.0
\end{bmatrix}
$$

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:

```
CALL STPMV( 'L', 'N', 'U', 4, AP, X, 1 )
```

$AP = (. , 1.0, 2.0, 3.0, . , 3.0, 4.0, . , 3.0, . )$

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

```
$X = (1.0, 3.0, 11.0, 24.0)$
```

Example 5

This example shows the computation $x \leftarrow A^T x$. Matrix $A$ is a real 4 by 4 upper triangular matrix that is not unit triangular, stored in upper-triangular-packed storage mode. Vector $x$ is a vector of length 4. Matrix $A$ 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:

```
CALL STPMV( 'U', 'T', 'N', 4, AP, X, 1 )
```

$AP = (1.0, 2.0, 2.0, 3.0, 2.0, 3.0, 2.0, 5.0, 3.0, 1.0)$

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

```
$X = (5.0, 18.0, 32.0, 41.0)$
```
Example 6

This example shows the computation $x^*A^Hx$. Matrix $A$ is a complex 4 by 4 upper triangular matrix that is unit triangular, stored in upper-triangular-packed storage mode. Vector $x$ is a vector of length 4. Matrix $A$ 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 $A$ 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:

```
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))

STRSV, DTRSV, CTRSV, ZTRSV, STPSV, DTPSV, CTPSV, and ZTPSV
(Solution of a Triangular System of Equations with a Single Right-Hand Side)

Purpose

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 \( x \)
and triangular matrix \( A \) or its transpose:

<table>
<thead>
<tr>
<th>Solution</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x \leftarrow A^{-1}x )</td>
<td>( Ax = b )</td>
</tr>
<tr>
<td>( x \leftarrow A^{\top}x )</td>
<td>( A^{\top}x = b )</td>
</tr>
</tbody>
</table>

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 \( x \)
and and triangular matrix \( A \), its transpose, or its conjugate transpose:

<table>
<thead>
<tr>
<th>Solution</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x \leftarrow A^{-1}x )</td>
<td>( Ax = b )</td>
</tr>
<tr>
<td>( x \leftarrow A^{\top}x )</td>
<td>( A^{\top}x = b )</td>
</tr>
<tr>
<td>( x \leftarrow A^{\ast}x )</td>
<td>( A^{\ast}x = b )</td>
</tr>
</tbody>
</table>

Matrix \( A \) 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 \( b \) 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 108. Data Types

<table>
<thead>
<tr>
<th>( A, x )</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>STRSV and STPSV</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DTRSV and DTPSV</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CTRSV and CTPSV</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZTRSV and ZTPSV</td>
</tr>
</tbody>
</table>

Note: On certain processors, SIMD algorithms may be used if alignment
requirements are met. For further details, see “Use of SIMD Algorithms by Some
Subroutines in the Libraries Provided by ESSL” on page 32.

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)
```
<table>
<thead>
<tr>
<th>C and C++</th>
<th>\texttt{strsv}</th>
<th>\texttt{dtrsv}</th>
<th>\texttt{ctrsv}</th>
<th>\texttt{ztrsv} (\texttt{uplo}, \texttt{transa}, \texttt{diag}, \texttt{n}, \texttt{a}, \texttt{lda}, \texttt{x}, \texttt{incx});</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>\texttt{stpsv}</td>
<td>\texttt{dtpsv}</td>
<td>\texttt{ctpsv}</td>
<td>\texttt{ztpsv} (\texttt{uplo}, \texttt{transa}, \texttt{diag}, \texttt{n}, \texttt{ap}, \texttt{x}, \texttt{incx});</td>
</tr>
<tr>
<td>CBLAS</td>
<td>\texttt{cblas_strsv}</td>
<td>\texttt{cblas_dtrsv}</td>
<td>\texttt{cblas_ctrsv}</td>
<td>\texttt{cblas_ztrsv} (\texttt{cblas_layout}, \texttt{cblas_uplo}, \texttt{cblas_transa}, \texttt{cblas_diag}, \texttt{n}, \texttt{a}, \texttt{lda}, \texttt{x}, \texttt{incx});</td>
</tr>
<tr>
<td></td>
<td>\texttt{cblas_stpsv}</td>
<td>\texttt{cblas_dtpsv}</td>
<td>\texttt{cblas_ctpsv}</td>
<td>\texttt{cblas_ztpsv} (\texttt{cblas_layout}, \texttt{cblas_uplo}, \texttt{cblas_transa}, \texttt{cblas_diag}, \texttt{n}, \texttt{ap}, \texttt{x}, \texttt{incx});</td>
</tr>
</tbody>
</table>

**On Entry**

\texttt{cblas_layout}

indicates whether the input matrices are stored in row major order or column major order, where:

- If \texttt{cblas_layout} = \texttt{CblasRowMajor}, the matrices are stored in row major order.
- If \texttt{cblas_layout} = \texttt{CblasColMajor}, the matrices are stored in column major order.

Specified as: an object of enumerated type \texttt{CBLAS\_LAYOUT}. It must be \texttt{CblasRowMajor} or \texttt{CblasColMajor}.

\texttt{uplo}

indicates whether matrix \texttt{A} is an upper or lower triangular matrix, where:

If \texttt{uplo} = \texttt{U}, \texttt{A} is an upper triangular matrix.
If \texttt{uplo} = \texttt{L}, \texttt{A} is a lower triangular matrix.

Specified as: a single character. It must be \texttt{U} or \texttt{L}.

\texttt{cblas_uplo}

indicates whether matrix \texttt{A} is an upper or lower triangular matrix, where:

If \texttt{cblas_uplo} = \texttt{CblasUpper}, \texttt{A} is an upper triangular matrix.
If \texttt{cblas_uplo} = \texttt{CblasLower}, \texttt{A} is a lower triangular matrix.

Specified as: an object of enumerated type \texttt{CBLAS\_UPLO}. It must be \texttt{CblasUpper} or \texttt{CblasLower}.

\texttt{transa}

indicates the form of matrix \texttt{A} used in the system of equations, where:

If \texttt{transa} = \texttt{N}, \texttt{A} is used, resulting in solution 1.
If \texttt{transa} = \texttt{T}, \texttt{A}^T is used, resulting in solution 2.
If \texttt{transa} = \texttt{C}, \texttt{A}^H is used, resulting in solution 3.

Specified as: a single character. It must be \texttt{N}, \texttt{T}, or \texttt{C}.

\texttt{cblas_transa}

indicates the form of matrix \texttt{A} to use in the computation, where:

If \texttt{cblas_transa} = \texttt{CblasNoTrans}, \texttt{A} is used, resulting in solution 1.
If \texttt{cblas_transa} = \texttt{CblasTrans}, \texttt{A}^T is used, resulting in solution 2.
If \texttt{cblas_transa} = \texttt{CblasConjTrans}, \texttt{A}^H is used, resulting in solution 3.

Specified as: an object of enumerated type \texttt{CBLAS\_TRANSPOSE}. It must be \texttt{CblasNoTrans}, \texttt{CblasTrans}, or \texttt{CblasConjTrans}.

\texttt{diag}

indicates the characteristics of the diagonal of matrix \texttt{A}, where:
If \( \text{diag} = 'U' \), \( A \) is a unit triangular matrix.

If \( \text{diag} = 'N' \), \( A \) is not a unit triangular matrix.

Specified as: a single character. It must be 'U' or 'N'.

**cblas_diag**

indicates the characteristics of the diagonal of matrix \( A \), where:

If \( \text{diag} = \text{CblasUnit} \), \( A \) is a unit triangular matrix.

If \( \text{diag} = \text{CblasNonUnit} \) \( A \) is not a unit triangular matrix.

Specified as: an object of enumerated type CBLAS_DIAG. It must be CblasNonUnit or CblasUnit.

\( n \) is the order of triangular matrix \( A \).

Specified as: an integer; \( n \geq 0 \) and \( n \leq \text{lda} \).

\( a \) is the upper or lower triangular matrix \( A \) of order \( n \), stored in upper- or lower-triangular storage mode, respectively. Specified as: an \( \text{lda} \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 108 on page 398.

\( \text{lda} \)

is the leading dimension of the array specified for \( a \).

Specified as: an integer; \( \text{lda} > 0 \) and \( \text{lda} \equiv n \).

\( \text{ap} \) 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 108 on page 398.

\( x \) 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 \( 1+(n-1)|\text{incx}| \), containing numbers of the data type indicated in Table 108 on page 398.

\( \text{incx} \)

is the stride for vector \( x \).

Specified as: an integer; \( \text{incx} > 0 \) or \( \text{incx} < 0 \).

**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 108 on page 398.

**Notes**

1. These subroutines accept lowercase letters for the \( \text{uplo} \), \( \text{transa} \), and \( \text{diag} \) arguments.

2. For STRSV, DTRSV, STPSV, and DTPSV, if you specify 'C' for the \( \text{transa} \) 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 93.

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:

$$x \rightarrow A^{-1}x$$
$$x \rightarrow A^T x$$
$$x \rightarrow A^{-h}x \text{ (only for CTRSV, ZTRSV, CTPSV, and ZTPSV)}$$

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 [40 on page 1365], [44 on 
page 1366], and [46 on page 1366].

Error conditions

Computational Errors

None

Input-Argument Errors

1. cblas_layout ≠ CblasRowMajor or CblasColMajor
2. uplo ≠ 'L' or 'U'
3. cblas_uplo ≠ CblasLower or CblasUpper
4. transa ≠ 'T', 'N', or 'C'
5. cblas_transa ≠ CblasNoTrans, CblasTrans, or CblasConjTrans
6. diag ≠ 'N' or 'U'
7. cblas_diag ≠ CblasNonUnit or CblasUnit
8. $n < 0$
9. lda ≤ 0
10. lda < $n$
11. incx = 0

Examples

Example 1

This example shows the solution $x \rightarrow A^{-1}x$. Matrix $A$ is a real 4 by 4 lower unit 
triangular matrix, stored in lower-triangular storage mode. Vector $x$ is a vector 
of length 4.

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 STRSV( 'L' , 'N' , 'U' , 4 , A , 4 , X , 1 )
```

```
A =

<table>
<thead>
<tr>
<th>1.0</th>
<th>2.0</th>
<th>3.0</th>
<th>2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>2.0</td>
<td>2.0</td>
<td>5.0</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>1.0</td>
</tr>
</tbody>
</table>
```

```
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 \( x^T A x \). Matrix \( A \) is a real 4 by 4 upper nonunit triangular matrix, stored in upper-triangular storage mode. Vector \( x \) 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 )
```

```
A =

<table>
<thead>
<tr>
<th>1.0</th>
<th>2.0</th>
<th>3.0</th>
<th>2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>2.0</td>
<td>2.0</td>
<td>5.0</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>1.0</td>
</tr>
</tbody>
</table>
```

```
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 \( x^H A^H x \). Matrix \( A \) is a complex 4 by 4 upper unit triangular matrix, stored in upper-triangular storage mode. Vector \( x \) is a vector of length 4.

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 CTRSV( 'U' , 'C' , 'U' , 4 , A , 4 , X , 1 )
```

```
A =

<table>
<thead>
<tr>
<th>(2.0, 2.0)</th>
<th>(3.0, 3.0)</th>
<th>(2.0, 2.0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>(2.0, 2.0)</td>
<td>(5.0, 5.0)</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>(3.0, 3.0)</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>
```

```
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 $x \cdot A^{-1} x$. Matrix $A$ is a real 4 by 4 lower unit triangular matrix, stored in lower-triangular-packed storage mode. Vector $x$ is a vector of length 4. Matrix $A$ is:

\[
\begin{bmatrix}
1.0 & . & . & . \\
1.0 & 1.0 & . & . \\
2.0 & 3.0 & 1.0 & . \\
3.0 & 4.0 & 3.0 & 1.0
\end{bmatrix}
\]

**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:

\[
\text{CALL STPSV('L', 'N', 'U', 4, AP, X, 1 )}
\]

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 $x \cdot A^T x$. Matrix $A$ is a real 4 by 4 upper nonunit triangular matrix, stored in upper-triangular-packed storage mode. Vector $x$ is a vector of length 4. Matrix $A$ 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:

\[
\text{CALL STPSV('U', 'T', 'N', 4, AP, X, 1 )}
\]

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 $x \cdot A^H x$. Matrix $A$ is a complex 4 by 4 upper unit triangular matrix, stored in upper-triangular-packed storage mode. Vector $x$ is a vector of length 4. Matrix $A$ 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 $A$ 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:

```
CALLCTPSV( '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))
```
STBMV, DTBMV, CTBMV, and ZTBMV (Matrix-Vector Product for a Triangular Band Matrix, Its Transpose, or Its Conjugate Transpose)

**Purpose**

STBMV and DTBMV compute one of the following matrix-vector products, using the vector $x$ and triangular band matrix $A$ or its transpose:

- $x + Ax$
- $x + A^T x$

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:

- $x + Ax$
- $x + A^T x$
- $x + A^H x$

Matrix $A$ can be either upper or lower triangular and is stored in upper- or lower-triangular-band-packed storage mode, respectively.

<table>
<thead>
<tr>
<th>$A, x$</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>STBMV</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DTBMV</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CTBMV</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZTBMV</td>
</tr>
</tbody>
</table>

**Syntax**

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL STBMV</th>
<th>DTBMV</th>
<th>CTBMV</th>
<th>ZTBMV (uplo, transa, diag, n, k, a, lda, x, incx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>stbmv</td>
<td>dtbmv</td>
<td>ctbmv</td>
<td>ztbmv (uplo, transa, diag, n, k, a, lda, x, incx);</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_stbmv</td>
<td>cblas_dtbmv</td>
<td>cblas_ctbmv</td>
<td>cblas_ztbmv (cblas_layout, cblas_uplo, cblas_transa, cblas_diag, n, k, a, lda, x, incx);</td>
</tr>
</tbody>
</table>

**On Entry**

- **cblas_layout** indicates whether the input matrices are stored in row major or column major order, where:
  - If $cblas_layout = \text{CblasRowMajor}$, the matrices are stored in row major order.
  - If $cblas_layout = \text{CblasColMajor}$, the matrices are stored in column major order.

  Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

- **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'.

cblas_uplo
indicates whether matrix A is an upper or lower triangular matrix, where:
If cblas_uplo = CblasUpper, A is an upper triangular matrix.
If cblas_uplo = CblasLower, A is a lower triangular matrix.
Specified as: an object of enumerated type CBLAS_UPLO. It must be CblasUpper or CblasLower.

tansa
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'.

cblas_transa
indicates the form of matrix A to use in the computation, where:
If cblas_transa = CblasNoTrans, A is used in the computation.
If cblas_transa = CblasTrans, A^T is used in the computation.
If cblas_transa = CblasConjTrans, A^H is used in the computation.
Specified as: an object of enumerated type CBLAS_TRANSPOSE. It must be CblasNoTrans, CblasTrans, or CblasConjTrans.

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'.

cblas_diag
indicates the characteristics of the diagonal of matrix A, where:
If diag = CblasUnit, A is a unit triangular matrix.
If diag = CblasNonUnit A is not a unit triangular matrix.
Specified as: an object of enumerated type CBLAS_DIAG. It must be CblasNonUnit or CblasUnit.

n is the order of triangular band matrix A. Specified as: an integer; n ≥ 0.
k is the upper or lower band width k of the matrix A.
Specified as: an integer; k ≥ 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 A^T or 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 109 on page 405.
lda
is the leading dimension of the array specified for a.
Specified as: an 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 109 on page 405.

\( incx \)
is the stride for vector \( x \).
Specified as: an 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 109 on page 405.

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 109. For an illustration of this technique, see Example 4.
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 109.
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:
\[ x \leftarrow Ax \]
\[ x \leftarrow A^T x \]
\[ x \leftarrow A^H x \]  (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 [42 on page 1365], [54 on page 1366], and [46 on page 1366]. If \( n \) is 0, no computation is performed.

**Error conditions**

**Computational Errors**

None

**Input-Argument Errors**

1. \( cblas_layout \neq \text{CblasRowMajor} \) or \( \text{CblasColMajor} \)
2. \( \text{uplo} \neq 'L' \) or \( 'U' \)
3. \( cblas_uplo \neq \text{CblasLower} \) or \( \text{CblasUpper} \)
4. \( \text{transa} \neq 'T', 'N', \) or \( 'C' \)
5. \( cblas_transa \neq \text{CblasNoT}, \text{CblasT}, \text{or CblasConjT} \)
6. \( \text{diag} \neq 'N' \) or \( 'U' \)
7. \( cblas_diag \neq \text{CblasNonUnit} \) or \( \text{CblasUnit} \)
8. \( n < 0 \)
9. \( k < 0 \)
10. \( lda \leq 0 \)
11. \( lda < k+1 \)
12. \( incx = 0 \)

**Examples**

**Example 1**

This example shows the computation \( x \leftarrow Ax \). Matrix \( A \) 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 \( x \) is a vector of length 7. Matrix \( A \) is:

\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 2.0 & 2.0 & 2.0 & 2.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 3.0 & 3.0 & 3.0 & 3.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 4.0 & 4.0 & 4.0 & 4.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 5.0 & 5.0 & 5.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 6.0 & 6.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 7.0
\end{bmatrix}
\]

Call Statement and Input:

```
CALL STBMV( 'U', 'N', 'N', 7, 3, A, 5, X, 1 )
```

Ex1: ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
\[
\begin{bmatrix}
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{bmatrix}
\]

\[
X = (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0)
\]

Output:
\[
X = (10.0, 28.0, 54.0, 88.0, 90.0, 78.0, 49.0)
\]

**Example 2**

This example shows the computation \(x^T A x\). Matrix \(A\) 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 \(x\) is a vector of length 7. Matrix \(A\) is:

\[
A = \begin{bmatrix}
1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 2.0 & 3.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 2.0 & 3.0 & 4.0 & 5.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 3.0 & 4.0 & 5.0 & 6.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 4.0 & 5.0 & 6.0 & 7.0 \\
\end{bmatrix}
\]

Call Statement and Input:

\[
\begin{array}{cccccccc}
\text{UPLO} & \text{TRANS} & \text{DIAG} & \text{N} & \text{K} & \text{A} & \text{LDA} & \text{X} & \text{INCX}
\end{array}
\]

\[
\begin{array}{cccccccc}
' \text{L}' & ' \text{T}' & ' \text{N}' & 7 & 3 & \text{A} & 5 & \text{X} & 1
\end{array}
\]

\[
\begin{bmatrix}
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & . \\
1.0 & 2.0 & 3.0 & 4.0 & . & . & . \\
. & . & . & . & . & . & . \\
\end{bmatrix}
\]

\[
X = (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0)
\]

Output:
\[
X = (10.0, 28.0, 54.0, 88.0, 90.0, 78.0, 49.0)
\]

**Example 3**

This example shows the computation \(x^H A^H x\). 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:

\[
A = \begin{bmatrix}
(1.0, 1.0) & (1.0, 1.0) & (1.0, 1.0) & (1.0, 1.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (2.0, 2.0) & (2.0, 2.0) & (2.0, 2.0) & (2.0, 2.0) & (0.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (0.0, 0.0) & (3.0, 3.0) & (3.0, 3.0) & (3.0, 3.0) & (3.0, 3.0) & (0.0, 0.0) \\
(0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (4.0, 4.0) & (4.0, 4.0) & (4.0, 4.0) & (4.0, 4.0) \\
(0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (5.0, 5.0) & (5.0, 5.0) & (5.0, 5.0) \\
(0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (6.0, 6.0) & (6.0, 6.0) \\
(0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (7.0, 7.0) \\
\end{bmatrix}
\]

Call Statement and Input:

\[
\begin{array}{cccccccc}
\text{UPLO} & \text{TRANS} & \text{DIAG} & \text{N} & \text{K} & \text{A} & \text{LDA} & \text{X} & \text{INCX}
\end{array}
\]

\[
\begin{array}{cccccccc}
' \text{U}' & ' \text{C}' & ' \text{N}' & 7 & 3 & \text{A} & 5 & \text{X} & 1
\end{array}
\]
Example 4

This example shows the computation $x + A^T x$, where $k > n$. Matrix $A$ 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 $x$ is a vector of length 4. Matrix $A$ is:

$$ A = \begin{bmatrix}
  1.0 & 1.0 & 1.0 & 1.0 \\
  . & 2.0 & 2.0 & 2.0 \\
  . & . & 3.0 & 3.0 \\
  . & . & . & 4.0 \\
\end{bmatrix} $$

Call Statement and Input:

```
CALL STBMV('U', 'T', 'N', 4, 5, A, 6, X, 1)
```

Output:

```
A = \begin{bmatrix}
  1.0 & 1.0 & 1.0 & 1.0 \\
  . & 2.0 & 2.0 & 2.0 \\
  . & . & 3.0 & 3.0 \\
  . & . & . & 4.0 \\
\end{bmatrix}
```

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

Output:

$X = (1.0, 5.0, 14.0, 30.0)$
STBSV, DTBSV, CTBSV, and ZTBSV (Triangular Band Equation Solve)

Purpose

STBSV and DTBSV solve one of the following triangular banded systems of equations with a single right-hand side, using the vector \( x \) and triangular band matrix \( A \) or its transpose:

Solution | Equation
--- | ---
1. \( x \leftarrow A^{-1}x \) | \( Ax = b \)
2. \( x \leftarrow A^{T}x \) | \( A^{T}x = b \)

CTBSV and ZTBSV solve one of the following triangular banded systems of equations with a single right-hand side, using the vector \( x \) and triangular band matrix \( A \), its transpose, or its conjugate transpose:

Solution | Equation
--- | ---
1. \( x \leftarrow A^{-1}x \) | \( Ax = b \)
2. \( x \leftarrow A^{T}x \) | \( A^{T}x = b \)
3. \( x \leftarrow A^{H}x \) | \( A^{H}x = b \)

Matrix \( A \) can be either upper or lower triangular and is stored in upper- or lower-triangular-band-packed storage mode, respectively.

Table 110. Data Types

<table>
<thead>
<tr>
<th>( A, x )</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>STBSV</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DTBSV</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CTBSV</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZTBSV</td>
</tr>
</tbody>
</table>

Syntax

**Fortran**

```fortran
CALL STBSV | DTBSV | CTBSV | ZTBSV (uplo, trans, diag, n, k, a, lda, x, incx)
```

**C and C++**

```c
stbsv | dtbsv | ctbsv | ztbsv (uplo, trans, diag, n, k, a, lda, x, incx);
```

**CBLAS**

```c
blas_stbsv | blas_dtbsv | blas_ctbsv | blas_ztbsv (blas_layout, blas_uplo, blas_trans, blas_diag, n, k, a, lda, x, incx);
```

On Entry

- **blas_layout**
  - indicates whether the input matrices are stored in row major order or column major order, where:
    - If `blas_layout` = CblasRowMajor, the matrices are stored in row major order.
    - If `blas_layout` = CblasColMajor, the matrices are stored in column major order.
    - Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

- **uplo**
  - indicates whether matrix \( A \) is an upper or lower triangular band matrix, where:
If $\text{uplo} = 'U'$, $A$ is an upper triangular matrix.
If $\text{uplo} = 'L'$, $A$ is a lower triangular matrix.
Specified as: a single character. It must be 'U' or 'L'.

$cblas\_uplo$
indicates whether matrix $A$ is an upper or lower triangular matrix, where:
If $cblas\_uplo = \text{CblasUpper}$, $A$ is an upper triangular matrix.
If $cblas\_uplo = \text{CblasLower}$, $A$ is a lower triangular matrix.
Specified as: an object of enumerated type CBLAS_UPLO. It must be CblasUpper or CblasLower.

$trans$
indicates the form of matrix $A$ used in the system of equations, where:
If $trans = 'N'$, $A$ 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'.

$cblas\_transa$
indicates the form of matrix $A$ to use in the computation, where:
If $cblas\_transa = \text{CblasNoTrans}$, $A$ is used, resulting in solution 1.
If $cblas\_transa = \text{CblasTrans}$, $A^T$ is used, resulting in solution 2.
If $cblas\_transa = \text{CblasConjTrans}$, $A^H$ is used, resulting in solution 3.
Specified as: an object of enumerated type CBLAS_TRANSPOSE. It must be CblasNoTrans, CblasTrans, or CblasConjTrans.

$diag$
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'.

$cblas\_diag$
indicates the characteristics of the diagonal of matrix $A$, where:
If $diag = \text{CblasUnit}$, $A$ is a unit triangular matrix.
If $diag = \text{CblasNonUnit}$, $A$ is not a unit triangular matrix.
Specified as: an object of enumerated type CBLAS_DIAG. It must be CblasNonUnit or CblasUnit.

$n$
is the order of triangular band matrix $A$. Specified as: an integer; $n \geq 0$.

$k$
is the upper or lower band width $k$ of the matrix $A$. Specified as: an 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 110 on page 411.
lda

is the leading dimension of the array specified for a. Specified as: an integer; 
lda > 0 and lda ≥ 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 110 on 
page 411.

incx

is the stride for vector x. Specified as: an 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 110 on page 411.

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 ≥ 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_{i,1} and all other elements 
in the array specified for A, as described in "Triangular Band Matrix" on page 
109. For an illustration of this technique, see Example 3.
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 109.
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. x=A^{-1}x
2. x=A^{T}x
3. x=A^{H}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 [42 on page 1365], [54 on page 1366], and [46 on page 1366]. If \( n \) is 0, no computation is performed.

**Error conditions**

**Computational Errors**

None

**Input-Argument Errors**

1. \( \text{cblas_layout} \neq \text{CblasRowMajor or CblasColMajor} \)
2. \( n < 0 \)
3. \( k < 0 \)
4. \( lda = 0 \)
5. \( lda < k+1 \)
6. \( \text{incx} = 0 \)
7. \( \text{uplo} \neq 'L' \text{ or 'U'} \)
8. \( \text{cblas_uplo} \neq \text{CblasLower or CblasUpper} \)
9. \( \text{trans} \neq 'T', 'N', \text{ or 'C'} \)
10. \( \text{cblas_transa} \neq \text{CblasNoTrans, CblasTrans, or CblasConjTrans} \)
11. \( \text{diag} \neq 'N' \text{ or 'U'} \)
12. \( \text{cblas_diag} \neq \text{CblasNonUnit or CblasUnit} \)

**Examples**

**Example 1**

This example shows the solution \( x = A^{-1}x \). 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:

\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 4.0 & 2.0 & 3.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 4.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 4.0 & 2.0 & 2.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 3.0 & 1.0 & 1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 3.0 & 2.0 & 2.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 3.0 & 1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 2.0 & 2.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \\
\end{bmatrix}
\]

Call Statement and Input:

\[
\text{CALL STBSV( 'U', 'N', 'N', 9, 2, A, 3, X, 1 )}
\]

\[
A = \begin{bmatrix}
. & . & 1.0 & 3.0 & 1.0 & 2.0 & 1.0 & 2.0 & 0.0 \\
. & . & 1.0 & 2.0 & 1.0 & 2.0 & 1.0 & 2.0 & 1.0 \\
1.0 & 4.0 & 4.0 & 4.0 & 3.0 & 3.0 & 3.0 & 2.0 & 1.0 \\
\end{bmatrix}
\]

\[
X = (2.0, 7.0, 1.0, 8.0, 2.0, 8.0, 1.0, 8.0, 3.0)
\]
Output:
\[ \mathbf{x} = (1.0, 1.0, 0.0, 1.0, 0.0, 2.0, 0.0, 1.0, 3.0) \]

**Example 2**
This example shows the solution \( \mathbf{x} \mathbf{A}^T \mathbf{x} \), solving the same system as in Example 1. Matrix \( \mathbf{A} \) 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 \( \mathbf{x} \) is a vector of length 9 where matrix \( \mathbf{A} \) is:

\[
\begin{bmatrix}
1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 4.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 2.0 & 4.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 3.0 & 1.0 & 4.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 1.0 & 2.0 & 3.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 2.0 & 1.0 & 3.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & 3.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 2.0 & 1.0 & 2.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 2.0 & 1.0 \\
\end{bmatrix}
\]

Call Statement and Input:
```
CALL STBSV( 'L', 'T', 'N', 9, 2, A, 3, X, 1 )
```

\[
\mathbf{A} = \begin{bmatrix}
1.0 & 4.0 & 4.0 & 4.0 & 3.0 & 3.0 & 2.0 & 1.0 \\
1.0 & 2.0 & 1.0 & 2.0 & 1.0 & 2.0 & 1.0 & 2.0 \\
1.0 & 3.0 & 1.0 & 2.0 & 1.0 & 2.0 & 0.0 & . \\
\end{bmatrix}
\]

\[ \mathbf{x} = (\text{same as input } \mathbf{x} \text{ in Example 1}) \]

Output:
\[ \mathbf{x} = (\text{same as output } \mathbf{x} \text{ in Example 1}) \]

**Example 3**
This example shows the solution \( \mathbf{x} \mathbf{A}^T \mathbf{x} \), where \( k > n \). Matrix \( \mathbf{A} \) 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 \( \mathbf{x} \) is a vector of length 4 where matrix \( \mathbf{A} \) is:

\[
\begin{bmatrix}
1.0 & 2.0 & 3.0 & 2.0 \\
0.0 & 2.0 & 2.0 & 5.0 \\
0.0 & 0.0 & 3.0 & 3.0 \\
0.0 & 0.0 & 0.0 & 1.0 \\
\end{bmatrix}
\]

Call Statement and Input:
```
CALL STBSV( 'U', 'T', 'N', 4, 5, A, 6, X, 1 )
```

\[
\mathbf{A} = \begin{bmatrix}
. & . & . & . \\
. & . & 2.0 & . \\
. & . & 3.0 & 5.0 \\
. & 2.0 & 2.0 & 3.0 \\
1.0 & 2.0 & 3.0 & 1.0 \\
\end{bmatrix}
\]

\[ \mathbf{x} = (5.0, 18.0, 32.0, 41.0) \]
Example 4

This example shows the solution \( x^* A^T x \). Matrix \( A \) 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 \( x \) is a vector of length 7. Matrix \( A \) is:

\[
\begin{bmatrix}
(1.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) \\
(1.0, 2.0) & (2.0, 1.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) \\
(1.0, 3.0) & (2.0, 2.0) & (3.0, 1.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) \\
(1.0, 4.0) & (2.0, 3.0) & (3.0, 3.0) & (4.0, 1.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (2.0, 4.0) & (3.0, 3.0) & (4.0, 2.0) & (2.0, 1.0) & (0.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (0.0, 0.0) & (3.0, 3.0) & (4.0, 3.0) & (5.0, 1.0) & (3.0, 1.0) & (0.0, 0.0) \\
(0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (4.0, 4.0) & (5.0, 2.0) & (6.0, 1.0) & (2.0, 1.0)
\end{bmatrix}
\]

Call Statement and Input:

```
CALL CTBSV( 'L', 'T', 'N', 7, 3, A, 4, X, 1 )
```

\[
A = \begin{bmatrix}
(1.0, 0.0) & (2.0, 1.0) & (3.0, 1.0) & (4.0, 1.0) & (2.0, 1.0) & (3.0, 1.0) & (2.0, 1.0) \\
(1.0, 2.0) & (2.0, 2.0) & (3.0, 3.0) & (4.0, 2.0) & (5.0, 1.0) & (6.0, 1.0) & \cdot \\
(1.0, 3.0) & (2.0, 3.0) & (3.0, 3.0) & (4.0, 3.0) & (5.0, 2.0) & \cdot & \cdot \\
(1.0, 4.0) & (2.0, 4.0) & (3.0, 3.0) & (4.0, 4.0) & \cdot & \cdot & \cdot
\end{bmatrix}
\]

\[
X = ((2.0, 2.0), (7.0, 1.0), (1.0, 1.0), (8.0, 1.0), (2.0, 0.0), (8.0, 1.0), (1.0, 2.0))
\]

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 Matrix-Vector Subprograms

This contains the sparse matrix-vector subprogram descriptions.
DSMMX (Matrix-Vector Product for a Sparse Matrix in Compressed-Matrix Storage Mode)

Purpose

This subprogram computes the matrix-vector product for sparse matrix $A$, stored in compressed-matrix storage mode, using the matrix and vectors $x$ and $y$:

$$y = Ax$$

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 $y = A^T x$.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL DSMMX (m, nz, ac, ka, lda, x, y)</td>
</tr>
<tr>
<td>C and C++</td>
<td>dsmmx (m, nz, ac, ka, lda, x, y);</td>
</tr>
</tbody>
</table>

On Entry

- $m$ is the number of rows in sparse matrix $A$ and the number of elements in vector $y$. Specified as: an integer; $m \geq 0$.
- $nz$ is the maximum number of nonzero elements in each row of sparse matrix $A$. Specified as: an 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 integers, where $1 \leq (\text{elements of } KA) \leq n$.
- $lda$ is the size of the leading dimension of the arrays specified for $ac$ and $ka$. Specified as: an 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.

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.
3. For a description of how sparse matrices are stored in compressed-matrix storage mode, see “Compressed-Matrix Storage Mode” on page 117.
4. If your sparse matrix is stored by rows, as defined in “Storage-by-Rows” on page 122, 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 1323, 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:

\[ y \leftarrow Ax \]

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:

\[
\begin{bmatrix}
  y_1 \\
  . \\
  . \\
  . \\
  y_m
\end{bmatrix}
\leftarrow
\begin{bmatrix}
  a_{11} & \cdots & a_{1n} \\
  . & . & . \\
  . & . & . \\
  . & . & . \\
  a_{m1} & \cdots & a_{mn}
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  . \\
  . \\
  . \\
  x_n
\end{bmatrix}
\]

See reference [87 on page 1368]. 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 1323.

Error conditions

**Computational Errors**

None

**Input-Argument Errors**

1. \( m < 0 \)
2. \( lda \leq 0 \)
3. \( m > lda \)
4. \( nz < 0 \)

Examples

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:

\[
\begin{bmatrix}
4.0 & 0.0 & 7.0 & 0.0 & 0.0 & 0.0 \\
3.0 & 4.0 & 0.0 & 2.0 & 0.0 & 0.0 \\
0.0 & 2.0 & 4.0 & 0.0 & 4.0 & 0.0 \\
0.0 & 0.0 & 7.0 & 4.0 & 0.0 & 1.0 \\
1.0 & 0.0 & 0.0 & 3.0 & 4.0 & 0.0 \\
1.0 & 1.0 & 0.0 & 0.0 & 3.0 & 4.0
\end{bmatrix}
\]

Call Statement and Input:

\[
\begin{array}{|c|c|c|c|c|c|c|}
\hline
M & NZ & AC & KA & LDA & X & Y \\
\hline
2 & 6 & AC & KA & 6 & X & Y \\
\hline
\end{array}
\]

\text{CALL DSMMX( 6, 4, AC, KA, 6, X, Y )}

\[
AC = \begin{bmatrix}
4.0 & 7.0 & 0.0 & 0.0 \\
4.0 & 3.0 & 2.0 & 0.0 \\
4.0 & 2.0 & 4.0 & 0.0 \\
4.0 & 7.0 & 1.0 & 0.0 \\
4.0 & 1.0 & 3.0 & 0.0 \\
4.0 & 1.0 & 1.0 & 3.0
\end{bmatrix}
\]

\[
KA = \begin{bmatrix}
1 & 3 & 1 & 1 \\
2 & 1 & 4 & 1 \\
3 & 2 & 5 & 1 \\
4 & 3 & 6 & 1 \\
5 & 1 & 4 & 1 \\
6 & 1 & 2 & 5
\end{bmatrix}
\]

\[
X = (1.0, 2.0, 3.0, 4.0, 5.0, 6.0)
\]

Output:

\[
Y = (25.0, 19.0, 36.0, 43.0, 33.0, 42.0)
\]
DSMTM (Transpose a Sparse Matrix in Compressed-Matrix Storage Mode)

Purpose

This subprogram transposes sparse matrix $A$, stored in compressed-matrix storage mode, where $A$ contains long-precision real numbers.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL DSMTM (m, nz, ac, ka, lda, n, nt, at, kt, ldt, aux, naux)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>dsmtm (m, nz, ac, ka, lda, n, nt, at, kt, ldt, aux, naux);</td>
</tr>
</tbody>
</table>

On Entry

- $m$ is the number of rows in sparse matrix $A$. Specified as: an integer; $m \geq 0$.
- $nz$ is the maximum number of nonzero elements in each row of sparse matrix $A$. Specified as: an integer; $nz \geq 0$.
- $ac$ is the $m \times n$ 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 integers, where $1 \leq (\text{elements of } KA) \leq n$.
- $lda$ is the size of the leading dimension of the arrays specified for $ac$ and $ka$. Specified as: an integer; $lda > 0$ and $lda \geq m$.
- $n$ is the number of columns in sparse matrix $A$. Specified as: an integer; $0 \leq n \leq ldt$ and $n \geq (\text{maximum column index in } KA)$.
- $nt$ is the number of columns in output arrays $AT$ and $KT$ that are available for use. Specified as: an 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: an 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: an 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, \( naux \geq n \).

**On Return**

- **n** is the number of rows in the transposed matrix \( A^T \). Returned as: an 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: an integer; \( nt \leq m \).

- **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 \( ldl \) 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 \( ldl \) by (at least) \( nt \) array, containing integers, where \( 1 \leq (\text{elements of } KT) \leq m \).

**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.
4. For a description of how sparse matrices are stored in compressed-matrix storage mode, see "Compressed-Matrix Storage Mode" on page 117.
5. If your sparse matrix is stored by rows, as defined in "Storage-by-Rows" on page 122, 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 1323, 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 51.

**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 [87 on page 1368]. 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 1323.
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:
   \[ \text{naux} < n \]
   \[ nt \leq 0 \]
9. Error 2015 is recoverable or \( naux \neq 0 \), and \( naux \) is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

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 \( A^T \), stored in compressed-matrix storage mode in arrays \( AT \) and \( KT \), is as follows. Matrix \( A^T \) 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  LTD  AUX  NAUX
    |  |  |  |  |  |  |  |  |  |  |
CALL DSMTM( 5 , 2 , AC , KA , 5 , 5 , 4 , AT , KT , 6 , AUX , 5 )
```

Chapter 8. Linear Algebra Subprograms 423
\[
\begin{bmatrix}
11.0 & 0.0 \\
21.0 & 23.0 \\
33.0 & 34.0 \\
42.0 & 44.0 \\
51.0 & 53.0
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 1 \\
1 & 3 \\
3 & 4 \\
2 & 4 \\
1 & 3
\end{bmatrix}
\]

Output:
\[
\begin{align*}
N &= 4 \\
NT &= 3
\end{align*}
\]

\[
\begin{bmatrix}
11.0 & 21.0 & 51.0 & 0.0 \\
42.0 & 0.0 & 0.0 & 0.0 \\
35.0 & 23.0 & 53.0 & 0.0 \\
34.0 & 44.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 \\
\cdots & \cdots & \cdots & \cdots
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 2 & 5 & 1 \\
4 & 1 & 1 & 1 \\
3 & 2 & 5 & 1 \\
3 & 4 & 1 & 1 \\
1 & 1 & 1 & 1 \\
\cdots & \cdots & \cdots & \cdots
\end{bmatrix}
\]
DSDMX (Matrix-Vector Product for a Sparse Matrix or Its Transpose in Compressed-Diagonal Storage Mode)

**Purpose**

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 $y$:

\[
\begin{align*}
y &= Ax \\
y &= A^T x
\end{align*}
\]

where $A$, $x$, and $y$ contain long-precision real numbers.

**Syntax**

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL DSDMX (iopt, n, nd, ad, lda, trans, la, x, y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>dsdmx (iopt, n, nd, ad, lda, trans, la, x, y);</td>
</tr>
</tbody>
</table>

**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: an 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: an 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: an integer; $nd \geq 0$.

- $ad$ is the sparse matrix $A$ of order $n$, stored in compressed diagonal storage in an array, referred to as $A0$. 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 $trans = 'T'$, $A^T$ is used in the computation.

  **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: 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: an integer; $lda > 0$ and $lda \geq n$.

- $trans$ indicates the form of matrix $A$ to use in the computation, where:
If $\text{trans} = 'N'$, $A$ is used in the computation.

If $\text{trans} = 'T'$, $A^T$ is used in the computation.

Specified as: a single character; $\text{trans} = 'N'$ or 'T'.

$la$ is the array, referred to as $\text{LA}$, containing the diagonal numbers $k$ for the diagonals stored in each corresponding column in array $\text{AD}$. (For an explanation of how diagonal numbers are assigned, see "Compressed-Diagonal Storage Mode" on page 118.)

Specified as: a one-dimensional array of (at least) length $nd$, containing integers; $1 - n \leq \text{LA}(i) \leq n - 1$.

$x$ is the vector $x$ of length $n$. Specified as: a one-dimensional array, containing long-precision real numbers.

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 $\text{trans}$ 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 118.

Function

The matrix-vector product of a square sparse matrix or its transpose, is computed for a matrix stored in compressed-diagonal storage mode:

$$y \leftarrow Ax$$

$$y \leftarrow A^T x$$

where:

$A$ is a sparse matrix of order $n$, stored in compressed-diagonal storage mode in $\text{AD}$ and $\text{LA}$, using the storage variation for either general or symmetric sparse matrices, as indicated by the $\text{iopt}$ argument.

$x$ and $y$ are vectors of length $n$.

It is expressed as follows for $y \leftarrow Ax$:

$$
\begin{bmatrix}
  y_1 \\
  \vdots \\
  y_n
\end{bmatrix}
\leftarrow
\begin{bmatrix}
  a_{11} & \cdots & a_{1n} \\
  \vdots & \ddots & \vdots \\
  a_{n1} & \cdots & a_{nn}
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  \vdots \\
  x_n
\end{bmatrix}
$$

It is expressed as follows for $y \leftarrow A^T x$:
If \( n \) is 0, no computation is performed; if \( nd \) is 0, output vector \( y \) is set to zero, because matrix \( A \) 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 \( j = 1, n \)

### Examples

#### Example 1

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 general sparse matrices, storing all nonzero diagonals. Matrix \( A \) is:

\[
\begin{bmatrix}
4.0 & 0.0 & 7.0 & 0.0 & 0.0 & 0.0 \\
3.0 & 4.0 & 0.0 & 2.0 & 0.0 & 0.0 \\
0.0 & 2.0 & 4.0 & 0.0 & 4.0 & 0.0 \\
0.0 & 0.0 & 7.0 & 4.0 & 0.0 & 1.0 \\
1.0 & 0.0 & 0.0 & 3.0 & 4.0 & 0.0 \\
1.0 & 1.0 & 0.0 & 0.0 & 3.0 & 4.0 \\
\end{bmatrix}
\]

Call Statement and Input:

```fortran
CALL DSDMX( 0 , 6 , 5 , AD , 6 , 'N' , LA , X , Y )
```

\( AD = \begin{bmatrix}
4.0 & 0.0 & 0.0 & 0.0 & 7.0 \\
4.0 & 0.0 & 0.0 & 3.0 & 2.0 \\
4.0 & 0.0 & 0.0 & 2.0 & 4.0 \\
4.0 & 0.0 & 0.0 & 7.0 & 1.0 \\
4.0 & 0.0 & 1.0 & 3.0 & 0.0 \\
4.0 & 1.0 & 1.0 & 3.0 & 0.0 \\
\end{bmatrix} \)

\( LA = (0, -5, -4, -1, 2) \)

\( X = (1.0, 2.0, 3.0, 4.0, 5.0, 6.0) \)

Output:

\( Y = (25.0, 19.0, 36.0, 43.0, 33.0, 42.0) \)

#### Example 2
This example shows the matrix-vector product using \textit{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:

\[
\begin{bmatrix}
11.0 & 0.0 & 13.0 & 0.0 & 15.0 & 0.0 \\
0.0 & 22.0 & 0.0 & 24.0 & 0.0 & 26.0 \\
13.0 & 0.0 & 33.0 & 0.0 & 35.0 & 0.0 \\
0.0 & 24.0 & 0.0 & 44.0 & 0.0 & 46.0 \\
15.0 & 0.0 & 35.0 & 0.0 & 55.0 & 0.0 \\
0.0 & 26.0 & 0.0 & 46.0 & 0.0 & 66.0 \\
\end{bmatrix}
\]

Call Statement and Input:

\[
\begin{array}{llllllll}
\text{IOPT} & \text{N} & \text{ND} & \text{AD} & \text{LDA} & \text{TRANS} & \text{LA} & \text{X} & \text{Y} \\
\hline
1 & 6 & 3 & \text{AD} & 6 & 'N' & \text{LA} & \text{X} & \text{Y} \\
\end{array}
\]

\[
\begin{array}{llllll}
\text{AD} & = & \begin{bmatrix}
11.0 & 13.0 & 0.0 \\
22.0 & 24.0 & 0.0 \\
33.0 & 35.0 & 0.0 \\
44.0 & 46.0 & 0.0 \\
55.0 & 0.0 & 15.0 \\
66.0 & 0.0 & 26.0 \\
\end{bmatrix} \\
\text{LA} & = & (0, 2, -4) \\
\text{X} & = & (1.0, 2.0, 3.0, 4.0, 5.0, 6.0) \\
\text{Y} & = & (125.0, 296.0, 287.0, 500.0, 395.0, 632.0) \\
\end{array}
\]

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 \textit{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:

\[
\begin{array}{llllllll}
\text{IOPT} & \text{N} & \text{ND} & \text{AD} & \text{LDA} & \text{TRANS} & \text{LA} & \text{X} & \text{Y} \\
\hline
0 & 6 & 5 & \text{AD} & 6 & 'T' & \text{LA} & \text{X} & \text{Y} \\
\end{array}
\]

\[
\begin{array}{llll}
\text{AD} & = & (\text{same as input AD in Example 1}) \\
\text{LA} & = & (\text{same as input LA in Example 1}) \\
\text{X} & = & (\text{same as input X in Example 1}) \\
\text{Y} & = & (21.0, 20.0, 47.0, 35.0, 50.0, 28.0) \\
\end{array}
\]
Chapter 9. Matrix Operations

The matrix operation subroutines are described here.

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 [40 on page 1365]. The matrix operation subroutines also include the commonly used matrix operations: addition, subtraction, multiplication, and transposition.

Table 11. List of Matrix Operation Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGEADD</td>
<td>DGEADD</td>
<td>“SGEADD, DGEADD, CGEADD, and ZGEADD (Matrix Addition for General Matrices or Their Transposes)” on page 434</td>
</tr>
<tr>
<td>CGEADD</td>
<td>ZGEADD</td>
<td></td>
</tr>
<tr>
<td>SGESUB</td>
<td>DGESUB</td>
<td>“SGESUB, DGESUB, CGESUB, and ZGESUB (Matrix Subtraction for General Matrices or Their Transposes)” on page 440</td>
</tr>
<tr>
<td>CGESUB</td>
<td>ZGESUB</td>
<td></td>
</tr>
<tr>
<td>SGEMUL</td>
<td>DGEMUL</td>
<td>“SGEMUL, DGEMUL, CGEMUL, and ZGEMUL (Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes)” on page 446</td>
</tr>
<tr>
<td>CGEMUL</td>
<td>ZGEMUL</td>
<td>DGEMLP*</td>
</tr>
<tr>
<td>SGEMMS</td>
<td>DGEMMS</td>
<td>“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 455</td>
</tr>
<tr>
<td>CGEMMS</td>
<td>ZGEMMS</td>
<td></td>
</tr>
<tr>
<td>SGEMM*</td>
<td>DGEMM*</td>
<td>“SGEMM, DGEMM, CGEMM, and ZGEMM (Combined Matrix Multiplication and Addition for General Matrices, Their Transposes, or Conjugate Transposes)” on page 461</td>
</tr>
<tr>
<td>cblas_sgemm*</td>
<td>cblas_dgemm*</td>
<td></td>
</tr>
<tr>
<td>cblas_cgemm*</td>
<td>cblas_zgemm*</td>
<td></td>
</tr>
<tr>
<td>SSYMM*</td>
<td>DSYMM*</td>
<td>“SSYMM, DSYMM, CSYMM, ZSYMM, CHEMM, and ZHEMM (Matrix-Matrix Product Where One Matrix is Real or Complex Symmetric or Complex Hermitian)” on page 470</td>
</tr>
<tr>
<td>CSYMM*</td>
<td>ZSYMM*</td>
<td></td>
</tr>
<tr>
<td>CHEMM*</td>
<td>ZHEMM*</td>
<td></td>
</tr>
<tr>
<td>cblas_ssymm*</td>
<td>cblas_dsymm*</td>
<td></td>
</tr>
<tr>
<td>cblas_csymm*</td>
<td>cblas_zsymm*</td>
<td></td>
</tr>
<tr>
<td>cblas_chemm*</td>
<td>cblas_zhemm*</td>
<td></td>
</tr>
<tr>
<td>STRMM*</td>
<td>DTRMM*</td>
<td>“STRMM, DTRMM, CTRMM, and ZTRMM (Triangular Matrix-Matrix Product)” on page 478</td>
</tr>
<tr>
<td>CTRMM*</td>
<td>cblas_strmm*</td>
<td></td>
</tr>
<tr>
<td>cblas_ctrmm*</td>
<td>cblas_zctrmm*</td>
<td></td>
</tr>
<tr>
<td>STRSM*</td>
<td>DTRSM*</td>
<td>“STRSM, DTRSM, CTRSM, and ZTRSM (Solution of Triangular Systems of Equations with Multiple Right-Hand Sides)” on page 486</td>
</tr>
<tr>
<td>CTRSM*</td>
<td>cblas_strsm*</td>
<td></td>
</tr>
<tr>
<td>cblas_cstrsm*</td>
<td>cblas_zctrsm*</td>
<td></td>
</tr>
<tr>
<td>SSYRK*</td>
<td>DSYRK*</td>
<td>“SSYRK, DSYRK, CSYRK, ZSYRK, CHERK, and ZHERK (Rank-K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix)” on page 494</td>
</tr>
<tr>
<td>CSYRK*</td>
<td>ZSYRK*</td>
<td></td>
</tr>
<tr>
<td>CHERK*</td>
<td>ZHERK*</td>
<td></td>
</tr>
<tr>
<td>cblas_ssyrk*</td>
<td>cblas_dssyrk*</td>
<td></td>
</tr>
<tr>
<td>cblas_csyrk*</td>
<td>cblas_zssyrk*</td>
<td></td>
</tr>
<tr>
<td>cblas_cherk*</td>
<td>cblas_zcherk*</td>
<td></td>
</tr>
</tbody>
</table>
Table 11. List of Matrix Operation Subroutines (continued)

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSYR2K*</td>
<td>DSYR2K*</td>
<td>&quot;SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, and ZHER2K (Rank-2K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix)&quot; on page 501</td>
</tr>
<tr>
<td>CSYR2K*</td>
<td>ZSYR2K*</td>
<td></td>
</tr>
<tr>
<td>CHER2K*</td>
<td>ZHER2K*</td>
<td></td>
</tr>
<tr>
<td>cblas_ssymr2k*</td>
<td>cblas_dsymr2k*</td>
<td></td>
</tr>
<tr>
<td>cblas_csymr2k*</td>
<td>cblas_zsymr2k*</td>
<td></td>
</tr>
<tr>
<td>cblas_cher2k*</td>
<td>cblas_zher2k*</td>
<td></td>
</tr>
<tr>
<td>SGETMI</td>
<td>DGETMI</td>
<td>&quot;SGETMI, DGETMI, CGETMI, ZGETMI, CGECMI and ZGECMI (General Matrix Transpose or Conjugate Transpose [In-Place])&quot; on page 509</td>
</tr>
<tr>
<td>CGETMI</td>
<td>ZGETMI</td>
<td></td>
</tr>
<tr>
<td>CGECMI</td>
<td>ZGECMI</td>
<td></td>
</tr>
<tr>
<td>SGETMO</td>
<td>DGETMO</td>
<td>&quot;SGETMO, DGETMO, CGETMO, ZGETMO, CGECMO, and ZGECMO (General Matrix Transpose or Conjugate Transpose [Out-of-Place])&quot; on page 512</td>
</tr>
<tr>
<td>CGETMO</td>
<td>ZGETMO</td>
<td></td>
</tr>
<tr>
<td>CGECMO</td>
<td>ZGECMO</td>
<td></td>
</tr>
</tbody>
</table>

* Level 3 BLAS

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

Use Considerations

This 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 $\text{transa}$ and $\text{transb}$ arguments:

- 'N' Normal form
- 'T' Transposed form
- 'C' Conjugate transposed form

The four and nine possible combinations are defined as follows:

<table>
<thead>
<tr>
<th>Real Combinations</th>
<th>Complex Combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AB$</td>
<td>$AB$</td>
</tr>
<tr>
<td>$A^TB$</td>
<td>$A^T\bar{B}$</td>
</tr>
<tr>
<td>$AB^T$</td>
<td>$\bar{A}B^T$</td>
</tr>
<tr>
<td>$A^TB^T$</td>
<td>$A^T\bar{B}^T$</td>
</tr>
<tr>
<td>$A^TB^H$</td>
<td>$A^T\bar{B}^H$</td>
</tr>
<tr>
<td>$AB^H$</td>
<td>$AB^H$</td>
</tr>
<tr>
<td>$A^TB^H$</td>
<td>$A^T\bar{B}^H$</td>
</tr>
</tbody>
</table>
Transposing or Conjugate Transposing:
This 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 81.

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” on page 436 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 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 when the AltiVec or VSX unit is not used.
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 64.

For Large Matrices
If you are using large square matrices in your matrix multiplication operations, you may 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 [40 on page 1365] and [43 on page 1365].
Matrix Operation Subroutines

This contains the matrix operation subroutine descriptions.
SGEADD, DGEADD, CGEADD, and ZGEADD (Matrix Addition for General Matrices or Their Transposes)

Purpose

These subroutines can perform any one of the following matrix additions, using matrices $A$ and $B$ or their transposes, and matrix $C$:

\[ C + A + B \]
\[ C + A^T + B \]
\[ C + A + B^T \]
\[ C + A^T + B^T \]

**Table 112. Data Types**

<table>
<thead>
<tr>
<th>$A$, $B$, $C$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGEADD</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGEADD</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGEADD</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGEADD</td>
</tr>
</tbody>
</table>

**Note:** On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

**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);
```

**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 112, 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: an integer; $lda > 0$ and:

If $transa = 'N'$, $lda \geq m$.

If $transa = 'T'$, $lda \geq n$. 
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', $A^T$ 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', $B^T$ is used in the computation, and B has n rows and m columns.

Note: No data should be moved to form $B^T$; 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 112 on page 434 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: an 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', $B^T$ 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: an integer; ldc > 0 and ldc $\geq$ m.

m
is the number of rows in matrix C.
Specified as: an integer; 0 $\leq$ m $\leq$ ldc.

n
is the number of columns in matrix C.
Specified as: an 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 112 on page 434

Notes
1. All subroutines accept lowercase letters for the transa and transb arguments.
Matrix $C$ must have no common elements with matrices $A$ or $B$. However, $C$ may (exactly) coincide with $A$ if $\text{transa} = 'N'$, and $C$ may (exactly) coincide with $B$ if $\text{transb} = 'N'$. Otherwise, results are unpredictable. See “Concepts” on page 75.

**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 $C ← A + B$
- $c_{ij} = a_{ij} + b_{ji}$ for $C ← A + B^T$
- $c_{ij} = a_{ji} + b_{ij}$ for $C ← A^T + B$
- $c_{ij} = a_{ji} + b_{ji}$ for $C ← A^T + B^T$

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 $C^T$ of each of the four computations listed under “Function” by using the following matrix identities:

- $(A+B)^T = A^T + B^T$
- $(A+B^T)^T = A^T + B$
- $(A^T + B)^T = A + B^T$
- $(A^T + B^T)^T = A + B$

Be careful that your output array receiving $C^T$ has dimensions large enough to hold the transposed matrix. See Example 4.

**Error conditions**

**Input-Argument Errors**

1. $\text{lda}$, $\text{ldb}$, $\text{ldc} ≤ 0$
2. $m$, $n < 0$
3. $m > \text{ldc}$
4. $\text{transa}$, $\text{transb} ≠ 'N'$ or 'T'
5. $\text{transa} = 'N'$ and $m > \text{lda}$
6. $\text{transa} = 'T'$ and $n > \text{lda}$
7. $\text{transb} = 'N'$ and $m > \text{ldb}$
8. $\text{transb} = 'T'$ and $n > \text{ldb}$

**Examples**

**Example 1**

This example shows the computation $C ← A + B$, 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:

```
CALL SGEADD( A , 6 , 'N' , B , 4 , 'N' , C , 5 , 4 , 3 )
```
Example 2

This example shows the computation $C=A^T+B$, where $A$, $B$, and $C$ are the same size as arrays $A$, $B$, and $C$, in which they are contained.

Call Statement and Input:

```
CALL SGEADD( A, 3, 'T', B, 4, 'N', C, 4, 4, 3 )
```

Output:

```
A =

\[
\begin{bmatrix}
110000.0 & 120000.0 & 130000.0 \\
210000.0 & 220000.0 & 230000.0 \\
310000.0 & 320000.0 & 330000.0 \\
410000.0 & 420000.0 & 430000.0 \\
\vdots & \vdots & \vdots \\
\end{bmatrix}
\]

B =

\[
\begin{bmatrix}
11.0 & 12.0 & 13.0 \\
21.0 & 22.0 & 23.0 \\
31.0 & 32.0 & 33.0 \\
41.0 & 42.0 & 43.0 \\
\end{bmatrix}
\]

C =

\[
\begin{bmatrix}
110011.0 & 120012.0 & 130013.0 \\
210021.0 & 220022.0 & 230023.0 \\
310031.0 & 320032.0 & 330033.0 \\
410041.0 & 420042.0 & 430043.0 \\
\vdots & \vdots & \vdots \\
\end{bmatrix}
\]

Example 3

This example shows computation $C=A+B^T$, 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:

```
CALL SGEADD( A, 5, 'N', B, 3, 'T', C, 4, 4, 3 )
```

Output:

```
C =

\[
\begin{bmatrix}
110011.0 & 120012.0 & 310013.0 \\
210021.0 & 220022.0 & 320023.0 \\
310031.0 & 320032.0 & 330033.0 \\
410041.0 & 420042.0 & 340043.0 \\
\end{bmatrix}
\]
Example 4

This example shows how to produce the transpose of the result of the computation performed in Example 3, \( C+A+B \), which uses the calling sequence:

\[
\text{CALL SGEADD( } A \text{, 5, 'N', } B \text{, 3, 'T', } C \text{, 4, 4, 3 })
\]

You instead code a calling sequence for \( C^T+A^T+B \), as shown below, where the resulting matrix \( C^T \) in the output array \( C^T \) is the transpose of the matrix in the output array \( C \) in Example 3. Note that the array \( C^T \) has dimensions large enough to receive the transposed matrix. For a description of all the matrix identities, see “Special Usage” on page 436.

Call Statement and Input:

\[
A \quad \text{LDA TRANS A} \quad B \quad \text{LDB TRANS B} \quad C \quad \text{LDC M N}
\]

\[
\text{CALL SGEADD( } A \text{, 5, 'T', } B \text{, 3, 'N', } C \text{, 4, 4, 3 })
\]

\[
A = \begin{bmatrix}
110000.0 & 120000.0 & 130000.0 \\
210000.0 & 220000.0 & 230000.0 \\
310000.0 & 320000.0 & 330000.0 \\
410000.0 & 420000.0 & 430000.0 \\
. & . & .
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
11.0 & 12.0 & 13.0 & 14.0 \\
21.0 & 22.0 & 23.0 & 24.0 \\
31.0 & 32.0 & 33.0 & 34.0
\end{bmatrix}
\]

Output:

\[
C = \begin{bmatrix}
110011.0 & 120021.0 & 130031.0 \\
210012.0 & 220022.0 & 230032.0 \\
310013.0 & 320023.0 & 330033.0 \\
410014.0 & 420024.0 & 430034.0
\end{bmatrix}
\]

Example 5

This example shows the computation \( C+A^T+B^T \), 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:
Example 6

This example shows the computation $C = A + B$, 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:

\[
A = \begin{bmatrix}
11.0 & 12.0 & 13.0 & 14.0 \\
21.0 & 22.0 & 23.0 & 24.0 \\
31.0 & 32.0 & 33.0 & 34.0 
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
1.0 & 5.0 \\
2.0 & 4.0 \\
3.0 & 3.0 \\
6.0 & 6.0 \\
\end{bmatrix}
\]

\[
C = \begin{bmatrix}
1100.0 & 1200.0 & 1300.0 & 1400.0 \\
2100.0 & 2200.0 & 2300.0 & 2400.0 \\
3100.0 & 3200.0 & 3300.0 & 3400.0 
\end{bmatrix}
\]
SGESUB, DGESUB, CGESUB, and ZGESUB (Matrix Subtraction for General Matrices or Their Transposes)

Purpose

These subroutines can perform any one of the following matrix subtractions, using matrices A and B or their transposes, and matrix C:

\[
\begin{align*}
C & \leftarrow A - B \\
C & \leftarrow A^T - B \\
C & \leftarrow A - B^T \\
C & \leftarrow A^T - B^T \\
\end{align*}
\]

Table 113. Data Types

<table>
<thead>
<tr>
<th>A, B, C</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGESUB</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGESUB</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGESUB</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGESUB</td>
</tr>
</tbody>
</table>

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

Fortran

```fortran
CALL SGESUB | DGESUB | CGESUB | ZGESUB (a, lda, transa, b, ldb, transb, c, ldc, m, n)
```

C and C++

```c
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 113, 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: an 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 $transa = 'T'$, $A^T$ 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 $transb = 'T'$, $B^T$ is used in the computation, and $B$ has $n$ rows and $m$ columns.

Note: No data should be moved to form $B^T$; 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 112 on page 434 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: an 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 $transb = 'T$', $B^T$ 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: an integer; $ldc > 0$ and $ldc \geq m$.

$m$
is the number of rows in matrix $C$.
Specified as: an integer; $0 \leq m \leq ldc$.

$n$
is the number of columns in matrix $C$.
Specified as: an 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 113 on page 440.

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 75.

**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:

- $c_{ij} = a_{ij} - b_{ij}$ for $C ← A - B$
- $c_{ij} = a_{ij} - b_{ji}$ for $C ← A - B^T$
- $c_{ij} = a_{ji} - b_{ij}$ for $C ← A^T - B$
- $c_{ij} = a_{ji} - b_{ji}$ for $C ← A^T - B^T$

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 $C^T$ of each of the four computations listed under “Function” by using the following matrix identities:

- $(A - B)^T = A^T - B^T$
- $(A - B^T)^T = A^T - B$
- $(A^T - B)^T = A - B^T$
- $(A^T - B^T)^T = A - B$

Be careful that your output array receiving $C^T$ has dimensions large enough to hold the transposed matrix. See Example 5.

**Error conditions**

**Computational Errors**
None

**Input-Argument Errors**

1. $lda$, $ldb$, $ldc \leq 0$
2. $m$, $n < 0$
3. $m > ldc$
4. $transa$, $transb \neq 'N'$ or 'T'
5. $transa = 'N'$ and $m > lda$
6. $transa = 'T'$ and $n > lda$
7. $transb = 'N'$ and $m > ldb$
8. $transb = 'T'$ and $n > ldb$

**Examples**

**Example 1**

This example shows the computation $C ← A - B$, 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:
Example 2

This example shows the computation $C \leftarrow A^\top - B$, where $A$, $B$, and $C$ are the same size as arrays $A$, $B$, and $C$, in which they are contained.

Call Statement and Input:

\[
\begin{align*}
A & = \begin{bmatrix}
110000.0 & 120000.0 & 130000.0 \\
210000.0 & 220000.0 & 230000.0 \\
310000.0 & 320000.0 & 330000.0 \\
410000.0 & 420000.0 & 430000.0 \\
\end{bmatrix} \\
B & = \begin{bmatrix}
-11.0 & -12.0 & -13.0 \\
-21.0 & -22.0 & -23.0 \\
-31.0 & -32.0 & -33.0 \\
-41.0 & -42.0 & -43.0 \\
\end{bmatrix}
\end{align*}
\]

Output:

\[
\begin{align*}
C & = \begin{bmatrix}
110011.0 & 120012.0 & 130013.0 \\
210021.0 & 220022.0 & 230023.0 \\
310031.0 & 320032.0 & 330033.0 \\
410041.0 & 420042.0 & 430043.0 \\
\end{bmatrix}
\end{align*}
\]

Example 3

This example shows computation $C \leftarrow A - B^\top$, 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:
Example 4

This example shows the computation $C \leftarrow A^T \cdot B^T$, 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:

```
```

<table>
<thead>
<tr>
<th>A</th>
<th>LDA TRANSA</th>
<th>B</th>
<th>LDB TRANSB</th>
<th>C</th>
<th>LDC</th>
<th>M</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>110000.0</td>
<td>120000.0</td>
<td>130000.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>210000.0</td>
<td>220000.0</td>
<td>230000.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>310000.0</td>
<td>320000.0</td>
<td>330000.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>410000.0</td>
<td>420000.0</td>
<td>430000.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$A = \begin{bmatrix}
110.0 & 12.0 & -13.0 & -14.0 \\
-21.0 & -22.0 & -23.0 & -24.0 \\
-31.0 & -32.0 & -33.0 & -34.0
\end{bmatrix}$

$B = \begin{bmatrix}
-11.0 & -12.0 & -13.0 & -14.0 \\
-21.0 & -22.0 & -23.0 & -24.0 \\
-31.0 & -32.0 & -33.0 & -34.0
\end{bmatrix}$

Output:

<table>
<thead>
<tr>
<th>C</th>
<th>LDC</th>
<th>M</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>110011.0</td>
<td>120021.0</td>
<td>130031.0</td>
<td></td>
</tr>
<tr>
<td>210012.0</td>
<td>220022.0</td>
<td>230032.0</td>
<td></td>
</tr>
<tr>
<td>310013.0</td>
<td>320023.0</td>
<td>330033.0</td>
<td></td>
</tr>
<tr>
<td>410014.0</td>
<td>420024.0</td>
<td>430034.0</td>
<td></td>
</tr>
</tbody>
</table>

Example 5

This example shows how to produce the transpose of the result of the computation performed in Example 4, $C \leftarrow A^T \cdot B^T$, which uses the calling sequence:

```
```

You instead code a calling sequence for $C^T \cdot A \cdot B$, as shown below, where the resulting matrix $C^T$ in the output array $C^T$ is the transpose of the matrix in the output array $C$ in Example 4. Note that the array $C^T$ has dimensions large enough to receive the transposed matrix. For a description of all the matrix identities, see “Special Usage” on page 442.
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.
SGEMUL, DGEMUL, CGEMUL, and ZGEMUL (Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes)

Purpose

SGEMUL and DGEMUL can perform any one of the following matrix multiplications, using matrices $A$ and $B$ or their transposes, and matrix $C$:

- $C \leftarrow AB$
- $C \leftarrow A^T B$
- $C \leftarrow A B^T$
- $C \leftarrow A^H B^T$

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 $C$:

- $C \leftarrow AB$
- $C \leftarrow A^T B$
- $C \leftarrow A B^T$
- $C \leftarrow A^H B^T$

Table 114. Data Types

<table>
<thead>
<tr>
<th>$A$, $B$, $C$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGEMUL</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGEMUL</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGEMUL</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGEMUL</td>
</tr>
</tbody>
</table>

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

Fortran
CALL SGEMUL \mid DGEMUL \mid CGEMUL \mid ZGEMUL (a, lda, transa, b, ldb, transb, c, ldc, l, m, n)

C and C++
sgemul \mid dgemul \mid cgemul \mid 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 114, 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) $l$. 

446 ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
**lda**

is the leading dimension of the array specified for a.

Specified as: an integer; \( lda \geq 1 \).

If \( transa = 'N' \), \( lda \geq l \).

If \( transa = 'T' \) or \( 'C' \), \( lda \geq m \).

**transa**

indicates the form of matrix \( A \) to use in the computation, where:

If \( transa = 'N' \), \( A \) is used in the computation.

If \( transa = 'T' \), \( A^T \) is used in the computation.

If \( transa = 'C' \), \( A^H \) 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 \( transb = 'T' \), \( B^T \) is used in the computation, and \( B \) has \( n \) rows and \( m \) columns.

If \( transb = 'C' \), \( B^H \) is used in the computation, and \( B \) has \( n \) rows and \( m \) columns.

**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 114 on page 446, 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: an integer; \( ldb > 0 \) and:

If \( transb = 'N' \), \( ldb \geq m \).

If \( transb = 'T' \) or \( 'C' \), \( ldb \geq n \).

**transb**

indicates the form of matrix \( B \) to use in the computation, where:

If \( transb = 'N' \), \( B \) is used in the computation.

If \( transb = 'T' \), \( B^T \) is used in the computation.

If \( transb = 'C' \), \( B^H \) 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: an integer; \( ldc > 0 \) and \( ldc \geq l \).

\( l \)

is the number of rows in matrix \( C \).
Specified as: an integer; \(0 \leq l \leq \text{ldc}\).

\(m\) has the following meaning, where:
- If \(\text{transa} = 'N'\), it is the number of columns in matrix \(A\).
- If \(\text{transa} = 'T'\) or \('C'\), it is the number of rows in matrix \(A\).

In addition:
- If \(\text{transb} = 'N'\), it is the number of rows in matrix \(B\).
- If \(\text{transb} = 'T'\) or \('C'\), it is the number of columns in matrix \(B\).

Specified as: an integer; \(m \geq 0\).

\(n\) is the number of columns in matrix \(C\).

Specified as: an integer; \(n \geq 0\).

**On Return**

\(c\) is the \(l\) by \(n\) matrix \(C\), containing the results of the computation. Returned as: an \(\text{ldc}\) by (at least) \(n\) numbers of the data type indicated in Table 114 on page 446.

**Notes**

1. All subroutines accept lowercase letters for the \(\text{transa}\) and \(\text{transb}\) arguments.
2. Matrix \(C\) must have no common elements with matrices \(A\) or \(B\); otherwise, results are unpredictable. See “Concepts” on page 75.

**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:
\[ c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj} \quad \text{for } C \leftarrow A B \]
\[ c_{ij} = \sum_{k=1}^{m} a_{ki} b_{kj} \quad \text{for } C \leftarrow A^T B \]
\[ c_{ij} = \sum_{k=1}^{m} \overline{a_{ki}} b_{kj} \quad \text{for } C \leftarrow A^H B \]
\[ c_{ij} = \sum_{k=1}^{m} a_{ik} b_{jk} \quad \text{for } C \leftarrow A B^T \]
\[ c_{ij} = \sum_{k=1}^{m} a_{ki} b_{jk} \quad \text{for } C \leftarrow A^T B^T \]
\[ c_{ij} = \sum_{k=1}^{m} \overline{a_{ki}} b_{jk} \quad \text{for } C \leftarrow A^H B^T \]
\[ c_{ij} = \sum_{k=1}^{m} a_{ik} \overline{b}_{jk} \quad \text{for } C \leftarrow A B^H \]
\[ c_{ij} = \sum_{k=1}^{m} a_{ki} \overline{b}_{jk} \quad \text{for } C \leftarrow A^T B^H \]
\[ c_{ij} = \sum_{k=1}^{m} \overline{a_{ki}} \overline{b}_{jk} \quad \text{for } C \leftarrow A^H B^H \]

for \( i = 1, l \) and \( j = 1, n \)

See reference [46 on page 1366]. 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 \( C^T \) or the conjugate transpose \( C^H \) of some of the computations performed by these subroutines:

**Transpose**

\[
(AB)^T = B^T A^T \\
(A^T B)^T = B^T A \\
(AB^T)^T = BA^T \\
(A^T B^T)^T = BA
\]

**Conjugate Transpose**

\[
(AB)^H = B^H A^H \\
(A^H B)^H = B^H A \\
(AB^H)^H = BA^H \\
(A^H B^H)^H = BA
\]

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 \( C^T \) or \( C^H \), has dimensions large enough to hold the resulting transposed or conjugate transposed matrix. See Example 2 and Example 4.
Error conditions

Resource Errors
Unable to allocate internal 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 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 lda > l
8. transb = 'N' and m > ldb
9. transb = 'T' or 'C' and n > ldb

Examples

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:

\[
\begin{array}{cccccccc}
A & LDA & TRANS & B & LDB & TRANSB & C & ldc & l & m & n \\
\hline
1.0 & 2.0 & -1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 2.0 \\
2.0 & 0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 2.0 \\
1.0 & -1.0 & -1.0 & 1.0 & 1.0 & 2.0 & -1.0 & 0.0 \\
-3.0 & 2.0 & 2.0 & 2.0 & 2.0 & 0.0 & -2.0 & 1.0 \\
4.0 & 0.0 & -2.0 & 1.0 & -1.0 & -1.0 & 1.0 & 1.0 \\
-1.0 & -1.0 & 1.0 & -3.0 & 2.0 & -1.0 & 1.0 & -1.0 \\
\end{array}
\]

A =

\[
\begin{bmatrix}
1.0 & -1.0 & 0.0 & 2.0 \\
2.0 & 2.0 & -1.0 & -2.0 \\
1.0 & 0.0 & -1.0 & 1.0 \\
-3.0 & -1.0 & 1.0 & -1.0 \\
4.0 & 2.0 & -1.0 & 1.0 \\
\end{bmatrix}
\]

B =

Output:

\[
\begin{array}{cccc}
23.0 & 12.0 & -6.0 & 2.0 \\
-4.0 & -5.0 & 1.0 & 3.0 \\
3.0 & 0.0 & 1.0 & 4.0 \\
-3.0 & 5.0 & -2.0 & -10.0 \\
-5.0 & -7.0 & 4.0 & 4.0 \\
15.0 & 6.0 & -5.0 & 6.0 \\
\hline
\end{array}
\]

C =

Example 2
This example shows how to produce the transpose of the result of the computation performed in Example 1. \( C \leftarrow AB \), which uses the calling sequence:

\[
\text{CALL SGEOM}(A, 8, 'N', B, 6, 'N', C, 7, 6, 5, 4)
\]

You instead code a calling sequence for \( C^T \leftarrow B^T A^T \), as shown below, where the resulting matrix \( C^T \) in the output array \( C \) is the transpose of the matrix in the output array \( C \) in Example 1. Note that the array \( C^T \) has dimensions large enough to receive the transposed matrix. For a description of all the matrix identities, see “Special Usage” on page 449.

Call Statement and Input:

\[
\begin{align*}
A &\ LDA \ TRANS\ A & B & \ LDB \ TRANS\ B & C & \ LDC & L & M & N \\
\text{CALL SGEOM}( & B, 6, 'T', A, 8, 'T', & CT, 5, 4, 5, 6) \\
\end{align*}
\]

\[
\begin{bmatrix}
1.0 & -1.0 & 0.0 & 2.0 \\
2.0 & 2.0 & -1.0 & -2.0 \\
1.0 & 0.0 & -1.0 & 1.0 \\
-3.0 & -1.0 & 1.0 & -1.0 \\
4.0 & 2.0 & -1.0 & 1.0 \\
\end{bmatrix}
\]

\[
B =
\begin{bmatrix}
1.0 & 2.0 & -1.0 & -1.0 & 4.0 \\
2.0 & 0.0 & 1.0 & 1.0 & -1.0 \\
1.0 & -1.0 & -1.0 & 1.0 & 2.0 \\
-3.0 & 2.0 & 2.0 & 2.0 & 0.0 \\
4.0 & 0.0 & -2.0 & 1.0 & -1.0 \\
-1.0 & -1.0 & 1.0 & -3.0 & 2.0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
1.0 & -3.0 & 2.0 \\
2.0 & 4.0 & 0.0 \\
1.0 & -1.0 & -1.0 \\
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
23.0 & -4.0 & 3.0 & -3.0 & -5.0 & 15.0 \\
12.0 & -5.0 & 0.0 & 5.0 & -7.0 & 6.0 \\
-6.0 & 1.0 & 1.0 & -2.0 & 4.0 & -5.0 \\
2.0 & 3.0 & 4.0 & -10.0 & 4.0 & 6.0 \\
\end{bmatrix}
\]

\[
CT =
\begin{bmatrix}
23.0 & -4.0 & 3.0 & -3.0 & -5.0 & 15.0 \\
12.0 & -5.0 & 0.0 & 5.0 & -7.0 & 6.0 \\
-6.0 & 1.0 & 1.0 & -2.0 & 4.0 & -5.0 \\
2.0 & 3.0 & 4.0 & -10.0 & 4.0 & 6.0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
23.0 & -4.0 & 3.0 & -3.0 & -5.0 & 15.0 \\
12.0 & -5.0 & 0.0 & 5.0 & -7.0 & 6.0 \\
-6.0 & 1.0 & 1.0 & -2.0 & 4.0 & -5.0 \\
2.0 & 3.0 & 4.0 & -10.0 & 4.0 & 6.0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
23.0 & -4.0 & 3.0 & -3.0 & -5.0 & 15.0 \\
12.0 & -5.0 & 0.0 & 5.0 & -7.0 & 6.0 \\
-6.0 & 1.0 & 1.0 & -2.0 & 4.0 & -5.0 \\
2.0 & 3.0 & 4.0 & -10.0 & 4.0 & 6.0 \\
\end{bmatrix}
\]

Example 3

This example shows the computation \( C \leftarrow A^T \), where \( A \) and \( C \) are contained in larger arrays \( A \) and \( C \), respectively, and \( B \) is the same size as the

Call Statement and Input:

\[
\begin{align*}
A &\ LDA \ TRANS\ A & B & \ LDB \ TRANS\ B & C & \ LDC & L & M & N \\
\text{CALL SGEOM}( & A, 4, 'T', B, 3, 'N', & C, 5, 3, 3, 6) \\
\end{align*}
\]

\[
\begin{bmatrix}
1.0 & -3.0 & 2.0 \\
2.0 & 4.0 & 0.0 \\
1.0 & -1.0 & -1.0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
1.0 & -3.0 & 2.0 & 2.0 & -1.0 & 2.0 \\
2.0 & 4.0 & 0.0 & 0.0 & 1.0 & -2.0 \\
1.0 & -1.0 & -1.0 & -1.0 & 1.0 \\
\end{bmatrix}
\]

Output:
Example 4

This example shows how to produce the transpose of the result of the computation performed in Example 3, \( C^T A^T B \), which uses the calling sequence:

```fortran
CALL SGEMUL (A, 4, 'T', B, 3, 'N', C, 5, 3, 3, 6)
```

You instead code the calling sequence for \( C^T B^T A \), 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 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 449.

Call Statement and Input:

```fortran
CALL SGEMUL( B, 3, 'T', A, 4, 'N', CT, 8, 6, 3, 3)
```

Output:

\[
\begin{bmatrix}
6.0 & 4.0 & 1.0
\end{bmatrix}
\begin{bmatrix}
4.0 & 26.0 & -5.0
\end{bmatrix}
\begin{bmatrix}
1.0 & -5.0 & 5.0
\end{bmatrix}
\begin{bmatrix}
6.0 & 4.0 & 1.0 & 0.0 & -1.0
4.0 & 26.0 & -5.0 & -5.0 & 8.0 & -15.0
1.0 & -5.0 & 5.0 & -1.0 & 3.0
\end{bmatrix}
\]

Example 5

This example shows the computation \( C^T A B^T \), 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:

```fortran
CALL SGEMUL( A, 4, 'N', B, 3, 'T', C, 5, 3, 2, 3)
```

Output:

\[
\begin{bmatrix}
1.0 & -3.0
2.0 & 4.0
1.0 & -1.0
\end{bmatrix}
\begin{bmatrix}
6.0 & 4.0 & 1.0
4.0 & 26.0 & -5.0
1.0 & -5.0 & 5.0
\end{bmatrix}
\begin{bmatrix}
1.0 & -5.0 & 5.0
0.0 & 8.0 & -1.0
-1.0 & -15.0 & 3.0
\end{bmatrix}
\begin{bmatrix}
1.0 & -5.0 & 5.0 & 0.0 & 8.0 & -1.0 & -1.0 & -15.0 & 3.0
\end{bmatrix}
\]

452 ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
\[
B = \begin{bmatrix}
1.0 & -3.0 \\
2.0 & 4.0 \\
1.0 & -1.0
\end{bmatrix}
\]

Output:
\[
C = \begin{bmatrix}
10.0 & -10.0 & 4.0 \\
-10.0 & 20.0 & -2.0 \\
4.0 & -2.0 & 2.0
\end{bmatrix}
\]

**Example 6**

This example shows the computation \(C = A^T B^T\), 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:**
\[
\text{CALL SGEMUL(} A, 3, 'T', B, 3, 'T', C, 1, 1, 3, 3 )
\]

\[
A = \begin{bmatrix}
1.0 \\
2.0 \\
1.0
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
1.0 & -3.0 & 2.0 \\
2.0 & 4.0 & 0.0 \\
1.0 & -1.0 & -1.0
\end{bmatrix}
\]

Output:
\[
B = \begin{bmatrix}
-3.0 & 10.0 & -2.0
\end{bmatrix}
\]

**Example 7**

This example shows the computation \(C = A^T B\) using complex data, where \(A\), \(B\), and \(C\) are contained in larger arrays \(A\), \(B\), and \(C\), respectively.

**Call Statement and Input:**
\[
\text{CALL CGEMUL(} A, 6, 'T', B, 7, 'N', C, 3, 2, 3, 3 )
\]

\[
A = \begin{bmatrix}
(1.0, 2.0) & (3.0, 4.0) \\
(4.0, 6.0) & (7.0, 1.0) \\
(6.0, 3.0) & (2.0, 5.0)
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
(1.0, 9.0) & (2.0, 6.0) & (5.0, 6.0) \\
(2.0, 5.0) & (6.0, 2.0) & (6.0, 4.0) \\
(2.0, 6.0) & (5.0, 4.0) & (2.0, 6.0)
\end{bmatrix}
\]
Example 8

This example shows the computation $C := AB^H$ 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:

\[
\begin{align*}
A & = \begin{bmatrix}
1.0 & 2.0 & -3.0 & 2.0 \\
2.0 & 6.0 & 4.0 & 5.0 \\
1.0 & 2.0 & -1.0 & 8.0 \\
\cdot & \cdot & \cdot & \cdot
\end{bmatrix} \\
B & = \begin{bmatrix}
1.0 & 3.0 & -3.0 & 2.0 \\
2.0 & 5.0 & 4.0 & 6.0 \\
1.0 & 1.0 & -1.0 & 9.0 \\
\end{bmatrix}
\end{align*}
\]

Output:

\[
C = \begin{bmatrix}
20.0 & -1.0 & 12.0 & 25.0 & 24.0 & 26.0 \\
18.0 & -23.0 & 80.0 & -2.0 & 49.0 & -37.0 \\
26.0 & -23.0 & 56.0 & 37.0 & 76.0 & 2.0 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{bmatrix}
\]
SGEMMS, DGEMMS, CGEMMS, and ZGEMMS (Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes Using Winograd's Variation of Strassen's Algorithm)

Purpose

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 $C$:

$C \leftarrow AB$

$C \leftarrow AB^T$

$C \leftarrow A^TB$

$C \leftarrow A^TB^T$

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 $C$:

$C \leftarrow AB$

$C \leftarrow AB^H$

$C \leftarrow A^TB$

$C \leftarrow A^TB^H$

$C \leftarrow A^HB$

$C \leftarrow A^HB^H$

Table 115. Data Types

<table>
<thead>
<tr>
<th>$A, B, C$</th>
<th>aux</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SGEMMS</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DGEMMS</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>CGEMMS</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>ZGEMMS</td>
</tr>
</tbody>
</table>

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);
```

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 115] 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: an 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', *A*<sup>T</sup> is used in the computation.
- If *transa* = 'C', *A*<sup>H</sup> 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', *B*<sup>T</sup> is used in the computation, and *B* has *n* rows and *m* columns.
- If *transb* = 'C', *B*<sup>H</sup> is used in the computation, and *B* has *n* rows and *m* columns.

**Note:** No data should be moved to form *B*<sup>T</sup> or *B*<sup>H</sup>; 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 115 on page 4557 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: an 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', *B*<sup>T</sup> is used in the computation.
- If *transb* = 'C', *B*<sup>H</sup> 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: an integer; *ldc* > 0 and *ldc* ≥ *l*.

**l**

is the number of rows in matrix *C*. 
Specified as: an integer; $0 \leq l \leq ldc$.

$m$ has the following meaning, where:
If $\text{transa} = \text{'}N\text{'}$, it is the number of columns in matrix $A$.
If $\text{transa} = \text{'}T\text{'}$ or $\text{'}C\text{'}$, it is the number of rows in matrix $A$.
In addition:
If $\text{transb} = \text{'}N\text{'}$, it is the number of rows in matrix $B$.
If $\text{transb} = \text{'}T\text{'}$ or $\text{'}C\text{'}$, it is the number of columns in matrix $B$.
Specified as: an integer; $m \geq 0$.

$n$ is the number of columns in matrix $C$.
Specified as: an 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 115 on page 455.

$naux$ is the size of the work area specified by $aux$—that is, the number of elements in $aux$.
Specified as: an 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 $naux \geq 0$.

Note:
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 458.

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 115 on page 455.
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 75.
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 51.

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 448.

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 [17 on page 1364].) 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 449, 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 \( C^T \) or \( C^H \), has dimensions large enough to hold the resulting transposed or conjugate transposed matrix. See Example 2 and Example 4.

Error conditions

Resource Errors
  Error 2015 is unrecoverable, \( naux = 0 \), and unable to allocate work area.

Computational Errors
  None

Input-Argument Errors
  1. \( lda, ldb, ldc \leq 0 \)
  2. \( l, m, n < 0 \)
  3. \( l > ldc \)
  4. \( transa, transb \neq 'N' \) or \( 'T' \) for SGEMMS and Dgemms
  5. \( transa, transb \neq 'N', 'T', \) 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 \( naux \) not equal to 0, and \( naux \) is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example shows the computation \( C \leftarrow AB \), 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] for SGEMUL.

Call Statement and Input:

\[
\begin{array}{cccccccccccc}
\text{A} & \text{LDA TRANS}A & \text{B} & \text{LDB TRANS}B & \text{C} & \text{LDC} & \text{L} & \text{M} & \text{N} & \text{AUX} & \text{NAUX} \\
\end{array}
\]

\[
\text{CALL SGEMMS( A, LDA, 'N', B, LDB, 'N', C, LDC, L, M, N, AUX, NAUX )}
\]

Example 2

This example shows the computation \( C \leftarrow AB^H \), 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] for CGEMUL.
Call Statement and Input:

\[
\begin{array}{cccccccccc}
A & \text{LDA} & \text{TRANSA} & B & \text{LDB} & \text{TRANSB} & C & \text{LDC} & L & M & N & \text{AUX} & \text{NAUX} \\
\mid & \mid & \mid & \mid & \mid & \mid & \mid & \mid & \mid & \mid & \mid & \mid & \mid \\
\end{array}
\]

\[
\text{CALL CGEMMS(} \ A, 4, 'N', B, 3, 'C', C, 4, 3, 2, 3, \text{AUX, 0) }
\]
SGEMM, DGEMM, CGEMM, and ZGEMM (Combined Matrix Multiplication and Addition for General Matrices, Their Transposes, or Conjugate Transposes)

Purpose

SGEMM and DGEMM can perform any one of the following combined matrix computations, using scalars $\alpha$ and $\beta$, matrices $A$ and $B$ or their transposes, and matrix $C$:

\[
\begin{align*}
C & \leftarrow \alpha AB + \beta C \\
C & \leftarrow \alpha A^T B + \beta C
\end{align*}
\]

CGEMM and ZGEMM can perform any one of the following combined matrix computations, using scalars $\alpha$ and $\beta$, matrices $A$ and $B$, their transposes or their conjugate transposes, and matrix $C$:

\[
\begin{align*}
C & \leftarrow \alpha AB + \beta C \\
C & \leftarrow \alpha A^T B + \beta C \\
C & \leftarrow \alpha A^H B + \beta C \\
C & \leftarrow \alpha A^T B^T + \beta C \\
C & \leftarrow \alpha A^H B^T + \beta C
\end{align*}
\]

Table 116. Data Types

<table>
<thead>
<tr>
<th>$A$, $B$, $C$, $\alpha$, $\beta$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGEMM</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGEMM</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGEMM</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGEMM</td>
</tr>
</tbody>
</table>

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

Fortran

\[
\text{CALL SGEMM | DGEMM | CGEMM | ZGEMM (transa, transb, l, n, m, alpha, a, lda, b, ldb, beta, c, ldc)}
\]

C and C++

\[
\text{sgemm | dgemm | cgemm | zgemm (transa, transb, l, n, m, alpha, a, lda, b, ldb, beta, c, ldc);}
\]

CBLAS

\[
\text{cblas_sgemm | cblas_dgemm | cblas_cgemm | cblas_zgemm (cblas_layout, cblas_transa, cblas_transb, l, n, m, alpha, a, lda, b, ldb, beta, c, ldc);}
\]

On Entry

\textit{cblas\_layout}

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If $\textit{cblas\_layout} = \text{CblasRowMajor}$, the matrices are stored in row major order.
- If $\textit{cblas\_layout} = \text{CblasColMajor}$, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS\_LAYOUT. It must be CblasRowMajor or CblasColMajor.
**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; $transa = 'N'$, 'T', or 'C'.

**cblas_transa**
indicates the form of matrix $A$ to use in the computation, where:

If $cblas_transa = CblasNoTrans$, $A$ is used in the computation.

If $cblas_transa = CblasTrans$, $A^T$ is used in the computation.

If $cblas_transa = CblasConjTrans$, $A^H$ is used in the computation.

Specified as: an object of enumerated type CBLAS_TRANSPOSE. It must be CblasNoTrans, CblasTrans, or CblasConjTrans.

**transb**
indicates the form of matrix $B$ to use in the computation, where:

If $transb = 'N'$, $B$ is used in the computation.

If $transb = 'T'$, $B^T$ is used in the computation.

If $transb = 'C'$, $B^H$ is used in the computation.

Specified as: a single character; $transb = 'N'$, 'T', or 'C'.

**cblas_transb**
indicates the form of matrix $B$ to use in the computation, where:

If $cblas_transb = CblasNoTrans$, $B$ is used in the computation.

If $cblas_transb = CblasTrans$, $B^T$ is used in the computation.

If $cblas_transb = CblasConjTrans$, $B^H$ is used in the computation.

Specified as: an object of enumerated type CBLAS_TRANSPOSE. It must be CblasNoTrans, CblasTrans, or CblasConjTrans.

**l**
is the number of rows in matrix $C$.

Specified as: an integer; $0 \leq l \leq ldc$.

**n**
is the number of columns in matrix $C$.

Specified as: an 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: an integer; $m \geq 0$.

**alpha**
is the scalar $\alpha$.

Specified as: a number of the data type indicated in Table 116 on page 461.
$a$ is the matrix $A$, where:

- If $\text{transa} = 'N'$, $A$ is used in the computation, and $A$ has $l$ rows and $m$ columns.
- If $\text{transa} = 'T'$, $A^T$ is used in the computation, and $A$ has $m$ rows and $l$ columns.
- If $\text{transa} = 'C'$, $A^H$ 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 116 on page 461 where:

- If $\text{transa} = 'N'$, its size must be $\text{lda}$ by (at least) $m$.
- If $\text{transa} = 'T'$ or 'C', its size must be $\text{lda}$ by (at least) $l$.

$\text{lda}$

is the leading dimension of the array specified for $a$.

Specified as: an integer; $\text{lda} > 0$ and:

- If $\text{transa} = 'N'$, $\text{lda} \geq l$.
- If $\text{transa} = 'T'$ or 'C', $\text{lda} \geq m$.

$b$ is the matrix $B$, where:

- If $\text{transb} = 'N'$, $B$ is used in the computation, and $B$ has $m$ rows and $n$ columns.
- If $\text{transb} = 'T$', $B^T$ is used in the computation, and $B$ has $n$ rows and $m$ columns.
- If $\text{transb} = 'C$', $B^H$ is used in the computation, and $B$ has $n$ rows and $m$ columns.

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 116 on page 461 where:

- If $\text{transb} = 'N'$, its size must be $\text{ldb}$ by (at least) $n$.
- If $\text{transb} = 'T'$ or 'C', its size must be $\text{ldb}$ by (at least) $m$.

$\text{ldb}$

is the leading dimension of the array specified for $b$.

Specified as: an integer; $\text{ldb} > 0$ and:

- If $\text{transb} = 'N'$, $\text{ldb} \geq m$.
- If $\text{transb} = 'T'$ or 'C', $\text{ldb} \geq n$.

$\beta$ is the scalar $\beta$.

Specified as: a number of the data type indicated in Table 116 on page 461.

$c$ is the $l$ by $n$ matrix $C$.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 116 on page 461.

$\text{ldc}$

is the leading dimension of the array specified for $c$.

Specified as: an integer; $\text{ldc} > 0$ and $\text{ldc} \geq l$. 

Chapter 9. Matrix Operations  463
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 116 on page 461

Notes

1. All subroutines accept lowercase letters for the \textit{transa} and \textit{transb} arguments.
2. For SGEMM and DGEMM, if you specify 'C' for the \textit{transa} or \textit{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 75.

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:

\[
c_{ij} = \begin{cases} 
\alpha \sum_{k=1}^{m} a_{ik} b_{kj} + \beta c_{ij} & \text{for } C \leftarrow A B + \beta C \\
\alpha \sum_{k=1}^{m} a_{ki} b_{kj} + \beta c_{ij} & \text{for } C \leftarrow A^T B + \beta C \\
\alpha \sum_{k=1}^{m} a_{ki} b_{kj} + \beta c_{ij} & \text{for } C \leftarrow A^T B^T + \beta C \\
\alpha \sum_{k=1}^{m} a_{ki} b_{tk} + \beta c_{ij} & \text{for } C \leftarrow A B^T + \beta C \\
\alpha \sum_{k=1}^{m} a_{ki} b_{kj} + \beta c_{ij} & \text{for } C \leftarrow A^T B^T + \beta C \\
\alpha \sum_{k=1}^{m} a_{ki} b_{tk} + \beta c_{ij} & \text{for } C \leftarrow A B^H + \beta C \\
\alpha \sum_{k=1}^{m} a_{ki} b_{kj} + \beta c_{ij} & \text{for } C \leftarrow A^T B^H + \beta C \\
\alpha \sum_{k=1}^{m} a_{ki} b_{tk} + \beta c_{ij} & \text{for } C \leftarrow A B^H + \beta C \\
\alpha \sum_{k=1}^{m} a_{ki} b_{tk} + \beta c_{ij} & \text{for } C \leftarrow A^T B^H + \beta C \\
\end{cases}
\]

for \( i = 1, l \) and \( j = 1, n \)

See references [40 on page 1365] and [46 on page 1366]. In the following three cases, no computation is performed:

\bullet \ l \ is \ 0.
- $n$ is 0.
- $\beta$ is 1 and $\alpha$ is 0.

Assuming the above conditions do not exist, if $\beta \neq 1$ and $m$ is 0, then $\beta C$ is returned.

**Special Usage**

**Equivalence Rules**

The equivalence rules, defined for matrix multiplication of $A$ and $B$ in “Special Usage” on page 449, 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.

**Error conditions**

**Resource Errors**

Unable to allocate internal work area.

**Computational Errors**

None

**Input-Argument Errors**

1. $\text{cblas\_layout} \neq \text{CblasRowMajor} \text{ or CblasColMajor}$
2. $\text{lda}, \text{ldb}, \text{ldc} \leq 0$
3. $l, m, n < 0$
4. $l > \text{ldc}$
5. $\text{transa}, \text{transb} \neq \text{'N'}, \text{'T'}, \text{ or 'C'}$
6. $\text{transa} = \text{'N'}$ and $l > \text{lda}$
7. $\text{transa} = \text{'T'}$ or $\text{'C'}$ and $m > \text{lda}$
8. $\text{cblas\_transa} \neq \text{CblasNoTrans, CblasTrans, or CblasConjTrans}$
9. $\text{cblas\_transa} = \text{CblasNoTrans and } l > \text{lda}$
10. $\text{cblas\_transa} = \text{CblasTrans, or CblasConjTrans and } m > \text{lda}$
11. $\text{transb} = \text{'N'}$ and $m > \text{ldb}$
12. $\text{transb} = \text{'T'}$ or $\text{'C'}$ and $n > \text{ldb}$
13. $\text{cblas\_transb} \neq \text{CblasNoTrans, CblasTrans, or CblasConjTrans}$
14. $\text{cblas\_transb} = \text{CblasNoTrans and } m > \text{ldb}$
15. $\text{cblas\_transb} = \text{CblasTrans, or CblasConjTrans and } n > \text{ldb}$

**Examples**

**Example 1**

This example shows the computation $C + \alpha AB + \beta C$, where $A$, $B$, and $C$ 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 )
```
\[
A = \begin{bmatrix}
1.0 & 2.0 & -1.0 & -1.0 & 4.0 \\
2.0 & 0.0 & 1.0 & 1.0 & -1.0 \\
1.0 & -1.0 & -1.0 & 1.0 & 2.0 \\
-3.0 & 2.0 & 2.0 & 2.0 & 0.0 \\
4.0 & 0.0 & -2.0 & 1.0 & -1.0 \\
-1.0 & -1.0 & 1.0 & -3.0 & 2.0 \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
1.0 & -1.0 & 0.0 & 2.0 \\
2.0 & 2.0 & -1.0 & -2.0 \\
1.0 & 0.0 & -1.0 & 1.0 \\
-3.0 & -1.0 & 1.0 & -1.0 \\
4.0 & 2.0 & -1.0 & 1.0 \\
\end{bmatrix}
\]

\[
C = \begin{bmatrix}
0.5 & 0.5 & 0.5 & 0.5 \\
0.5 & 0.5 & 0.5 & 0.5 \\
0.5 & 0.5 & 0.5 & 0.5 \\
\end{bmatrix}
\]

Output:
\[
C = \begin{bmatrix}
24.0 & 13.0 & -5.0 & 3.0 \\
-3.0 & -4.0 & 2.0 & 4.0 \\
4.0 & 1.0 & 2.0 & 5.0 \\
-2.0 & 6.0 & -1.0 & -9.0 \\
-4.0 & -6.0 & 5.0 & 5.0 \\
16.0 & 7.0 & -4.0 & 7.0 \\
\end{bmatrix}
\]

Example 2
This example shows the computation \(C \leftarrow \alpha AB^T + \beta C\), 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:
\[
\begin{array}{cccccccccccc}
\text{TRANSA} & \text{TRANSB} & L & N & M & \text{ALPHA} & A & \text{LDA} & B & \text{LDB} & \text{BETA} & C & \text{LDC}
\end{array}
\]
\[
\text{CALL SGEMM('N', 'T', 3, 3, 2, 1.0, A, 4, B, 3, 2.0, C, 5)}
\]

\[
A = \begin{bmatrix}
1.0 & -3.0 \\
2.0 & 4.0 \\
1.0 & -1.0 \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
1.0 & -3.0 \\
2.0 & 4.0 \\
1.0 & -1.0 \\
\end{bmatrix}
\]

\[
C = \begin{bmatrix}
0.5 & 0.5 & 0.5 \\
0.5 & 0.5 & 0.5 \\
\end{bmatrix}
\]
\[
C = \begin{bmatrix}
0.5 & 0.5 & 0.5 \\
: & : & : \\
: & : & :
\end{bmatrix}
\]

Output:
\[
C = \begin{bmatrix}
11.0 & -9.0 & 5.0 \\
-9.0 & 21.0 & -1.0 \\
5.0 & -1.0 & 3.0 \\
: & : & :
\end{bmatrix}
\]

**Example 3**

This example shows the computation \( C \leftarrow \alpha AB + \beta C \) using complex data, where \( A \), \( B \), and \( C \) are contained in larger arrays, \( A \), \( B \), and \( C \), respectively.

**Call Statement and Input:**

\[
\text{CALL CGEMM( 'N', 'N', 6, 2, 3, ALPHA, A, 8, B, 4, BETA, C, 8 )}
\]

\[
\text{ALPHA} = (1.0, 0.0) \\
\text{BETA} = (2.0, 0.0)
\]

\[
A = \begin{bmatrix}
(1.0, 5.0) & (9.0, 2.0) & (1.0, 9.0) \\
(2.0, 4.0) & (8.0, 3.0) & (1.0, 8.0) \\
(3.0, 3.0) & (7.0, 5.0) & (1.0, 7.0) \\
(4.0, 2.0) & (4.0, 7.0) & (1.0, 5.0) \\
(5.0, 1.0) & (5.0, 1.0) & (1.0, 6.0) \\
(6.0, 6.0) & (3.0, 6.0) & (1.0, 4.0) \\
: & : & :
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
(1.0, 8.0) & (2.0, 7.0) \\
(4.0, 4.0) & (6.0, 8.0) \\
(6.0, 2.0) & (4.0, 5.0) \\
: & : & :
\end{bmatrix}
\]

\[
C = \begin{bmatrix}
(0.5, 0.0) & (0.5, 0.0) \\
(0.5, 0.0) & (0.5, 0.0) \\
(0.5, 0.0) & (0.5, 0.0) \\
(0.5, 0.0) & (0.5, 0.0) \\
: & : & :
\end{bmatrix}
\]

Output:
\[
C = \begin{bmatrix}
(-22.0, 113.0) & (-35.0, 142.0) \\
(-19.0, 114.0) & (-35.0, 141.0) \\
(-20.0, 119.0) & (-43.0, 146.0) \\
(-27.0, 110.0) & (-58.0, 131.0) \\
(6.0, 103.0) & (0.0, 112.0) \\
(-55.0, 116.0) & (-75.0, 135.0) \\
: & : & :
\end{bmatrix}
\]

**Example 4**
This example shows how to obtain the conjugate transpose of $AB^H$.

\[ (AB^H)^H = B^A = BA^H \]

This shows the conjugate transpose of the computation performed in Example 8 for CGEMUL, which uses the following calling sequence:

\[
\text{CALL CGEMUL(} A, 4, 'N', B, 3, 'C', C, 4, 3, 2, 3 \text{)}
\]

You instead code the calling sequence for $C^\beta C + \alpha BA^H$, where $\beta = 0$, $\alpha = 1$, and the array $C$ has the correct dimensions to receive the transposed matrix. Because $\beta$ is zero, $\beta C = 0$. For a description of all the matrix identities, see “Special Usage” on page 449.

Call Statement and Input:

\[
\text{TRANSA TRANSB } L \ N \ M \ \text{ALPHA} \ A \ \text{LDA} \ B \ \text{LDB} \ BETA \ C \ \text{LDC}
\]

\[
\text{CALL CGEMM(} 'N', 'C', 3, 3, 2, \text{ALPHA}, B, 3, A, 3, BETA, C, 4 \text{)}
\]

\[
\text{ALPHA} = (1.0, 0.0)
\]

\[
\text{BETA} = (0.0, 0.0)
\]

\[
B = \begin{bmatrix}
(1.0, 3.0) & (-3.0, 2.0) \\
(2.0, 5.0) & (4.0, 6.0) \\
(1.0, 1.0) & (-1.0, 9.0)
\end{bmatrix}
\]

\[
A = \begin{bmatrix}
(1.0, 2.0) & (-3.0, 2.0) \\
(2.0, 6.0) & (4.0, 5.0) \\
(1.0, 2.0) & (-1.0, 8.0)
\end{bmatrix}
\]

\[
C = (\text{not relevant})
\]

Output:

\[
C = \begin{bmatrix}
(20.0, 1.0) & (18.0, 23.0) & (26.0, 23.0) \\
(12.0, -25.0) & (80.0, 2.0) & (56.0, -37.0) \\
(24.0, -26.0) & (49.0, 37.0) & (76.0, -2.0)
\end{bmatrix}
\]

Example 5

This example shows the computation $C^\alpha A^T B^H + \beta C$ 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 $\beta$ 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:

\[
\text{TRANSA TRANSB } L \ N \ M \ \text{ALPHA} \ A \ \text{LDA} \ B \ \text{LDB} \ BETA \ C \ \text{LDC}
\]

\[
\text{CALL CGEMM(} 'T', 'C', 1, 3, 3, \text{ALPHA}, A, 3, B, 3, BETA, C, 1 \text{)}
\]

\[
\text{ALPHA} = (1.0, 1.0)
\]

\[
\text{BETA} = (0.0, 0.0)
\]

\[
\text{()} \text{()}
\]
\[
A = \begin{bmatrix}
(1.0, 2.0) \\
(2.0, 5.0) \\
(1.0, 6.0)
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
(1.0, 6.0) & (-3.0, 4.0) & (2.0, 6.0) \\
(2.0, 3.0) & (4.0, 6.0) & (0.0, 3.0) \\
(1.0, 3.0) & (-1.0, 6.0) & (-1.0, 9.0)
\end{bmatrix}
\]

\[C = \text{(not relevant)}\]

Output:

\[
C = \begin{bmatrix}
(86.0, 44.0) & (58.0, 70.0) & (121.0, 55.0)
\end{bmatrix}
\]
SSYMM, DSYMM, CSYMM, ZSYMM, CHEMM, and ZHEMM (Matrix-Matrix Product Where One Matrix is Real or Complex Symmetric or Complex Hermitian)

Purpose

These subroutines compute one of the following matrix-matrix products, using the scalars $\alpha$ and $\beta$ and matrices $A$, $B$, and $C$:

1. $C \leftarrow \alpha AB + \beta C$
2. $C \leftarrow \alpha BA + \beta C$

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 117. Data Types

<table>
<thead>
<tr>
<th>$\alpha$, $A$, $B$, $\beta$, $C$</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SSYMM</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DSYMM</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CSYMM and CHEMM</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZSYMM and ZHEMM</td>
</tr>
</tbody>
</table>

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

Fortran

```fortran
CALL SSYMM | DSYMM | CSYMM | ZSYMM | CHEMM | ZHEMM (side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc);
```

C and C++

```c
ssymm | dsymm | csymm | zsymm | chemm | zhemm (side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc);
```

CBLAS

```c
clblas_ssymm | clblas_dsymm | clblas_csymm | clblas_zsymm | clblas_chemm | clblas_zhemm (cblas_layout, cblas_side, cblas_uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc);
```

On Entry

- **cblas_layout**
  - indicates whether the input and output matrices are stored in row major order or column major order, where:
    - If `cblas_layout = CblasRowMajor`, the matrices are stored in row major order.
    - If `cblas_layout = CblasColMajor`, the matrices are stored in column major order.
  - Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

- **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'.

$cblas\_side$
indicates whether matrix $A$ is located to the left or right of rectangular matrix $B$ in the equation used for this computation, where:
If $cblas\_side = \text{CblasLeft}$, $A$ is to the left of $B$, resulting in equation 1.
If $cblas\_side = \text{CblasRight}$, $A$ is to the right of $B$, resulting in equation 2.
Specified as: an object of enumerated type CBLAS\_SIDE. It must be CblasLeft or CblasRight.

$uplo$
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'.

$cblas\_uplo$
indicates the storage mode used for matrix $A$, where:
If $cblas\_uplo = \text{CblasUpper}$, $A$ is stored in upper storage mode.
If $cblas\_uplo = \text{CblasLower}$, $A$ is stored in lower storage mode.
Specified as: an object of enumerated type CBLAS\_UPLO. It must be CblasUpper or CblasLower.

$m$
is the number of rows in rectangular matrices $B$ and $C$, and:
If $side = 'L'$, $m$ is the order of matrix $A$.
Specified as: an integer; $0 \leq m \leq \text{ldb}$, $m \leq \text{ldc}$, and:
If $side = 'L'$, $m \leq \text{lda}$.

$n$
is the number of columns in rectangular matrices $B$ and $C$, and:
If $side = 'R'$, $n$ is the order of matrix $A$.
Specified as: an integer; $n \geq 0$ and:
If $side = 'R'$, $n \leq \text{lda}$.

$alpha$
is the scalar $\alpha$.
Specified as: a number of the data type indicated in Table 117 on page 470.

$a$
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 117 on page 470, where:
If \( \text{side} = 'L' \), its size must be \( \text{lda} \) by (at least) \( m \).

If \( \text{side} = 'R' \), it size must be \( \text{lda} \) by (at least) \( n \).

\( \text{lda} \)

is the leading dimension of the array specified for \( a \).

Specified as: an integer; \( \text{lda} > 0 \) and:

If \( \text{side} = 'L' \), \( \text{lda} \geq m \).

If \( \text{side} = 'R' \), \( \text{lda} \geq n \).

\( b \)

is the \( m \) by \( n \) rectangular matrix \( B \).

Specified as: an \( \text{ldb} \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 117 on page 470.

\( \text{ldb} \)

is the leading dimension of the array specified for \( b \).

Specified as: an integer; \( \text{ldb} > 0 \) and \( \text{ldb} \geq m \).

\( \beta \)

is the scalar \( \beta \).

Specified as: a number of the data type indicated in Table 117 on page 470.

\( c \)

is the \( m \) by \( n \) rectangular matrix \( C \).

Specified as: an \( \text{ldc} \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 117 on page 470.

\( \text{ldc} \)

is the leading dimension of the array specified for \( c \).

Specified as: an integer; \( \text{ldc} > 0 \) and \( \text{ldc} \geq m \).

On Return

\( c \)

is the \( m \) by \( n \) matrix \( C \), containing the results of the computation.

Returned as: an \( \text{ldc} \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 117 on page 470.

Notes

1. These subroutines accept lowercase letters for the \( \text{side} \) and \( \text{uplo} \) arguments.

2. Matrices \( A \), \( B \), and \( C \) must have no common elements; otherwise, results are unpredictable.

3. If matrix \( A \) is upper triangular (\( \text{uplo} = 'U' \)), these subroutines use only the data in the upper triangular portion of the array. If matrix \( A \) is lower triangular, (\( \text{uplo} = 'L' \)), 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 85. For a description of how complex Hermitian matrices are stored in upper and lower storage mode, see "Complex Hermitian Matrix" on page 90.
**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. $C \leftarrow \alpha AB + \beta C$
2. $C \leftarrow \alpha BA + \beta C$

where:

- $\alpha$ and $\beta$ 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 [40 on page 1365] and [46 on page 1366]. In the following two cases, no computation is performed:

- $n$ or $m$ is 0.
- $\beta$ is one and $\alpha$ is zero.

**Error conditions**

**Resource Errors**

Unable to allocate internal work area.

**Computational Errors**

None

**Input-Argument Errors**

1. $\text{cblas_layout} \neq \text{CblasRowMajor or CblasColMajor}$
2. $m < 0$
3. $m > \text{ldb}$
4. $m > \text{ldc}$
5. $n < 0$
6. $\text{lda}$, $\text{ldb}$, $\text{ldc} \leq 0$
7. $\text{side} \neq \text{'L' or 'R'}$
8. $\text{side} = \text{'L'}$ and $m > \text{lda}$
9. $\text{side} = \text{'R'}$ and $n > \text{lda}$
10. $\text{cblas_side} \neq \text{CblasLeft or CblasRight}$
11. $\text{cblas_side} = \text{CblasLeft and m > lda}$
12. $\text{cblas_side} = \text{CblasRight and n > lda}$
13. $\text{uplo} \neq \text{'L' or 'U'}$
14. $\text{cblas_uplo} \neq \text{CblasLower or CblasUpper}$

**Examples**

**Example 1**

This example shows the computation $C \leftarrow \alpha AB + \beta C$, where $A$ is a real symmetric matrix of order 5, stored in upper storage mode, and $B$ and $C$ are 5 by 4 rectangular matrices.

Call Statement and Input:
Example 2

This example shows the computation $C ← \alpha AB + \beta C$, 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:

```
CALL SSYMM( 'L', 'U', 5, 4, 2.0, A, 8, B, 6, 1.0, C, 5 )
```

\[
A = \begin{bmatrix}
1.0 & 2.0 & -1.0 & -1.0 & 4.0 \\
0.0 & 1.0 & 1.0 & -1.0 \\
-1.0 & 1.0 & 2.0 \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
1.0 & -1.0 & 0.0 & 2.0 \\
2.0 & 2.0 & -1.0 & -2.0 \\
-3.0 & -1.0 & 1.0 & -1.0 \\
4.0 & 2.0 & -1.0 & 1.0 \\
\end{bmatrix}
\]

\[
C = \begin{bmatrix}
23.0 & 12.0 & -6.0 & 2.0 \\
-4.0 & -5.0 & 1.0 & 3.0 \\
5.0 & 6.0 & -1.0 & -4.0 \\
-4.0 & 1.0 & 0.0 & -5.0 \\
8.0 & -4.0 & -2.0 & 13.0 \\
\end{bmatrix}
\]

Output:

\[
C = \begin{bmatrix}
69.0 & 36.0 & -18.0 & 6.0 \\
-12.0 & -15.0 & 3.0 & 9.0 \\
15.0 & 18.0 & -3.0 & -12.0 \\
-12.0 & 3.0 & 0.0 & -15.0 \\
8.0 & -20.0 & -2.0 & 35.0 \\
\end{bmatrix}
\]
Example 3

This example shows the computation $C \leftarrow aBA + \beta C$, 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:

```
CALL SSYMM( 'R', 'U', 2, 3, 2.0, A, 4, B, 3, 1.0, C, 5 )
```

\[
A = \begin{bmatrix}
1.0 & -3.0 & 1.0 \\
 & 4.0 & -1.0 \\
 & & 2.0
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
1.0 & -3.0 & 3.0 \\
2.0 & 4.0 & -1.0 \\
 & & 
\end{bmatrix}
\]

\[
C = \begin{bmatrix}
13.0 & -18.0 & 10.0 \\
-11.0 & 11.0 & -4.0 \\
 & & 
\end{bmatrix}
\]

Output:

\[
C = \begin{bmatrix}
24.0 & 16.0 & 4.0 & 4.0 & 0.0 & -4.0 \\
36.0 & 44.0 & 20.0 & 20.0 & 12.0 & -20.0 \\
-8.0 & -24.0 & 12.0 & 12.0 & -4.0 & 12.0 \\
& & & & & \\
& & & & & \\
& & & & & 
\end{bmatrix}
\]

Example 4

This example shows the computation $C \leftarrow aBA + \beta C$, where $A$ is a real symmetric matrix of order 3, stored in lower storage mode, and $B$ and $C$ are 3 by 3 square matrices.

Call Statement and Input:

```
CALL SSYMM( 'R', 'L', 3, 3, -1.0, A, 3, B, 3, 1.0, C, 3 )
```

\[
C = \begin{bmatrix}
39.0 & -54.0 & 30.0 \\
-33.0 & 33.0 & -12.0 \\
 & & 
\end{bmatrix}
\]
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)

\[
A = \begin{bmatrix}
1.0 & . & . \\
2.0 & 10.0 & . \\
1.0 & 11.0 & 4.0 \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
1.0 & -3.0 & 2.0 \\
2.0 & 4.0 & 0.0 \\
1.0 & -1.0 & -1.0 \\
\end{bmatrix}
\]

\[
C = \begin{bmatrix}
1.0 & 5.0 & -9.0 \\
-3.0 & 10.0 & -2.0 \\
-2.0 & 8.0 & 0.0 \\
\end{bmatrix}
\]

Output:

\[
C = \begin{bmatrix}
4.0 & 11.0 & 15.0 \\
-13.0 & -34.0 & -48.0 \\
0.0 & 27.0 & 14.0 \\
\end{bmatrix}
\]
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:

```
CALL CHEMM( 'R', 'L', 2, 3, ALPHA, A, 4, B, 3, BETA, C, 5 )
```

```
ALPHA = (2.0, 3.0)
```

```
BETA = (1.0, 6.0)
```

```
A =
(1.0, )
(3.0, 2.0) (4.0, )
(-1.0, 6.0) (1.0, 4.0) (2.0, )
```

```
B =
(1.0, 1.0) (-3.0, 2.0) (3.0, 3.0)
(2.0, 6.0) (4.0, 5.0) (-1.0, 4.0)
```

```
C =
(13.0, 6.0) (-18.0, 6.0) (10.0, 7.0)
(-11.0, 8.0) (11.0, 1.0) (-4.0, 2.0)
```

Output:

```
C =
(-137.0, 17.0) (-158.0, -102.0) (-39.0, 141.0)
(-154.0, -77.0) (-63.0, 186.0) (159.0, 104.0)
```

Chapter 9. Matrix Operations  477
STRMM, DTRMM, CTRMM, and ZTRMM (Triangular Matrix-Matrix Product)

Purpose

STRMM and DTRMM compute one of the following matrix-matrix products, using the scalar $\alpha$, rectangular matrix $B$, and triangular matrix $A$ or its transpose:

1. $B \leftarrow \alpha AB$
2. $B \leftarrow \alpha A^T B$
3. $B \leftarrow \alpha BA$
4. $B \leftarrow \alpha BA^T$

CTRMM and ZTRMM compute one of the following matrix-matrix products, using the scalar $\alpha$, rectangular matrix $B$, and triangular matrix $A$, its transpose, or its conjugate transpose:

1. $B \leftarrow \alpha AB$
2. $B \leftarrow \alpha A^T B$
3. $B \leftarrow \alpha BA$
4. $B \leftarrow \alpha BA^T$
5. $B \leftarrow \alpha A^H B$
6. $B \leftarrow \alpha BA^H$

<table>
<thead>
<tr>
<th>$A$, $B$, $\alpha$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>STRMM</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DTRMM</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CTRMM</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZTRMM</td>
</tr>
</tbody>
</table>

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

Fortran

```fortran
CALL STRMM | DTRMM | CTRMM | ZTRMM (side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb))
```

C and C++

```c
strmm | dtrmm | ctrmm | ztrmm (side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb);
```

<table>
<thead>
<tr>
<th>CBLAS</th>
</tr>
</thead>
</table>
| cblas_strmm | cblas_dtrmm | cblas_ctrmm | cblas_ztrmm (cblas_layout, cblas_side, cblas_uplo, cblas_transa, cblas_diag, m, n, alpha, a, lda, b, ldb));

On Entry

`cblas_layout`

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If `cblas_layout` = CblasRowMajor, the matrices are stored in row major order.
- If `cblas_layout` = CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

`side`

indicates whether the triangular matrix $A$ is located to the left or right of rectangular matrix $B$ in the equation used for this computation, where:
If \texttt{side} = 'L', \(A\) is to the left of \(B\) in the equation, resulting in either equation 1, 2, or 5.

If \texttt{side} = 'R', \(A\) is to the right of \(B\) in the equation, resulting in either equation 3, 4, or 6.

Specified as: a single character. It must be 'L' or 'R'.

\texttt{cblas\_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 \texttt{cblas\_side} = \texttt{CblasLeft}, \(A\) is to the left of \(B\) in the equation, resulting in either equation 1, 2, or 5.

If \texttt{cblas\_side} = \texttt{CblasRight}, \(A\) is to the right of \(B\) in the equation, resulting in either equation 3, 4, or 6.

Specified as: an object of enumerated type \texttt{CBLAS\_SIDE}. It must be \texttt{CblasLeft} or \texttt{CblasRight}.

\texttt{uplo}

indicates whether matrix \(A\) is an upper or lower triangular matrix, where:

If \texttt{uplo} = 'U', \(A\) is an upper triangular matrix.

If \texttt{uplo} = 'L', \(A\) is a lower triangular matrix.

Specified as: a single character. It must be 'U' or 'L'.

\texttt{cblas\_uplo}

indicates whether matrix \(A\) is an upper or lower triangular matrix, where:

If \texttt{cblas\_uplo} = \texttt{CblasUpper}, \(A\) is an upper triangular matrix.

If \texttt{cblas\_uplo} = \texttt{CblasLower}, \(A\) is a lower triangular matrix.

Specified as: an object of enumerated type \texttt{CBLAS\_UPLO}. It must be \texttt{CblasUpper} or \texttt{CblasLower}.

\texttt{transa}

indicates the form of matrix \(A\) to use in the computation, where:

If \texttt{transa} = 'N', \(A\) is used in the computation, resulting in either equation 1 or 3.

If \texttt{transa} = 'T', \(A^T\) is used in the computation, resulting in either equation 2 or 4.

If \texttt{transa} = 'C', \(A^H\) is used in the computation, resulting in either equation 5 or 6.

Specified as: a single character. It must be 'N', 'T', or 'C'.

\texttt{cblas\_transa}

indicates the form of matrix \(A\) to use in the computation, where:

If \texttt{cblas\_transa} = \texttt{CblasNoTrans}, \(A\) is used in the computation, resulting in either equation 1 or 3.

If \texttt{cblas\_transa} = \texttt{CblasTrans}, \(A^T\) is used in the computation, resulting in either equation 2 or 4.

If \texttt{cblas\_transa} = \texttt{CblasConjTrans}, \(A^H\) is used in the computation, resulting in either equation 5 or 6.

Specified as: an object of enumerated type \texttt{CBLAS\_TRANSPOSE}. It must be \texttt{CblasNoTrans}, \texttt{CblasTrans}, or \texttt{CblasConjTrans}. 
**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'.

**cblas_diag**
indicates the characteristics of the diagonal of matrix A, where:
If `diag` = CblasUnit, A is a unit triangular matrix.
If `diag` = CblasNonUnit A is not a unit triangular matrix.
Specified as: an object of enumerated type CBLAS_DIAG. It must be CblasNonUnit or CblasUnit.

*m* is the number of rows in rectangular matrix B, and:
If `side` = 'L', m is the order of triangular matrix A.
Specified as: an integer, where:
If `side` = 'L', 0 ≤ m ≤ lda and m ≤ ldb.
If `side` = 'R', 0 ≤ m ≤ 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: an integer; n ≥ 0 and:
If `side` = 'R', n ≤ lda.

**alpha**
is the scalar α.
Specified as: a number of the data type indicated in Table 118 on page 478.

**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 $A^T$ 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 118 on page 478, 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: an integer; lda > 0 and:
If `side` = 'L', lda ≥ m.
If `side` = 'R', lda ≥ 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 118 on page 478](#).

`ldb` is the leading dimension of the array specified for `b`.

Specified as: an 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 118 on page 478](#).

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 93.

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. `B+αAB`
2. `B+αAᵀB`
3. `B+αAᴴB` (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. `B+αBA`
5. `B+αBAᵀ`
6. `B+αBAᴴ` (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 [40 on page 1365](#) and [46 on page 1366](#). 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. cblas_layout ≠ CblasRowMajor or CblasColMajor
2. m < 0
3. m > ldb
4. n < 0
5. lda, ldb ≤ 0
6. side ≠ 'L' or 'R'
7. side = 'L' and m > lda
8. side = 'R' and n > lda
9. cblas_side ≠ CblasLeft or CblasRight
10. cblas_side = CblasLeft and m > lda
11. cblas_side = CblasRight and n > lda
12. uplo ≠ 'L' or 'U'
13. cblas_uplo ≠ CblasLower or CblasUpper
14. transa ≠ 'T', 'N', or 'C'
15. cblas_transa ≠ CblasNoTrans, CblasTrans, or CblasConjTrans
16. diag ≠ 'N' or 'U'
17. cblas_diag ≠ CblasNonUnit or CblasUnit

Examples

Example 1

This example shows the computation \( B \leftarrow \alpha AB \), where \( A \) is a 5 by 5 upper triangular matrix that is not unit triangular, and \( B \) 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)
```

\[
\begin{bmatrix}
3.0 & -1.0 & 2.0 & 2.0 & 1.0 \\
-2.0 & 4.0 & -1.0 & 3.0 \\
. & -3.0 & 0.0 & 2.0 \\
. & . & . & . \\
. & . & . & . \\
. & . & . & . \\
\end{bmatrix}
\]

\[
A =
\begin{bmatrix}
2.0 & 3.0 & 1.0 \\
5.0 & 5.0 & 4.0 \\
0.0 & 1.0 & 2.0 \\
3.0 & 1.0 & -3.0 \\
-1.0 & 2.0 & 1.0 \\
. & . & .
\end{bmatrix}
\]

Output:
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:

$$
\text{CALL STRMM( 'L', 'U', 'T', 'N', 5, 4, 1.0, A, 7, B, 6 )}
$$

$$
\begin{bmatrix}
-1.0 & -4.0 & -2.0 & 2.0 & 3.0 \\
. & -2.0 & 2.0 & 2.0 & 2.0 \\
. & . & -3.0 & -1.0 & 4.0 \\
\end{bmatrix}
$$

$$
A = 
\begin{bmatrix}
. & . & 1.0 & 0.0 \\
. & . & . & -2.0 \\
. & . & . & . \\
. & . & . & . \\
\end{bmatrix}
$$

$$
\begin{bmatrix}
1.0 & 2.0 & 3.0 & 4.0 \\
3.0 & 3.0 & -1.0 & 2.0 \\
\end{bmatrix}
$$

$$
B = 
\begin{bmatrix}
-2.0 & -1.0 & 0.0 & 1.0 \\
4.0 & 4.0 & -3.0 & -3.0 \\
2.0 & 2.0 & 2.0 & 2.0 \\
\end{bmatrix}
$$

Output:

$$
\begin{bmatrix}
-1.0 & -2.0 & -3.0 & -4.0 \\
2.0 & -2.0 & -14.0 & -12.0 \\
10.0 & 5.0 & -8.0 & -7.0 \\
14.0 & 15.0 & 1.0 & 8.0 \\
-3.0 & 4.0 & 3.0 & 16.0 \\
\end{bmatrix}
$$

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:

$$
\text{CALL STRMM( 'R', 'L', 'N', 'N', 3, 5, 1.0, A, 7, B, 4 )}
$$

$$
\begin{bmatrix}
2.0 & . & . & . & . \\
2.0 & 3.0 & . & . & . \\
2.0 & 1.0 & 1.0 & . & . \\
\end{bmatrix}
$$

$$
A = 
\begin{bmatrix}
2.0 & . & . & . & . \\
2.0 & 3.0 & . & . & . \\
2.0 & 1.0 & 1.0 & . & . \\
0.0 & 3.0 & 0.0 & -2.0 & . \\
\end{bmatrix}
$$
Example 4
This example shows the computation $B \leftarrow \alpha BA$, where $A$ is a 6 by 6 upper triangular matrix that is unit triangular, and $B$ is a 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 elements.

**Call Statement and Input:**
```
CALL STRMM( 'R', 'U', 'N', 'U', 1, 6, 1.0, A, 7, B, 2)
```

**Output:**
```
B =
[ 10.0  4.0  0.0  0.0  1.0 ]
[ 10.0 14.0 -4.0  6.0 -3.0 ]
[ -8.0  2.0 -5.0  4.0 -2.0 ]
```

Example 5
This example shows the computation $B \leftarrow \alpha A^H B$, where $A$ is a 5 by 5 upper triangular matrix that is not unit triangular, and $B$ is a 5 by 1 rectangular matrix.

**Call Statement and Input:**
```
CALL CTRMM( 'L', 'U', 'C', 'N', 5, 1, ALPHA, A, 6, B, 6 )
```

**Output:**
```
B =
[ 1.0  2.0  1.0  3.0 -1.0 -2.0 ]
[ 1.0  4.0 -2.0 10.0  2.0 -6.0 ]
```

484  ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
\[
A = \begin{bmatrix}
(-4.0, 1.0) & (4.0, -3.0) & (-1.0, 3.0) & (0.0, 0.0) & (-1.0, 0.0) \\
\cdot & (-2.0, 0.0) & (-3.0, -1.0) & (-2.0, -1.0) & (0.0, 3.0) \\
\cdot & \cdot & (-5.0, 3.0) & (-3.0, -3.0) & (-5.0, -5.0) \\
\cdot & \cdot & \cdot & (4.0, -4.0) & (2.0, 0.0) \\
\cdot & \cdot & \cdot & \cdot & (2.0, -1.0)
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
(3.0, 4.0) \\
(-4.0, 2.0) \\
(-5.0, 0.0) \\
(1.0, 3.0) \\
(3.0, 1.0)
\end{bmatrix}
\]

Output:

\[
B = \begin{bmatrix}
(-8.0, -19.0) \\
(8.0, 21.0) \\
(44.0, -8.0) \\
(13.0, -7.0) \\
(19.0, 2.0)
\end{bmatrix}
\]
STRSM, DTRSM, CTRSM, and ZTRSM (Solution of Triangular Systems of Equations with Multiple Right-Hand Sides)

Purpose

STRSM and DTRSM perform one of the following solves for a triangular system of equations with multiple right-hand sides, using scalar $\alpha$, rectangular matrix $B$, and triangular matrix $A$ or its transpose:

Solution

1. $B \leftarrow \alpha (A^{-1})B$
2. $B \leftarrow \alpha (A^T)B$
3. $B \leftarrow \alpha B(A^T)$
4. $B \leftarrow \alpha B(A^{-1})$

CTRSM and ZTRSM perform one of the following solves for a triangular system of equations with multiple right-hand sides, using scalar $\alpha$, rectangular matrix $B$, and triangular matrix $A$, its transpose, or its conjugate transpose:

Solution

1. $B \leftarrow \alpha (A^{-1})B$
2. $B \leftarrow \alpha (A^T)B$
3. $B \leftarrow \alpha B(A^T)$
4. $B \leftarrow \alpha B(A^{-1})$
5. $B \leftarrow \alpha (A^{H})B$
6. $B \leftarrow \alpha B(A^{H})$

Note: The term $X$ 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 119. Data Types

<table>
<thead>
<tr>
<th>$A, B, \alpha$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>STRSM</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DTRSM</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CTRSM</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZTRSM</td>
</tr>
</tbody>
</table>

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL STRSM</th>
<th>DTRSM</th>
<th>CTRSM</th>
<th>ZTRSM (side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>strsm</td>
<td>dtrsm</td>
<td>ctrsm</td>
<td>ztrsm (side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb);</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_strsm</td>
<td>cblas_dtrsm</td>
<td>cblas_ctrsm</td>
<td>cblas_ztrsm (cblas_layout, cblas_side, cblas_uplo, cblas_transa, cblas_diag, m, n, alpha, a, lda, b, ldb);</td>
</tr>
</tbody>
</table>

On Entry
`cblas_layout` indicates whether the input and output matrices are stored in row major order or column major order, where:

- If `cblas_layout` = CblasRowMajor, the matrices are stored in row major order.
- If `cblas_layout` = CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

`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'.

`cblas_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 `cblas_side` = CblasLeft, `A` is to the left of `B`, resulting in solution 1, 2, or 5.
If `cblas_side` = CblasRight, `A` is to the right of `B`, resulting in solution 3, 4, or 6.

Specified as: an object of enumerated type CBLAS_SIDE. It must be CblasLeft or CblasRight.

`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'.

`cblas_uplo` indicates whether matrix `A` is an upper or lower triangular matrix, where:

If `cblas_uplo` = CblasUpper, `A` is an upper triangular matrix.
If `cblas_uplo` = CblasLower, `A` is a lower triangular matrix.

Specified as: an object of enumerated type CBLAS_UPLO. It must be CblasUpper or CblasLower.

`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', `A` T is used, resulting in solution 2 or 4.
If `transa` = 'C', `A` H is used, resulting in solution 5 or 6.

Specified as: a single character. It must be 'N', 'T', or 'C'.

`cblas_transa` indicates the form of matrix `A` to use in the computation, where:

If `cblas_transa` = CblasNoTrans, `A` is used, resulting in solution 1 or 3.
If `cblas_transa` = CblasTrans, `A` T is used, resulting in solution 2 or 4.
If \( \text{cblas\_transa} = \text{CblasConjTrans} \), \( \text{A}^\text{H} \) is used, resulting in solution 5 or 6.

Specified as: an object of enumerated type CBLAS_TRANSPOSE. It must be CblasNoTrans, CblasTrans, or CblasConjTrans.

diag
indicates the characteristics of the diagonal of matrix \( \text{A} \), where:

- If diag = 'U', \( \text{A} \) is a unit triangular matrix.
- If diag = 'N', \( \text{A} \) is not a unit triangular matrix.

Specified as: a single character. It must be 'U' or 'N'.

cblas\_diag
indicates the characteristics of the diagonal of matrix \( \text{A} \), where:

- If diag = CblasUnit, \( \text{A} \) is a unit triangular matrix.
- If diag = CblasNonUnit \( \text{A} \) is not a unit triangular matrix.

Specified as: an object of enumerated type CBLAS_DIAG. It must be CblasNonUnit or CblasUnit.

\( m \) is the number of rows in rectangular matrix \( \text{B} \), and:

- If side = 'L', \( m \) is the order of triangular matrix \( \text{A} \).

Specified as: an integer, where:

- If side = 'L', \( 0 \leq m \leq \text{lda} \) and \( m \leq \text{ldb} \).
- If side = 'R', \( 0 \leq m \leq \text{ldb} \).

\( n \) is the number of columns in rectangular matrix \( \text{B} \), and:

- If side = 'R', \( n \) is the order of triangular matrix \( \text{A} \).

Specified as: an integer; \( n \geq 0 \), and:

- If side = 'R', \( n \leq \text{lda} \).

alpha
is the scalar \( \alpha \). Specified as: a number of the data type indicated in Table 119 on page 486.

\( a \) is the triangular matrix \( \text{A} \), of which only the upper or lower triangular portion is used, where:

- If side = 'L', \( \text{A} \) is order \( m \).
- If side = 'R', \( \text{A} \) is order \( n \).

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 119 on page 486, where:

- If side = 'L', its size must be \( \text{lda} \) by (at least) \( m \).
- If side = 'R', its size must be \( \text{lda} \) by (at least) \( n \).

\( \text{lda} \)
is the leading dimension of the array specified for \( a \).

Specified as: an integer; \( \text{lda} > 0 \), and:

- If side = 'L', \( \text{lda} \geq m \).
- If side = 'R', \( \text{lda} \geq n \).

\( b \) is the \( m \) by \( n \) rectangular matrix \( \text{B} \), 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 119 on page 486](#).

**ldb**

is the leading dimension of the array specified for *b*.

Specified as: an 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 119 on page 486](#).

**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 93.

**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. \( B + \alpha (A^{-1})B \)
2. \( B + \alpha (A^{T})B \)
3. \( B + \alpha B (A^{-1}) \)
4. \( B + \alpha B (A^{T}) \)
5. \( B + \alpha (A^{H})B \) (only for CTRSM and ZTRSM)
6. \( B + \alpha B (A^{H}) \) (only for CTRSM and ZTRSM)

where:

\( \alpha \) 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 [40 on page 1365](#) and [44 on page 1366](#).
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. $cblas_layout \neq$ CblasRowMajor or CblasColMajor
2. $m < 0$
3. $m > ldb$
4. $n < 0$
5. $lda, ldb \leq 0$
6. $side \neq 'L$ or '$R$
7. $side = 'L$ and $m > lda$
8. $side = 'R$ and $n > lda$
9. $cblas_side \neq$ CblasLeft or CblasRight
10. $cblas_side = CblasLeft$ and $m > lda$
11. $cblas_side = CblasRight$ and $n > lda$
12. $uplo \neq 'L$ or '$U$
13. $cblas_uplo \neq$ CblasLower or CblasUpper
14. $transa \neq 'T', 'N$, or 'C'
15. $cblas_transa \neq$ CblasNoTrans, CblasTrans, or CblasConjTrans
16. $diag \neq 'N$ or '$U$
17. $cblas_diag \neq$ CblasNonUnit or CblasUnit

Examples

Example 1
This example shows the solution $B+\alpha(A^{-1})B$, 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 =  3.0  -1.0  2.0  2.0  1.0
   -2.0  4.0 -1.0  3.0
   .   -3.0  0.0  2.0
B =  6.0  10.0 -2.0
   -16.0 -1.0  6.0
   -2.0  1.0 -4.0
   14.0  0.0 -14.0
   -1.0  2.0  1.0
```

Output:

```
490 ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
```
Example 2

This example shows the solution $B^r \alpha (A^T) B$, where $A$ is a real 5 by 5 upper triangular matrix that is not unit triangular, and $B$ is a real 5 by 4 rectangular matrix.

Call Statement and Input:

```
SIDE UPLO TRANS A DIAG M N ALPHA A LDA B LDB
```

Call STRSM('L', 'U', 'T', 'N', 5, 4, 1.0, A, 7, B, 6)

```
\[
B = \begin{bmatrix}
2.0 & 3.0 & 1.0 \\
5.0 & 5.0 & 4.0 \\
0.0 & 1.0 & 2.0 \\
3.0 & 1.0 & -3.0 \\
-1.0 & 2.0 & 1.0
\end{bmatrix}
\]
```

```
A = \begin{bmatrix}
-1.0 & -4.0 & -2.0 & 2.0 & 3.0 \\
. & -2.0 & 2.0 & 2.0 & 2.0 \\
. & . & -3.0 & -1.0 & 4.0 \\
. & . & . & 1.0 & 0.0 \\
. & . & . & . & -2.0 \\
. & . & . & . & .
\end{bmatrix}
```

```
B = \begin{bmatrix}
-1.0 & -2.0 & -3.0 & -4.0 \\
2.0 & -2.0 & -14.0 & -12.0 \\
10.0 & 5.0 & -8.0 & -7.0 \\
14.0 & 15.0 & 1.0 & 8.0 \\
-3.0 & 4.0 & 3.0 & 16.0 \\
. & . & . & .
\end{bmatrix}
```

Output:

```
B = \begin{bmatrix}
1.0 & 2.0 & 3.0 & 4.0 \\
3.0 & 3.0 & -1.0 & 2.0 \\
-2.0 & -1.0 & 0.0 & 1.0 \\
4.0 & 4.0 & -3.0 & -3.0 \\
2.0 & 2.0 & 2.0 & 2.0 \\
. & . & . & .
\end{bmatrix}
```

Example 3

This example shows the solution $B^r \alpha B(A^{-1})$, 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 TRANS A DIAG M N ALPHA A LDA B LDB
```

Call STRSM('R', 'L', 'N', 'N', 3, 5, 1.0, A, 7, B, 4)

```
\[
A = \begin{bmatrix}
2.0 & . & . & . & . \\
2.0 & 3.0 & . & . & . \\
2.0 & 1.0 & 1.0 & . & . \\
. & 0.0 & 3.0 & 0.0 & -2.0 \\
. & . & . & . & .
\end{bmatrix}
\]
```
Example 4

This example shows the solution $B = \alpha B(A^{-1})$, 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:

```fortran
call strsm('R', 'U', 'N', 'U', 1, 6, 1.0, a, 7, b, 2)
```

Output:

```
1.0 4.0 -2.0 10.0 2.0 -6.0
```

Example 5

This example shows the solution $B = \alpha B(A^{-1})$, 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:

```fortran
call ctrsm('R', 'L', 'N', 'N', 3, 5, alpha, a, 7, b, 4)
```

```
alpha = (1.0, 0.0)
```
\[
A = \begin{bmatrix}
(2.0, -3.0) & . & . & . \\
(2.0, -4.0) & (3.0, -1.0) & . & . \\
(2.0, 2.0) & (1.0, 2.0) & (1.0, 1.0) & . \\
(0.0, 0.0) & (3.0, -1.0) & (0.0, -1.0) & (-2.0, 1.0) \\
(2.0, 2.0) & (4.0, 0.0) & (-1.0, 2.0) & (2.0, -4.0) & (-1.0, -4.0) \\
. & . & . & . & . \\
. & . & . & . & . \\
. & . & . & . & . \\
. & . & . & . & . \\
. & . & . & . & . \\
. & . & . & . & . \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
(22.0, -41.0) & (7.0, -26.0) & (9.0, 0.0) & (-15.0, -3.0) & (-15.0, 8.0) \\
(29.0, -18.0) & (24.0, -10.0) & (9.0, 6.0) & (-12.0, -24.0) & (-19.0, -8.0) \\
(-15.0, 2.0) & (-3.0, -21.0) & (-2.0, 4.0) & (-4.0, -12.0) & (-10.0, -6.0) \\
. & . & . & . & . \\
. & . & . & . & . \\
. & . & . & . & . \\
. & . & . & . & . \\
. & . & . & . & . \\
. & . & . & . & . \\
. & . & . & . & . \\
. & . & . & . & . \\
. & . & . & . & . \\
\end{bmatrix}
\]

**Example 6**

This example shows the solution \(B = \alpha (A^{-1})B\), 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 )
```

\[\text{ALPHA} = (1.0, 0.0)\]

\[
A = \begin{bmatrix}
(-4.0, 1.0) & (4.0, -3.0) & (-1.0, 3.0) & (0.0, 0.0) & (-1.0, 0.0) \\
. & (-2.0, 0.0) & (-3.0, -1.0) & (-2.0, -1.0) & (4.0, 3.0) \\
. & . & (-5.0, 3.0) & (-3.0, 3.0) & (-5.0, -5.0) \\
. & . & . & (4.0, -4.0) & (2.0, 0.0) \\
. & . & . & . & \end{bmatrix}
\]

\[
B = \begin{bmatrix}
(-8.0, -19.0) \\
(8.0, 21.0) \\
(44.0, -8.0) \\
(13.0, -7.0) \\
(19.0, 2.0) \\
\end{bmatrix}
\]

Output:

\[
B = \begin{bmatrix}
(3.0, 4.0) \\
(-4.0, 2.0) \\
(-5.0, 0.0) \\
(1.0, 3.0) \\
(3.0, 1.0) \\
\end{bmatrix}
\]
SSYRK, DSYRK, CSYRK, ZSYRK, CHERK, and ZHERK (Rank-K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix)

Purpose

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 $\alpha$ and $\beta$, real or complex matrix $A$ or its transpose, and real or complex symmetric matrix $C$ to compute:

1. $C \leftarrow \alpha AA^T + \beta C$
2. $C \leftarrow \alpha A^T A + \beta C$

CHERK and ZHERK use the scalars $\alpha$ and $\beta$, complex matrix $A$ or its complex conjugate transpose, and complex Hermitian matrix $C$ to compute:

1. $C \leftarrow \alpha AA^H + \beta C$
2. $C \leftarrow \alpha A^H A + \beta C$

Table 120. Data Types

<table>
<thead>
<tr>
<th>$A$, $C$</th>
<th>$\alpha$, $\beta$</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SSYRK</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DSYRK</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision complex</td>
<td>CSYRK</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision complex</td>
<td>ZSYRK</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>CHERK</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>ZHERK</td>
</tr>
</tbody>
</table>

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

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);

CBLAS

cblas_ssyrk | cblas_dsyrk | cblas_csyrk | cblas_zsyrk | cblas_cherk | cblas_zherk (cblas_layout, cblas_uplo, cblas_trans, n, k, alpha, a, lda, beta, c, ldc);

On Entry

$cblas_layout$

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If $cblas_layout$ = CblasRowMajor, the matrices are stored in row major order.
- If $cblas_layout$ = CblasColMajor, the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

$uplo$

indicates the storage mode used for matrix $C$, where:
If $\text{uplo} = 'U'$, $C$ is stored in upper storage mode.

If $\text{uplo} = 'L'$, $C$ is stored in lower storage mode.

Specified as: a single character. It must be 'U' or 'L'.

$cblas\_uplo$

indicates the storage mode used for matrix $A$, where:

If $cblas\_uplo = \text{CblasUpper}$, $A$ is stored in upper storage mode.

If $cblas\_uplo = \text{CblasLower}$, $A$ is stored in lower storage mode.

Specified as: an object of enumerated type CBLAS_UPLO. It must be CblasUpper or CblasLower.

$\text{trans}$

indicates the form of matrix $A$ to use in the computation, where:

If $\text{trans} = 'N'$, $A$ is used, resulting in equation 1 or 3.

If $\text{trans} = 'T'$, $A^T$ is used, resulting in equation 2.

If $\text{trans} = 'C'$, $A^H$ 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'.

$cblas\_trans$

indicates the form of matrix $A$ to use in the computation, where:

If $cblas\_trans = \text{CblasNoTrans}$, $A$ is used, resulting in equation 1 or 3.

If $cblas\_trans = \text{CblasTrans}$, $A^T$ is used, resulting in equation 2.

If $cblas\_trans = \text{CblasConjTrans}$, $A^H$ is used, resulting in equation 4.

Specified as: an object of enumerated type CBLAS_TRANSPOSE, where:

For SSYRK and DSYRK, it must be CblasNoTrans, CblasTrans, or CblasConjTrans.

For CSYRK and ZSYRK, it must be CblasNoTrans or CblasTrans.

For CHERK and ZHERK, it must be CblasNoTrans or CblasConjTrans.

$n$

is the order of matrix $C$.

Specified as: an integer; $0 \leq n \leq \text{ldc}$ and:

If $\text{trans} = 'N'$, then $n \leq \text{lda}$.

$k$

has the following meaning, where:

If $\text{trans} = 'N'$, it is the number of columns in matrix $A$.

If $\text{trans} = 'T'$ or 'C', it is the number of rows in matrix $A$.

Specified as: an integer; $k \geq 0$ and:

If $\text{trans} = 'T'$ or 'C', then $k \leq \text{lda}$.

$\alpha$

is the scalar $\alpha$.

Specified as: a number of the data type indicated in Table 120 on page 494.
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 120 on page 494 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: an integer; lda > 0 and:
If trans = 'N', lda \( \geq \) n.
If trans = 'T' or 'C', lda \( \geq \) k.

beta is the scalar \( \beta \).
Specified as: a number of the data type indicated in Table 120 on page 494.

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 120 on page 494.

ldc is the leading dimension of the array specified for c.
Specified as: an integer; ldc > 0 and ldc \( \geq \) 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 120 on page 494.

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 \( C \) are assumed to be zero, so you do not have to set these values. On output, they are set to zero, except when \( \beta \) is one and \( \alpha \) 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 85. For a description of how complex Hermitian matrices are stored in upper and lower storage mode, see "Complex Hermitian Matrix" on page 90.

**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. \( C \leftarrow \alpha A A^T + \beta C \)
2. \( C \leftarrow \alpha A^T A + \beta C \)

For CHERK and ZHERK, matrix \( C \) is complex Hermitian. They perform:

1. \( C \leftarrow \alpha A A^H + \beta C \)
2. \( C \leftarrow \alpha A^H A + \beta C \)

where:

\( \alpha \) and \( \beta \) 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 [40 on page 1365] and [46 on page 1366]. In the following two cases, no computation is performed:

- \( n \) is 0.
- \( \beta \) is one, and \( \alpha \) is zero or \( k \) is zero.

Assuming the above conditions do not exist, if \( \beta \) is not one, and \( \alpha \) is zero or \( k \) is zero, then \( \beta C \) is returned.

**Error conditions**

**Resource Errors**

Unable to allocate internal work area.

**Computational Errors**

None

**Input-Argument Errors**

1. \( \text{cblas_layout} \neq \text{CblasRowMajor} \) or \( \text{CblasColMajor} \)
2. \( \text{lda}, \text{ldc} \leq 0 \)
3. \( \text{ldc} < n \)
4. \( k, n < 0 \)
5. \( \text{uplo} \neq \text{'U'} \) or \( \text{'L'} \)
6. \( \text{cblas_uplo} \neq \text{CblasLower} \) or \( \text{CblasUpper} \)
7. \( \text{trans} \neq \text{'N', 'T', or 'C'} \) for SSYRK and DSYRK
8. \( \text{trans} \neq \text{'N'} \) or \( \text{'T'} \) for CSYRK and ZSYRK
9. \( \text{trans} \neq \text{'N'} \) or \( \text{'C'} \) for CHERK and ZHERK
10. \( \text{trans} = \text{'N'} \) and \( \text{lda} < n \)
11. \( \text{trans} = \text{'T'} \) or \( \text{'C'} \) and \( \text{lda} < k \)
12. \( \text{cblas_trans} \neq \text{CblasNoTrans, CblasTrans, or CblasConjTrans} \) for SSYRK and DSYRK
13. \( \text{cblas_trans} \neq \text{CblasNoTrans} \) or \( \text{CblasTrans} \) for CSYRK and ZSYRK
14. \( \text{cblas_trans} \neq \text{CblasNoTrans} \) or \( \text{CblasConjTrans} \) for CHERK and ZHERK
15. \( \text{cblas_trans} = \text{CblasNoTrans} \) and \( \text{lda} < n \)
16. \( \text{cblas_trans} = \text{CblasTrans, or CblasConjTrans} \) and \( \text{lda} < k \)

Examples

Example 1

This example shows the computation \( C \leftarrow \alpha AA^T + \beta C \), 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:

\[
\begin{array}{cccccccc}
\text{UPLO} & \text{TRANS} & N & K & \text{ALPHA} & A & \text{LDA} & \text{BETA} & C & \text{LDC} \\
\text{CXXSYRK}( & 'U', & 'N', & 8, & 2, & 1.0 & A & 9, & 1.0 & C & 10 )
\end{array}
\]

\[
\begin{bmatrix}
0.0 & 8.0  \\
1.0 & 9.0  \\
2.0 & 10.0 \\
3.0 & 11.0 \\
4.0 & 12.0  \\
5.0 & 13.0  \\
6.0 & 14.0  \\
7.0 & 15.0  \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.0 & 1.0 & 3.0 & 6.0 & 10.0 & 15.0 & 21.0 & 28.0 \\
. & 2.0 & 4.0 & 7.0 & 11.0 & 16.0 & 22.0 & 29.0 \\
. & . & 5.0 & 8.0 & 12.0 & 17.0 & 23.0 & 30.0 \\
. & . & . & 9.0 & 13.0 & 18.0 & 24.0 & 31.0 \\
. & . & . & . & 14.0 & 19.0 & 25.0 & 32.0 \\
. & . & . & . & . & 20.0 & 26.0 & 33.0 \\
. & . & . & . & . & . & 27.0 & 34.0 \\
. & . & . & . & . & . & . & 35.0 \\
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
64.0 & 73.0 & 83.0 & 94.0 & 106.0 & 119.0 & 133.0 & 148.0 \\
84.0 & 96.0 & 109.0 & 123.0 & 138.0 & 154.0 & 171.0 & 189.0 \\
. & 109.0 & 124.0 & 140.0 & 157.0 & 175.0 & 194.0 & 214.0 \\
. & . & 139.0 & 157.0 & 176.0 & 196.0 & 217.0 & 240.0 \\
. & . & . & 174.0 & 195.0 & 217.0 & 240.0 & 263.0 \\
. & . & . & . & 214.0 & 238.0 & 263.0 & 290.0 \\
. & . & . & . & . & 259.0 & 286.0 & 309.0 \\
. & . & . & . & . & . & 309.0 & 337.0 \\
\end{bmatrix}
\]
Example 2

This example shows the computation $C \leftarrow \alpha A^T A + \beta C$, where $A$ is a 3 by 8 real rectangular matrix, and $C$ is a real symmetric matrix of order 8, stored in lower storage mode.

Call Statement and Input:

```
CALL SSYRK( 'L', 'T', 8, 3, 1.0, A, 4, 1.0, C, 8 )
```

$A = \begin{bmatrix}
0.0 & 3.0 & 6.0 & 9.0 & 12.0 & 15.0 & 18.0 & 21.0 \\
1.0 & 4.0 & 7.0 & 10.0 & 13.0 & 16.0 & 19.0 & 22.0 \\
2.0 & 5.0 & 8.0 & 11.0 & 14.0 & 17.0 & 20.0 & 23.0 \\
\end{bmatrix}$

$C = \begin{bmatrix}
0.0 & . & . & . & . & . & . & . \\
1.0 & 8.0 & . & . & . & . & . & . \\
2.0 & 9.0 & 15.0 & . & . & . & . & . \\
3.0 & 10.0 & 16.0 & 21.0 & . & . & . & . \\
4.0 & 11.0 & 17.0 & 22.0 & 26.0 & . & . & . \\
5.0 & 12.0 & 18.0 & 23.0 & 27.0 & 30.0 & . & . \\
6.0 & 13.0 & 19.0 & 24.0 & 28.0 & 31.0 & 33.0 & . \\
7.0 & 14.0 & 20.0 & 25.0 & 29.0 & 32.0 & 34.0 & 35.0
\end{bmatrix}$

Output:

$C = \begin{bmatrix}
5.0 & . & . & . & . & . & . & . \\
15.0 & 58.0 & . & . & . & . & . & . \\
25.0 & 95.0 & 164.0 & . & . & . & . & . \\
35.0 & 132.0 & 228.0 & 323.0 & . & . & . & . \\
45.0 & 169.0 & 292.0 & 414.0 & 535.0 & . & . & . \\
55.0 & 206.0 & 356.0 & 505.0 & 653.0 & 800.0 & . & . \\
65.0 & 243.0 & 420.0 & 596.0 & 771.0 & 945.0 & 1118.0 & . \\
75.0 & 280.0 & 484.0 & 687.0 & 889.0 & 1090.0 & 1290.0 & 1489.0
\end{bmatrix}$

Example 3

This example shows the computation $C \leftarrow \alpha A A^T + \beta C$, 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:

```
CALL CSYRK( 'U', 'N', 3, 5, ALPHA, A, 3, BETA, C, 4 )
```

$\text{ALPHA} = (1.0, 1.0)$
$\text{BETA} = (1.0, 1.0)$

$A = \begin{bmatrix}
(2.0, 0.0) & (3.0, 2.0) & (4.0, 1.0) & (1.0, 7.0) & (0.0, 0.0) \\
(3.0, 3.0) & (8.0, 0.0) & (2.0, 5.0) & (2.0, 4.0) & (1.0, 2.0) \\
(1.0, 3.0) & (2.0, 1.0) & (6.0, 0.0) & (3.0, 2.0) & (2.0, 2.0)
\end{bmatrix}$

$C = \begin{bmatrix}
(2.0, 1.0) & (1.0, 9.0) & (4.0, 5.0) \\
. & (3.0, 1.0) & (6.0, 7.0) \\
. & . & (8.0, 1.0)
\end{bmatrix}$

Output:
Example 4

This example shows the computation $C \leftarrow \alpha A^\dagger A + \beta C$, 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:

```
CALL CHERK( 'L' , 'C' , 3 , 5 , 1.0 , A , 5 , 1.0 , C , 4 )
```

$A = \begin{bmatrix}
(2.0, 0.0) & (3.0, 2.0) & (4.0, 1.0) \\
(3.0, 3.0) & (8.0, 0.0) & (2.0, 5.0) \\
(1.0, 3.0) & (2.0, 1.0) & (6.0, 0.0) \\
(3.0, 3.0) & (8.0, 0.0) & (2.0, 5.0) \\
(1.0, 9.0) & (3.0, 0.0) & (6.0, 7.0)
\end{bmatrix}$

$C = \begin{bmatrix}
(6.0, \cdot) & \cdot & \cdot \\
(3.0, 4.0) & (10.0, \cdot) & \cdot \\
(9.0, 1.0) & (12.0, 2.0) & (3.0, \cdot)
\end{bmatrix}$

Output:

$C = \begin{bmatrix}
(138.0, 0.0) & \cdot & \cdot \\
(65.0, 88.0) & (165.0, 0.0) & \cdot \\
(134.0, 46.0) & (88.0, -88.0) & (199.0, 0.0)
\end{bmatrix}$
SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, and ZHER2K (Rank-2K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix)

Purpose

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 \( \alpha \) and \( \beta \), real or complex matrices \( A \) and \( B \) or their transposes, and real or complex symmetric matrix \( C \) to compute:

1. \( C \leftarrow \alpha AB^T + \alpha BA^T + \beta C \)
2. \( C \leftarrow \alpha A^TB + \alpha B^TA + \beta C \)

CHER2K and ZHER2K use the scalars \( \alpha \) and \( \beta \), complex matrices \( A \) and \( B \) or their complex conjugate transposes, and complex Hermitian matrix \( C \) to compute:

3. \( C \leftarrow \alpha AB^H + \overline{\alpha BA^H} + \beta C \)
4. \( C \leftarrow \alpha A^HB + \overline{\alpha B^HA} + \beta C \)

Table 121. Data Types

<table>
<thead>
<tr>
<th>A, B, C, ( \alpha )</th>
<th>( \beta )</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SSYR2K</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DSYR2K</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision complex</td>
<td>CSYR2K</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision complex</td>
<td>ZSYR2K</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>CHER2K</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>ZHER2K</td>
</tr>
</tbody>
</table>

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

Fortran

\[
\text{CALL SSYR2K | DSYR2K | CSYR2K | ZSYR2K | CHER2K | ZHER2K (uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)}
\]

C and C++

\[
\text{ssyr2k | dsyr2k | csyr2k | zsy2k | cher2k | zher2k (uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)}
\]

CBLAS

\[
\text{cblas_ssy2k | cblas_dsy2k | cblas_csy2k | cblas_zsy2k | cblas_cher2k | cblas_zher2k (cblas_layout, cblas_uplo, cblas_trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc));}
\]

On Entry

\textit{cblas\_layout}

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If \textit{cblas\_layout} = CblasRowMajor, the matrices are stored in row major order.

Chapter 9. Matrix Operations 501
If \( cblas\_layout = \text{CblasColMajor} \), the matrices are stored in column major order.

Specified as: an object of enumerated type CBLAS_LAYOUT. It must be CblasRowMajor or CblasColMajor.

\( uplo \)
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'.

\( cblas\_uplo \)
indicates the storage mode used for matrix \( A \), where:

- If \( cblas\_uplo = \text{CblasUpper} \), \( A \) is stored in upper storage mode.
- If \( cblas\_uplo = \text{CblasLower} \), \( A \) is stored in lower storage mode.

Specified as: an object of enumerated type CBLAS_UPLO. It must be CblasUpper or CblasLower.

\( trans \)
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 \( trans = 'T' \), \( A^T \) and \( B^T \) are used, resulting in equation 2.
- If \( trans = 'C' \), \( A^H \) and \( B^H \) 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'.

\( cblas\_trans \)
indicates the form of matrix \( A \) to use in the computation, where:

- If \( cblas\_trans = \text{CblasNoTrans} \), \( A \) is used, resulting in equation 1 or 3.
- If \( cblas\_trans = \text{CblasTrans} \), \( A^T \) is used, resulting in equation 2.
- If \( cblas\_trans = \text{CblasConjTrans} \), \( A^H \) is used, resulting in equation 4.

Specified as: an object of enumerated type CBLAS_TRANSPOSE, where:

- For SSYR2K and DSYR2K, it must be CblasNoTrans, CblasTrans, or CblasConjTrans.
- For CSYR2K and ZSYR2K, it must be CblasNoTrans or CblasTrans.
- For CHER2K and ZHER2K, it must be CblasNoTrans or CblasConjTrans.

\( n \)
is the order of matrix \( C \).

Specified as: an integer; \( 0 \leq n \leq \text{ldc} \) and:

- If \( trans = 'N' \), then \( n \leq \text{lda} \) and \( n \leq \text{ldb} \).

\( k \)
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: an integer; \( k \geq 0 \) and:

If \( \text{trans} = 'T' \) or \( 'C' \), then \( k \leq \text{lda} \) and \( k \leq \text{ldb} \).

\( \alpha \)

is the scalar \( \alpha \).

Specified as: a number of the data type indicated in Table 121 on page 501

\( a \)

is the rectangular matrix \( A \), where:

If \( \text{trans} = 'N' \), \( A \) is \( n \) by \( k \).

If \( \text{trans} = 'T' \) 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 121 on page 501 where:

If \( \text{trans} = 'N' \), its size must be \( \text{lda} \) by (at least) \( k \).

If \( \text{trans} = 'T' \) or \( 'C' \), its size must be \( \text{lda} \) by (at least) \( n \).

\( \text{lda} \)

is the leading dimension of the array specified for \( a \).

Specified as: an integer; \( \text{lda} > 0 \) and:

If \( \text{trans} = 'N' \), \( \text{lda} \geq n \).

If \( \text{trans} = 'T' \) or \( 'C' \), \( \text{lda} \geq k \).

\( b \)

is the rectangular matrix \( B \), where:

If \( \text{trans} = 'N' \), \( B \) is \( n \) by \( k \).

If \( \text{trans} = 'T' \) 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 121 on page 501 where:

If \( \text{trans} = 'N' \), its size must be \( \text{ldb} \) by (at least) \( k \).

If \( \text{trans} = 'T' \) or \( 'C' \), its size must be \( \text{ldb} \) by (at least) \( n \).

\( \text{ldb} \)

is the leading dimension of the array specified for \( b \).

Specified as: an integer; \( \text{ldb} > 0 \) and:

If \( \text{trans} = 'N' \), \( \text{ldb} \geq n \).

If \( \text{trans} = 'T' \) or \( 'C' \), \( \text{ldb} \geq k \).

\( \beta \)

is the scalar \( \beta \).

Specified as: a number of the data type indicated in Table 121 on page 501

\( c \)

is matrix \( C \) of order \( n \), which is real symmetric, complex symmetric, or complex Hermitian, where:

If \( \text{uplo} = 'U' \), \( C \) is stored in upper storage mode.

If \( \text{uplo} = 'L' \), \( C \) is stored in lower storage mode.
Specified as: an \( ldc \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 121 on page 501.

\( ldc \)

is the leading dimension of the array specified for \( c \).

Specified as: an integer; \( ldc > 0 \) and \( ldc \geq n \).

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 121 on page 501.

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 \( C \) are assumed to be zero, so you do not have to set these values. On output, they are set to zero, except when \( \beta \) is one and \( \alpha \) 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 85. For a description of how complex Hermitian matrices are stored in upper and lower storage mode, see "Complex Hermitian Matrix" on page 90.

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. \( C \leftarrow \alpha AB^T + \alpha BA^T + \beta C \)
2. \( C \leftarrow \alpha A^T B + \alpha B^T A + \beta C \)

For CHER2K and ZHER2K, matrix \( C \) is complex Hermitian. They perform:

3. \( C \leftarrow \alpha AB^H + \bar{\alpha} BA^H + \beta C \)
4. \( C \leftarrow \alpha A^H B + \bar{\alpha} B^H A + \beta C \)

where:

\( \alpha \) and \( \beta \) 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 [40 on page 1365], [46 on page 1366], and [84 on page 1368]. In the following two cases, no computation is performed:

- \( n \) is 0.
- \( \beta \) is one, and \( \alpha \) is zero or \( k \) is zero.

Assuming the above conditions do not exist, if \( \beta \) is not one, and \( \alpha \) is zero or \( k \) is zero, then \( \beta C \) is returned.

**Error conditions**

**Resource Errors**
- Unable to allocate internal work area.

**Computational Errors**
- None

**Input-Argument Errors**

1. \( \text{cblas_layout} \neq \text{CblasRowMajor or CblasColMajor} \)
2. \( \text{lda, ldb, ldc} \leq 0 \)
3. \( \text{ldc} < n \)
4. \( \text{k, n} < 0 \)
5. \( \text{uplo} \neq \text{'U' or 'L'} \)
6. \( \text{cblas_uplo} \neq \text{CblasLower or CblasUpper} \)
7. \( \text{trans} \neq \text{'N', 'T', or 'C'} \) for SSYR2K and DSYR2K
8. \( \text{trans} \neq \text{'N' or 'T'} \) for CSYR2K and ZSYR2K
9. \( \text{trans} \neq \text{'N' or 'C'} \) for CHER2K and ZHER2K
10. \( \text{trans} = \text{'N'} \) and \( \text{lda} < n \)
11. \( \text{trans} = \text{'T' or 'C'} \) and \( \text{lda} < k \)
12. \( \text{trans} = \text{'N'} \) and \( \text{ldb} < n \)
13. \( \text{trans} = \text{'T' or 'C'} \) and \( \text{ldb} < k \)
14. \( \text{cblas_transa} \neq \text{CblasNoTrans, CblasTrans, or CblasConjTrans for SSYR2K and DSYR2K} \)
15. \( \text{cblas_transa} \neq \text{CblasNoTrans or CblasTrans for CSYR2K and ZSYR2K} \)
16. \( \text{cblas_transa} \neq \text{CblasNoTrans or CblasConjTrans for CHER2K and ZHER2K} \)
17. \( \text{cblas_transa} = \text{CblasNoTrans and lda} < n \)
18. \( \text{cblas_transa} = \text{CblasTrans, or CblasConjTrans and lda} < k \)
19. \( \text{cblas_trans} = \text{CblasNoTrans and ldb} < n \)
20. \( \text{cblas_trans} = \text{CblasNoTrans or CblasConjTrans and ldb} < k \)

**Examples**

**Example 1**

This example shows the computation \( C \leftarrow \alpha AB^T + \alpha BA^T + \beta C \), where \( A \) and \( B \) are 8 by 2 real rectangular matrices, 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 B LDB BETA C LDC
CALL SSYR2K('U', 'N', 8, 2, 1.0, A, 9, B, 8, 1.0, C, 10)
```

Chapter 9. Matrix Operations  505
Example 2

This example shows the computation $C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$, where $A$ and $B$ are 3 by 8 real rectangular matrices, and $C$ 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 )
```

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
       .       .       .   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 $C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$, where $A$ and $B$ are 3 by 8 real rectangular matrices, and $C$ 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 )
```

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
       .       .       .   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 $C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$, where $A$ and $B$ are 3 by 8 real rectangular matrices, and $C$ 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 )
```

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
       .       .       .   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 $C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$, where $A$ and $B$ are 3 by 8 real rectangular matrices, and $C$ 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 )
```

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
       .       .       .   174.0 177.0 181.0 186.0
       .       .       .       .   172.0 172.0 173.0
       .       .       .       .       .   163.0 160.0
       .       .       .       .       .       .   147.0
       .       .       .       .       .       .       .
       .       .       .       .       .       .       .       .
```
Example 3

This example shows the computation $C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$, where $A$ and $B$ are 3 by 5 complex rectangular matrices, and $C$ is a complex symmetric matrix of order 3, stored in lower storage mode.

Call Statement and Input:

```
CALL CSYR2K('L', 'N', 3, 5, ALPHA, A, 3, B, 3, BETA, C, 4 )
```

```
ALPHA = (1.0, 1.0)
BETA  = (1.0, 1.0)
```

```
A = [
(2.0, 5.0) (3.0, 2.0) (4.0, 1.0) (1.0, 7.0) (0.0, 0.0) \\
(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)
]
```

```
B = [
(1.0, 5.0) (6.0, 2.0) (3.0, 1.0) (2.0, 0.0) (1.0, 0.0) \\
(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)
]
```

```
C = [
(2.0, 3.0) . . . . .
(1.0, 9.0) (3.0, 3.0) .
(4.0, 5.0) (6.0, 7.0) (8.0, 3.0)
]
```

Output:

```
C = [
(-101.0, 121.0) . . . .
(-182.0, 192.0) (-274.0, 248.0) .
]
```
Example 4

This example shows the computation:

\[ \mathbf{C} \leftarrow \alpha \mathbf{A}^H \mathbf{B} + \overline{\alpha} \mathbf{B}^H \mathbf{A} + \beta \mathbf{C} \]

where \( \mathbf{A} \) and \( \mathbf{B} \) are 5 by 3 complex rectangular matrices, and \( \mathbf{C} \) 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:

\[
\begin{array}{cccccccc}
\text{UPLO} & \text{TRANS} & \text{N} & \text{K} & \text{ALPHA} & \text{A} & \text{LDA} & \text{B} & \text{LDB} & \text{BETA} & \text{C} & \text{LDC} \\
\end{array}
\]

\[
\text{CALL} \ 	ext{CHER2K}( \ 'U' , \ 'C' , \ 3 , \ 5 , \ \text{ALPHA} , \ \text{A} , \ 5 , \ \text{B} , \ 5 , \ 1.0 , \ \text{C} , \ 4 )
\]

\[
\text{ALPHA} \ = \ (1.0, \ 1.0)
\]

\[
\text{A} \ = \ 
\begin{bmatrix}
(2.0, \ 0.0) & (3.0, \ 2.0) & (4.0, \ 1.0) \\
(3.0, \ 3.0) & (8.0, \ 0.0) & (2.0, \ 5.0) \\
(1.0, \ 3.0) & (2.0, \ 1.0) & (6.0, \ 0.0) \\
(3.0, \ 3.0) & (8.0, \ 0.0) & (2.0, \ 5.0) \\
(1.0, \ 9.0) & (3.0, \ 0.0) & (6.0, \ 7.0)
\end{bmatrix}
\]

\[
\text{B} \ = \ 
\begin{bmatrix}
(4.0, \ 5.0) & (6.0, \ 7.0) & (8.0, \ 0.0) \\
(1.0, \ 9.0) & (3.0, \ 0.0) & (6.0, \ 7.0) \\
(3.0, \ 3.0) & (8.0, \ 0.0) & (2.0, \ 5.0) \\
(1.0, \ 3.0) & (2.0, \ 1.0) & (6.0, \ 0.0) \\
(2.0, \ 0.0) & (3.0, \ 2.0) & (4.0, \ 1.0)
\end{bmatrix}
\]

\[
\text{C} \ = \ 
\begin{bmatrix}
(6.0, \ .) & (3.0, \ 4.0) & (9.0, \ 1.0) \\
. & (10.0, \ .) & (12.0, \ 2.0) \\
. & . & (3.0, \ .)
\end{bmatrix}
\]

Output:

\[
\text{C} \ = \ 
\begin{bmatrix}
(102.0, \ 0.0) & (56.0, \ -143.0) & (244.0, \ -96.0) \\
. & (174.0, \ 0.0) & (238.0, \ 78.0) \\
. & . & (363.0, \ 0.0)
\end{bmatrix}
\]
SGETMI, DGETMI, CGETMI, ZGETMI, CGECMI and ZGECMI (General Matrix Transpose or Conjugate Transpose [In-Place])

Purpose

Subroutines SGETMI, DGETMI, CGETMI, and ZGETMI perform a transpose of an $n \times n$ matrix $A$ in place—that is, in matrix $A$:

$$A \leftarrow A^T$$

Subroutines CGECMI and ZGECMI perform a conjugate transpose of an $n \times n$ matrix $A$ in place—that is, in matrix $A$:

$$A \leftarrow A^H$$

Table 122. Data Types

<table>
<thead>
<tr>
<th>$A$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGETMI</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGETMI</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGETMI</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGETMI</td>
</tr>
</tbody>
</table>

Note: On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SGETMI</th>
<th>DGETMI</th>
<th>CGETMI</th>
<th>ZGETMI</th>
<th>CGECMI</th>
<th>ZGECMI $(a, lda, n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>sgetmi</td>
<td>dgetmi</td>
<td>cgetmi</td>
<td>zgetmi</td>
<td>cgecmi</td>
<td>zgecmi $(a, lda, n)$</td>
</tr>
</tbody>
</table>

On Entry

$a$ is the matrix $A$ having $n$ rows and $n$ columns.

Specified as: an $lda$ by (at least) $n$ array, containing numbers of the data type indicated in Table 122.

lda

is the leading dimension of the array specified for $a$.

Specified as: an integer; $lda > 0$ and $lda \geq n$.

$n$

is the number of rows and columns in matrix $A$.

Specified as: an integer; $n \geq 0$.

On Return

$a$ is the $n \times n$ matrix, containing the results of the operation.

Returned as: an $lda$ by (at least) $n$ array, containing numbers of the data type indicated in Table 122.
Function

Subroutines SGETMI, DGETMI, CGETMI, and ZGETMI perform a transpose of matrix \( A \) in place. For matrix \( A \) with elements \( a_{ij} \) where \( i, j = 1, n \), the in-place transpose is expressed as:

\[
a_{ji} = a_{ij} \quad \text{for} \quad i, j = 1, n
\]

Subroutines CGECMI and ZGECMI perform a conjugate transpose of matrix \( A \) in place. For matrix \( A \) with elements \( a_{ij} \) where \( i, j = 1, n \), the in-place conjugate transpose is expressed as:

\[
\bar{a}_{ji} = a_{ij} \quad \text{for} \quad i, j = 1, n
\]

If \( n \) is 0, no computation is performed.

Error conditions

Computational Errors
   None

Input-Argument Errors
   1. \( n < 0 \) or \( n > lda \)
   2. \( lda \leq 0 \)

Examples

Example 1

This example shows an in-place matrix transpose of matrix \( A \) having 5 rows and 5 columns.

Call Statement and Input:

\[
\begin{align*}
A & \quad \text{LDA} & \quad N \\
\end{align*}
\]

CALL SGETMI( \( A(2,3) \), 10, 5 )

\[
A = \begin{bmatrix}
\cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & 1.0 & 6.0 & 11.0 & 16.0 & 21.0 \\
\cdot & 2.0 & 7.0 & 12.0 & 17.0 & 22.0 \\
\cdot & 3.0 & 8.0 & 13.0 & 18.0 & 23.0 \\
\cdot & 4.0 & 9.0 & 14.0 & 19.0 & 24.0 \\
\cdot & 5.0 & 10.0 & 15.0 & 20.0 & 25.0 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\end{bmatrix}
\]

Output:

\[
A = \begin{bmatrix}
\cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 \\
\cdot & 6.0 & 7.0 & 8.0 & 9.0 & 10.0 \\
\cdot & 11.0 & 12.0 & 13.0 & 14.0 & 15.0 \\
\cdot & 16.0 & 17.0 & 18.0 & 19.0 & 20.0 \\
\cdot & 21.0 & 22.0 & 23.0 & 24.0 & 25.0 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\end{bmatrix}
\]
Example 2

This example shows an in-place matrix conjugate transpose of matrix \( A \) having 5 rows and 5 columns.

Call Statement and Input:
\[
\text{A} \quad \text{LDA} \quad \text{N}
\]
\[
\text{CALL} \ ZGECMI( \text{A}(2,3), 10, 5 )
\]
\[
\begin{bmatrix}
\cdots & \cdots & \cdots & \cdots & \cdots \\
\cdots & (1.0,1.0) & (6.0, 6.0) & (11.0,11.0) & (16.0,16.0) & (21.0,21.0) \\
\cdots & (2.0,2.0) & (7.0, 7.0) & (12.0,12.0) & (17.0,17.0) & (22.0,22.0) \\
\cdots & (3.0,3.0) & (8.0, 8.0) & (13.0,13.0) & (18.0,18.0) & (23.0,23.0) \\
\end{bmatrix}
\]
\[
\text{A} =
\begin{bmatrix}
\cdots & \cdots & \cdots & \cdots & \cdots \\
\cdots & (4.0,4.0) & (9.0, 9.0) & (14.0,14.0) & (19.0,19.0) & (24.0,24.0) \\
\cdots & (5.0,5.0) & (10.0,10.0) & (15.0,15.0) & (20.0,20.0) & (25.0,25.0) \\
\end{bmatrix}
\]

Output:
\[
\begin{bmatrix}
\cdots & \cdots & \cdots & \cdots & \cdots \\
\cdots & (1.0, -1.0) & (2.0, -2.0) & (3.0, -3.0) & (4.0, -4.0) & (5.0, -5.0) \\
\cdots & (6.0, -6.0) & (7.0, -7.0) & (8.0, -8.0) & (9.0, -9.0) & (10.0, -10.0) \\
\cdots & (11.0, -11.0) & (12.0, -12.0) & (13.0, -13.0) & (14.0, -14.0) & (15.0, -15.0) \\
\end{bmatrix}
\]
\[
\text{A} =
\begin{bmatrix}
\cdots & \cdots & \cdots & \cdots & \cdots \\
\cdots & (16.0, -16.0) & (17.0, -17.0) & (18.0, -18.0) & (19.0, -19.0) & (20.0, -20.0) \\
\cdots & (21.0, -21.0) & (22.0, -22.0) & (23.0, -23.0) & (24.0, -24.0) & (25.0, -25.0) \\
\end{bmatrix}
\]
SGEYMO, DGETMO, CGETMO, ZGETMO, CGECMO, and ZGECMO  
(General Matrix Transpose or Conjugate Transpose [Out-of-Place])

**Purpose**

Subroutines SGETMO, DGETMO, CGETMO, and ZGETMO perform a transpose of an \( m \) by \( n \) matrix \( A \) out of place, returning the result in matrix \( B \):

\[ B \leftarrow A^T \]

Subroutines CGECMO, and ZGECMO perform a conjugate transpose of an \( m \) by \( n \) matrix \( A \) out of place, returning the result in matrix \( B \):

\[ B \leftarrow A^H \]

**Table 123. Data Types**

<table>
<thead>
<tr>
<th>( A, B )</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGETMO</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGETMO</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGETMO</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>CGECMO</td>
</tr>
<tr>
<td></td>
<td>ZGETMO</td>
</tr>
<tr>
<td></td>
<td>ZGECMO</td>
</tr>
</tbody>
</table>

**Note:** On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

**Syntax**

**Fortran**

```
CALL SGETMO | DGETMO | CGETMO | ZGETMO | CGECMO | ZGECMO (a, lda, m, n, b, ldb);
```

**C and C++**

```
sgetmo | dgetmo | cgetmo | zgetmo | cgecmo | zgecmo (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 123.

- \( lda \) is the leading dimension of the array specified for \( a \).
  
  Specified as: an integer; \( lda > 0 \) and \( lda \equiv m \).

- \( m \) is the number of rows in matrix \( A \) and the number of columns in matrix \( B \).
  
  Specified as: an integer; \( m \geq 0 \).

- \( n \) is the number of columns in matrix \( A \) and the number of rows in matrix \( B \).
  
  Specified as: an integer; \( n \geq 0 \).

- \( b \) See **On Return**

- \( ldb \) is the leading dimension of the array specified for \( b \).
Specified as: an 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 operation.
- Returned as: an \( ldb \) by (at least) \( m \) array, containing numbers of the data type indicated in Table 123 on page 512.

**Notes**

1. The matrix \( B \) must have no common elements with matrix \( A \); otherwise, results are unpredictable. See "Concepts" on page 75.

**Function**

Subroutines SGETMO, DGETMO, CGETMO, and ZGETMO perform a transpose of matrix \( A \) out of place. For matrix \( A \) with elements \( a_{ij} \) where \( i = 1, m \) and \( j = 1, n \), the out-of-place transpose is expressed as:

\[
b_{ji} = a_{ij} \text{ for } i = 1, m \text{ and } j = 1, n
\]

Subroutines CGECMO and ZGECMO perform a conjugate transpose of matrix \( A \) out of place. For matrix \( A \) with elements \( a_{ij} \) where \( i = 1, m \) and \( j = 1, n \), the out-of-place transpose is expressed as:

\[
b_{ji} = \bar{a}_{ij} \text{ for } i = 1, m \text{ and } j = 1, n
\]

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. \( ldb \leq 0 \)

**Examples**

**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:

```
CALL SGETMO( A(2,3) , 10 , 5 , 4 , B(2,2) , 6 )
```

\[
A = \begin{bmatrix}
. & . & 1.0 & 6.0 & 11.0 & 16.0 \\
. & . & 2.0 & 7.0 & 12.0 & 17.0 \\
. & . & 3.0 & 8.0 & 13.0 & 18.0 \\
. & . & 4.0 & 9.0 & 14.0 & 19.0
\end{bmatrix}
\]
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:

\[
A \quad \text{LDA} \quad M \quad N \quad B \quad \text{LDB}
\]

\[
\text{CALL SGETMO( } A(2,3) , 10 , 5 , 4 , A(7,1) , 10 \ )
\]

Output:

\[
\begin{bmatrix}
  \cdot & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 \\
  \cdot & 6.0 & 7.0 & 8.0 & 9.0 & 10.0 \\
  \cdot & 11.0 & 12.0 & 13.0 & 14.0 & 15.0 \\
  \cdot & 16.0 & 17.0 & 18.0 & 19.0 & 20.0 \\
  \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{bmatrix}
\]

Example 3

This example shows an out-of-place matrix conjugate transpose of matrix \( A \), having 5 rows and 4 columns, with the result going into matrix \( B \).

Call Statement and Input:

\[
A \quad \text{LDA} \quad M \quad N \quad B \quad \text{LDB}
\]

\[
\text{CALL ZGECMO( } A(2,3) , 10 , 5 , 4 , B(2,2) , 6 \ )
\]

Output:

\[
\begin{bmatrix}
  \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \\
  \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
  \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
  \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
  \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{bmatrix}
\]
<table>
<thead>
<tr>
<th></th>
<th>11.0, -11.0</th>
<th>12.0, -12.0</th>
<th>13.0, -13.0</th>
<th>14.0, -14.0</th>
<th>15.0, -15.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.0, -16.0</td>
<td>17.0, -17.0</td>
<td>18.0, -18.0</td>
<td>19.0, -19.0</td>
<td>20.0, -20.0</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 10. Linear Algebraic Equations

The linear algebraic equation subroutines, provided in four areas, are described here.

Overview of the Linear Algebraic Equation Subroutines

This describes the subroutines in each of the four linear algebraic equation areas:

- "Dense Linear Algebraic Equation Subroutines"
- "Banded Linear Algebraic Equation Subroutines" on page 520
- "Sparse Linear Algebraic Equation Subroutines" on page 522
- "Linear Least Squares Subroutines" on page 523

Note: Some of the linear algebraic equations were designed in accordance with the 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 LAPACK, see [8 on page 1363].

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, indefinite real or complex symmetric or complex Hermitian matrices, and triangular matrices. Some of these subroutines correspond to the LAPACK routines described in reference [8 on page 1363].

Table 124. List of LAPACK Dense Linear Algebraic Equation Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGESVA</td>
<td>DGESVA</td>
<td>&quot;SGESV, DGESV, CGESV, ZGESV (General Matrix Factorization and Multiple Right-Hand Side Solve)&quot; on page 530</td>
</tr>
<tr>
<td>LAPACKE_sggesvA</td>
<td>LAPACKE_dggesvA</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_zggesvA</td>
<td>LAPACKE_iggesvA</td>
<td></td>
</tr>
<tr>
<td>CGESVA</td>
<td>ZGESVA</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgesvA</td>
<td>LAPACKE_zggesvA</td>
<td></td>
</tr>
<tr>
<td>SGETRFA</td>
<td>DGETRFa</td>
<td>&quot;SGETRF, DGETRF, CGETRF and ZGETRF (General Matrix Factorization)&quot; on page 534</td>
</tr>
<tr>
<td>LAPACKE_sgetrfA</td>
<td>LAPACKE_dgetrfA</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgetrfA</td>
<td>LAPACKE_zgetrfA</td>
<td></td>
</tr>
<tr>
<td>SGETRSA</td>
<td>DGETRSA</td>
<td>&quot;SGETRS, DGETRS, CGETRS, and ZGETRS (General Matrix Multiple Right-Hand Side Solve)&quot; on page 539</td>
</tr>
<tr>
<td>LAPACKE_sgetrsa</td>
<td>LAPACKE_dgetrsa</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgetrsa</td>
<td>LAPACKE_zgetrsa</td>
<td></td>
</tr>
<tr>
<td>SGECONA</td>
<td>DGECONA</td>
<td>&quot;SGECON, DGECON, CGECON, and ZGECON (Estimate the Reciprocal of the Condition Number of a General Matrix)&quot; on page 556</td>
</tr>
<tr>
<td>LAPACKE_sgeconA</td>
<td>LAPACKE_dgeconA</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_igeconA</td>
<td>LAPACKE_igeconA</td>
<td></td>
</tr>
<tr>
<td>CGECONA</td>
<td>ZGECONA</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_igeconA</td>
<td>LAPACKE_igeconA</td>
<td></td>
</tr>
<tr>
<td>SGETRIA</td>
<td>DGETRIA</td>
<td>&quot;SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD (General Matrix Inverse, Condition Number Reciprocal, and Determinant)&quot; on page 565</td>
</tr>
<tr>
<td>LAPACKE_sgetriA</td>
<td>LAPACKE_dgetriA</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgetriA</td>
<td>LAPACKE_zgetriA</td>
<td></td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 1986, 2016
<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLANGE&lt;sup&gt;+&lt;/sup&gt;</td>
<td>DLANGE&lt;sup&gt;+&lt;/sup&gt;</td>
<td>&quot;SLANGE, DLANGE, CLANGE, and ZLANGE (General Matrix Norm)&quot; on page 572</td>
</tr>
<tr>
<td>LAPACKE_slange&lt;sup&gt;+&lt;/sup&gt;</td>
<td>LAPACKE_dlane&lt;sup&gt;+&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_clange&lt;sup&gt;+&lt;/sup&gt;</td>
<td>LAPACKE_zlane&lt;sup&gt;+&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SPPSV&lt;sup&gt;+&lt;/sup&gt;</td>
<td>DPPSV&lt;sup&gt;+&lt;/sup&gt;</td>
<td>&quot;SPPSV, DPSSV, CPPSV, and ZPPSVV (Positive Definite Real Symmetric and Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)&quot; on page 575</td>
</tr>
<tr>
<td>LAPACKE_sppsv&lt;sup&gt;+&lt;/sup&gt;</td>
<td>LAPACKE_dpssv&lt;sup&gt;+&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cppsv&lt;sup&gt;+&lt;/sup&gt;</td>
<td>LAPACKE_zpssv&lt;sup&gt;+&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SPOSV&lt;sup&gt;+&lt;/sup&gt;</td>
<td>DPOSV&lt;sup&gt;+&lt;/sup&gt;</td>
<td>&quot;SPOSV, DPOSV, CPOSV, and ZPOSV (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)&quot; on page 581</td>
</tr>
<tr>
<td>LAPACKE_sposv&lt;sup&gt;+&lt;/sup&gt;</td>
<td>LAPACKE_dpsov&lt;sup&gt;+&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cppsov&lt;sup&gt;+&lt;/sup&gt;</td>
<td>LAPACKE_zpsov&lt;sup&gt;+&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SPOTRF&lt;sup&gt;+&lt;/sup&gt;</td>
<td>DPOTRF&lt;sup&gt;+&lt;/sup&gt;</td>
<td>&quot;SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF, ZPTTRF, SPPE, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)&quot; on page 587</td>
</tr>
<tr>
<td>LAPACKEスポット&lt;sup&gt;+&lt;/sup&gt;</td>
<td>LAPACKE_dpotr&lt;sup&gt;+&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpotr&lt;sup&gt;+&lt;/sup&gt;</td>
<td>LAPACKE_zpotr&lt;sup&gt;+&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spptr&lt;sup&gt;+&lt;/sup&gt;</td>
<td>LAPACKE_dpptr&lt;sup&gt;+&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpptr&lt;sup&gt;+&lt;/sup&gt;</td>
<td>LAPACKE_zpptr&lt;sup&gt;+&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SPOCON&lt;sup&gt;+&lt;/sup&gt;</td>
<td>DPOCON&lt;sup&gt;+&lt;/sup&gt;</td>
<td>&quot;SPOCON, DPOCON, CPOCON, ZPOCON, SPPCON, DPPCON, CPPCON, ZPPCON, and ZPPOCON (Estimate the Reciprocal of the Condition Number of a Positive Definite Real Symmetric or Complex Hermitian Matrix)&quot; on page 611</td>
</tr>
<tr>
<td>LAPACKE_spcon&lt;sup&gt;+&lt;/sup&gt;</td>
<td>LAPACKE_dpcon&lt;sup&gt;+&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpocon&lt;sup&gt;+&lt;/sup&gt;</td>
<td>LAPACKE_zpcon&lt;sup&gt;+&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sppo&lt;sup&gt;+&lt;/sup&gt;</td>
<td>LAPACKE_dpoph&lt;sup&gt;+&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cppcon&lt;sup&gt;+&lt;/sup&gt;</td>
<td>LAPACKE_zpoph&lt;sup&gt;+&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SPOTRI&lt;sup&gt;+&lt;/sup&gt;</td>
<td>DPOTRI&lt;sup&gt;+&lt;/sup&gt;</td>
<td>&quot;SPOTRI, DPOTRI, CPOTRI, ZPOTRI, SPOICD, DPOICD, CPOICD, ZPOICD, SPPTRI, DPPTRI, CPPTRI, ZPTTRI, SPPE, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Inverse, Condition Number Reciprocal, and Determinant)&quot; on page 626</td>
</tr>
<tr>
<td>LAPACKEスポット&lt;sup&gt;+&lt;/sup&gt;</td>
<td>LAPACKE_dpotr&lt;sup&gt;+&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpotr&lt;sup&gt;+&lt;/sup&gt;</td>
<td>LAPACKE_zpotr&lt;sup&gt;+&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spptr&lt;sup&gt;+&lt;/sup&gt;</td>
<td>LAPACKE_dpptr&lt;sup&gt;+&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpptr&lt;sup&gt;+&lt;/sup&gt;</td>
<td>LAPACKE_zpptr&lt;sup&gt;+&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SLANSY&lt;sup&gt;+&lt;/sup&gt;</td>
<td>DLANSY&lt;sup&gt;+&lt;/sup&gt;</td>
<td>&quot;SLANSY, DLANSY, CLANHE, ZLANHE, SLANSP, DLANSP, CLANHP and ZLANHP (Real Symmetric or Complex Hermitian Matrix Norm)&quot; on page 637</td>
</tr>
<tr>
<td>LAPACKE_slansy&lt;sup&gt;+&lt;/sup&gt;</td>
<td>LAPACKE_dlan&lt;sup&gt;+&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_clansy&lt;sup&gt;+&lt;/sup&gt;</td>
<td>LAPACKE_zlan&lt;sup&gt;+&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_slansp&lt;sup&gt;+&lt;/sup&gt;</td>
<td>LAPACKE_dlan&lt;sup&gt;+&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_clansp&lt;sup&gt;+&lt;/sup&gt;</td>
<td>LAPACKE_zlan&lt;sup&gt;+&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_clanhp&lt;sup&gt;+&lt;/sup&gt;</td>
<td>LAPACKE_zlan&lt;sup&gt;+&lt;/sup&gt;</td>
<td></td>
</tr>
</tbody>
</table>
Table 124. List of LAPACK Dense Linear Algebraic Equation Subroutines (continued)

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSYSV&lt;sup&gt;A&lt;/sup&gt;</td>
<td>DSYSV&lt;sup&gt;A&lt;/sup&gt;</td>
<td>SSYSV, DSYSV, CSYSV, ZSYSV, ZHSYV, CHESV, ZHESV, SSPSV, DSSPV, SSPSV, ZSPSV, CHPSV, and ZHPSV (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)” on page 642</td>
</tr>
<tr>
<td>CSYSV&lt;sup&gt;A&lt;/sup&gt;</td>
<td>ZSYSV&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CHESV&lt;sup&gt;A&lt;/sup&gt;</td>
<td>ZHSYV&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SSPSV&lt;sup&gt;A&lt;/sup&gt;</td>
<td>DSSPSV&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CSPSV&lt;sup&gt;A&lt;/sup&gt;</td>
<td>ZSPSV&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CHPSV&lt;sup&gt;A&lt;/sup&gt;</td>
<td>ZHPSV&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_sssysv&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_dsysv&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_csysv&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_zsysv&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_chesv&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_zhesv&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_sspsv&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_dspsv&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_cspsv&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_zspsv&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_chpsv&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_zhpsv&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SSYTRF&lt;sup&gt;A&lt;/sup&gt;</td>
<td>DSYTRF&lt;sup&gt;A&lt;/sup&gt;</td>
<td>SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, ZHETRF, SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF, and ZHPTRF (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Factorization)” on page 651</td>
</tr>
<tr>
<td>CSYTRF&lt;sup&gt;A&lt;/sup&gt;</td>
<td>ZSYTRF&lt;sup&gt;B&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CHETRF&lt;sup&gt;A&lt;/sup&gt;</td>
<td>ZHETRF&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SSPTRF&lt;sup&gt;A&lt;/sup&gt;</td>
<td>DSPTRF&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CSPTRF&lt;sup&gt;A&lt;/sup&gt;</td>
<td>ZSPTRF&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CHPTRF&lt;sup&gt;A&lt;/sup&gt;</td>
<td>ZHPTRF&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_ssytrf&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_dsytrf&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_csytrf&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_zsytrf&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_chetr&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_zhetr&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_ssptrf&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_dsptrf&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_csptrf&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_zsptrf&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_chptrf&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_zhptrf&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SSYTRS&lt;sup&gt;A&lt;/sup&gt;</td>
<td>DSYTRS&lt;sup&gt;A&lt;/sup&gt;</td>
<td>SSYTRS, DSYTRS, CSYTRS, ZSYTRS, CHETRS, ZHETRS, SSPTRS, DSPTRS, CSPTRRS, ZSPTRS, CHPTRRS, and ZHPTRRS (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)” on page 660</td>
</tr>
<tr>
<td>CSYTRS&lt;sup&gt;A&lt;/sup&gt;</td>
<td>ZSYTRS&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CHETRS&lt;sup&gt;A&lt;/sup&gt;</td>
<td>ZHETRS&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SSPTRS&lt;sup&gt;A&lt;/sup&gt;</td>
<td>DSPTRS&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CSPTRS&lt;sup&gt;A&lt;/sup&gt;</td>
<td>ZSPTRS&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CHPTRS&lt;sup&gt;A&lt;/sup&gt;</td>
<td>ZHPTRS&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_sssytrs&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_dsytrs&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_csytrs&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_zsytrs&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_chetrs&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_zhetrs&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_ssptrs&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_dsptrs&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_csptrs&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_zsptrs&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_chptrs&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_zhptrs&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>STRTRI&lt;sup&gt;A&lt;/sup&gt;</td>
<td>DTRTRI&lt;sup&gt;A&lt;/sup&gt;</td>
<td>STRTRI, DTRTRI, CTRTRI, ZTRTRI, STPTRI, DTPTRI, CTPTRI, and ZTPTRI (Triangular Matrix Inverse)” on page 682</td>
</tr>
<tr>
<td>CTRTRI&lt;sup&gt;A&lt;/sup&gt;</td>
<td>ZTRTRI&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>STPTRI&lt;sup&gt;A&lt;/sup&gt;</td>
<td>DTPTRI&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CTPTRI&lt;sup&gt;A&lt;/sup&gt;</td>
<td>ZTPTRI&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_sstrtri&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_dtrtri&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_stptri&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_ztrtri&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_cstrtri&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_ztrtri&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_cctptri&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_zctptri&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SLANTR&lt;sup&gt;A&lt;/sup&gt;</td>
<td>DLANTR&lt;sup&gt;A&lt;/sup&gt;</td>
<td>SLANTR, DLANTR, CLANTR, ZLANTR, SLANTP, DLANTP, CLANTP, and ZLANTP (Trapezoidal or Triangular Matrix Norm)” on page 690</td>
</tr>
<tr>
<td>CLANTR&lt;sup&gt;A&lt;/sup&gt;</td>
<td>ZLANTR&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SLANTP&lt;sup&gt;A&lt;/sup&gt;</td>
<td>DLANTP&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CLANTP&lt;sup&gt;A&lt;/sup&gt;</td>
<td>ZLANTP&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_slantr&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_dlantr&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_clantr&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_zlantr&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_slantr&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_dlantr&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACK_clantr&lt;sup&gt;A&lt;/sup&gt;</td>
<td>LAPACK_zlantr&lt;sup&gt;A&lt;/sup&gt;</td>
<td></td>
</tr>
</tbody>
</table>

<sup>A</sup> LAPACK
### Table 125. List of Dense Linear Algebraic Equation Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGEF</td>
<td>DGEF</td>
<td>“SGEF, DGEF, CGEF, and ZGEF (General Matrix Factorization)” on page 544</td>
</tr>
<tr>
<td>CGEF</td>
<td>ZGEF</td>
<td></td>
</tr>
<tr>
<td>DGEFP§</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SGESM</td>
<td>DGESM</td>
<td>“SGESM, DGESM, CGESM, and ZGESM (General Matrix, Its Transpose, or Its Conjugate Transpose Multiple Right-Hand Side Solve)” on page 531</td>
</tr>
<tr>
<td>CGESM</td>
<td>ZGESM</td>
<td></td>
</tr>
<tr>
<td>SGEFCD</td>
<td>DGEFCD</td>
<td>“SGEFCD and DGEFCD (General Matrix Factorization, Condition Number Reciprocal, and Determinant)” on page 561</td>
</tr>
<tr>
<td>SGEICD</td>
<td>DGEICD</td>
<td>“SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD (General Matrix Inverse, Condition Number Reciprocal, and Determinant)” on page 565</td>
</tr>
<tr>
<td>SPOF</td>
<td>DPOF</td>
<td>“SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPPF, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)” on page 587</td>
</tr>
<tr>
<td>CPOF</td>
<td>ZPOF</td>
<td></td>
</tr>
<tr>
<td>SPPF</td>
<td>DPPF</td>
<td>“SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPPF, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)” on page 587</td>
</tr>
<tr>
<td>SPOFCD</td>
<td>DPOFCD</td>
<td>“SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPPF, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)” on page 587</td>
</tr>
<tr>
<td>CPOFCD</td>
<td>DPOFCD</td>
<td>“SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPPF, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)” on page 587</td>
</tr>
<tr>
<td>DPPFCD</td>
<td>DPOFCD</td>
<td>“SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPPF, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)” on page 587</td>
</tr>
<tr>
<td>SPPICD</td>
<td>DPOICD</td>
<td>“SPOTRI, DPOTRI, CPOTRI, ZPOTRI, SPOICD, DPOICD, SPPIICD, DPPIICD, CPPTRI, ZPPTRI, SPPICD, and DPPICD (Positive Definite Real Symmetric or Complex Hermitian Matrix Inverse, Condition Number Reciprocal, and Determinant)” on page 626</td>
</tr>
<tr>
<td>SPOICD</td>
<td>DPOICD</td>
<td>“SPOTRI, DPOTRI, CPOTRI, ZPOTRI, SPOICD, DPOICD, SPPIICD, DPPIICD, CPPTRI, ZPPTRI, SPPICD, and DPPICD (Positive Definite Real Symmetric or Complex Hermitian Matrix Inverse, Condition Number Reciprocal, and Determinant)” on page 626</td>
</tr>
<tr>
<td>DBSSV</td>
<td></td>
<td>“DBSSV (Symmetric Indefinite Matrix Factorization and Multiple Right-Hand Side Solve)” on page 667</td>
</tr>
<tr>
<td>DBSTRF</td>
<td></td>
<td>“DBSTRF (Symmetric Indefinite Matrix Factorization)” on page 673</td>
</tr>
<tr>
<td>DBSTRS</td>
<td></td>
<td>“DBSTRS (Symmetric Indefinite Matrix Multiple Right-Hand Side Solve)” on page 678</td>
</tr>
<tr>
<td>STRF§</td>
<td>DTRF§</td>
<td>“STRTRI, DTRTRI, CTRTRI, ZTRTRI, STPTRI, DTPTRI, CTPTRI, and ZTPTRI (Triangular Matrix Inverse)” on page 682</td>
</tr>
<tr>
<td>STPI§</td>
<td>DTPI§</td>
<td></td>
</tr>
</tbody>
</table>

§ 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.

### Banded Linear Algebraic Equation Subroutines

The banded linear algebraic equation subroutines provide solutions to linear systems of equations for:
- Real or complex general band matrices
- Positive definite real symmetric or complex Hermitian band matrices
- Real or complex general tridiagonal matrices
- Positive definite real symmetric or complex Hermitian tridiagonal matrices
Table 126. List of LAPACK Banded Linear Algebraic Equation Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGBSV^a</td>
<td>DGBSV^a</td>
<td>&quot;SGBSV, DGBSV, CGBSV, and ZGBSV (General Band Matrix Factorization and Multiple Right-Hand Side Solve)&quot; on page 698</td>
</tr>
<tr>
<td>SGBTF^a</td>
<td>DGBTF^a</td>
<td>&quot;SGBTTF, DGBTTF, CGBTTF and ZGBTTF (General Band Matrix Factorization)&quot; on page 702</td>
</tr>
<tr>
<td>SGBTR^a</td>
<td>DGBTR^a</td>
<td>&quot;SGBTTR, DGBTTR, CGBTTR, and ZGBTTR (General Band Matrix Factorization and Multiple Right-Hand Side Solve)&quot; on page 706</td>
</tr>
<tr>
<td>SPBSV^a</td>
<td>DPBSV^a</td>
<td>&quot;SPBSV, DPBSV, CPBSV, and ZPBSV (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization and Multiple Right-Hand Side Solve)&quot; on page 715</td>
</tr>
<tr>
<td>SPTTF^a</td>
<td>DPTTF^a</td>
<td>&quot;SPTTTF, DPTTTF, CPBTTF, and ZPTTTF (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Multiple Right-Hand Side Solve)&quot; on page 721</td>
</tr>
<tr>
<td>SPTTR^a</td>
<td>DPTTR^a</td>
<td>&quot;SPTTTR, DPBTTR, CPBTTR and ZPTTTR (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization)&quot; on page 726</td>
</tr>
<tr>
<td>SPTBSV^a</td>
<td>DPBTSV^a</td>
<td>&quot;SPTBSV, DPBTSV, CPBTSV, and ZPBSV (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization and Multiple Right-Hand Side Solve)&quot; on page 731</td>
</tr>
<tr>
<td>SPTTF^a</td>
<td>DPBTF^a</td>
<td>&quot;SPTTTF, DPBTTF, CPBTTF, and ZPTTTF (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization)&quot; on page 735</td>
</tr>
<tr>
<td>SPTTR^a</td>
<td>DPBTR^a</td>
<td>&quot;SPTTTR, DPBTTR, CPBTTR, and ZPTTTR (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization)&quot; on page 740</td>
</tr>
<tr>
<td>SPTBSV^a</td>
<td>DPBSV^a</td>
<td>&quot;SPTBSV, DPBSV, CPBSV, and ZPBSV (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization)&quot; on page 746</td>
</tr>
<tr>
<td>SPTTF^a</td>
<td>DPBTF^a</td>
<td>&quot;SPTTTF, DPBTTF, CPBTTF, and ZPTTTF (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization)&quot; on page 751</td>
</tr>
<tr>
<td>SPTTR^a</td>
<td>DPBTR^a</td>
<td>&quot;SPTTTR, DPBTTR, CPBTTR, and ZPTTTR (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization)&quot; on page 755</td>
</tr>
<tr>
<td>SPTBSV^a</td>
<td>DPBSV^a</td>
<td>&quot;SPTBSV, DPBSV, CPBSV, and ZPBSV (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization)&quot; on page 760</td>
</tr>
</tbody>
</table>

^ LAPACK
Table 127. List of non-LAPACK Banded Linear Algebraic Equation Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGBF§</td>
<td>DGBF§</td>
<td>“SGBF and DGBF (General Band Matrix Factorization)” on page 761</td>
</tr>
<tr>
<td>SGBS§</td>
<td>DGBS§</td>
<td>“SGBS and DGBS (General Band Matrix Solve)” on page 712</td>
</tr>
<tr>
<td>SPBF§, SPBCHF§</td>
<td>DPBF§, DPBCHF§</td>
<td>“SPBF, DPBF, SPBCHF, and DPBCHF (Positive Definite Symmetric Band Matrix Factorization)” on page 768</td>
</tr>
<tr>
<td>SPBS§, SPBCHS§</td>
<td>DPBS§, DPBCHS§</td>
<td>“SPBS, DPBS, SPBCHS, and DPBCHS (Positive Definite Symmetric Band Matrix Solve)” on page 772</td>
</tr>
<tr>
<td>SGTF§</td>
<td>DGTF§</td>
<td>“SGTF and DGTF (General Tridiagonal Matrix Factorization)” on page 775</td>
</tr>
<tr>
<td>SGTS§</td>
<td>DGTTS§</td>
<td>“SGTS and DGTTS (General Tridiagonal Matrix Solve)” on page 778</td>
</tr>
<tr>
<td>SGTNPS, CGTNPS</td>
<td>DGTNPS, ZGTNPS</td>
<td>“SGTNPS, DGTNPS, ZGTNPS, CGTNPS, and ZGTNPS (General Tridiagonal Matrix Combined Factorization and Solve with No Pivoting)” on page 780</td>
</tr>
<tr>
<td>SPTF§</td>
<td>DPTF§</td>
<td>“SPTF and DPTF (Positive Definite Symmetric Tridiagonal Matrix Factorization)” on page 789</td>
</tr>
<tr>
<td>SPTS§</td>
<td>DPTS§</td>
<td>“SPTS and DPTS (Positive Definite Symmetric Tridiagonal Matrix Solve)” on page 791</td>
</tr>
</tbody>
</table>

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

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 128. List of Sparse Linear Algebraic Equation Subroutines

<table>
<thead>
<tr>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGSF</td>
<td>“DGSF (General Sparse Matrix Factorization Using Storage by Indices, Rows, or Columns)” on page 794</td>
</tr>
<tr>
<td>DGSS</td>
<td>“DGSS (General Sparse Matrix or Its Transpose Solve Using Storage by Indices, Rows, or Columns)” on page 800</td>
</tr>
<tr>
<td>DGKFS, DGKFS§</td>
<td>“DGKFS (General Sparse Matrix or Its Transpose Factorization, Determinant, and Solve Using Skyline Storage Mode)” on page 804</td>
</tr>
<tr>
<td>DSKFS, DSKFS§</td>
<td>“DSKFS (Symmetric Sparse Matrix Factorization, Determinant, and Solve Using Skyline Storage Mode)” on page 821</td>
</tr>
<tr>
<td>DSRIS</td>
<td>“DSRIS (Iterative Linear System Solver for a General or Symmetric Sparse Matrix Stored by Rows)” on page 839</td>
</tr>
<tr>
<td>DSMCG†</td>
<td>“DSMCG (Sparse Positive Definite or Negative Definite Symmetric Matrix Iterative Solve Using Compressed-Matrix Storage Mode)” on page 850</td>
</tr>
<tr>
<td>DSDCG</td>
<td>“DSDCG (Sparse Positive Definite or Negative Definite Symmetric Matrix Iterative Solve Using Compressed-Diagonal Storage Mode)” on page 858</td>
</tr>
</tbody>
</table>
Table 128. List of Sparse Linear Algebraic Equation Subroutines (continued)

<table>
<thead>
<tr>
<th>Long-precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSGMCG‡</td>
<td>“DSMGCG (General Sparse Matrix Iterative Solve Using Compressed-Matrix Storage Mode)” on page 866</td>
</tr>
<tr>
<td>DSDGCG</td>
<td>“DSDGCG (General Sparse Matrix Iterative Solve Using Compressed-Diagonal Storage Mode)” on page 873</td>
</tr>
</tbody>
</table>

‡ 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 Subroutines

The linear least squares subroutines provide least squares solutions to linear systems of equations for general matrices using a QR factorization or a singular value decomposition. Some of these subroutines correspond to the LAPACK routines described in reference [8 on page 1363].

Table 129. List of LAPACK Linear Least Squares Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGEQRFa</td>
<td>DGEQRFa</td>
<td>“SGEQRF, DGEQRF, CGEQRF, and ZGEQRF (General Matrix QR Factorization)” on page 895</td>
</tr>
<tr>
<td>LAPACK_SGEqrfa</td>
<td>LAPACK_DGEqrfa</td>
<td></td>
</tr>
<tr>
<td>LAPACK_SGEqrfb</td>
<td>LAPACK_ZGEqrfb</td>
<td></td>
</tr>
<tr>
<td>SGELSA</td>
<td>DGELSA</td>
<td>“SGELS, DGELS, CGELS, and ZGELS (Linear Least Squares Solution for a General Matrix)” on page 901</td>
</tr>
<tr>
<td>LAPACK_SGELSA</td>
<td>LAPACK_DGELSA</td>
<td></td>
</tr>
<tr>
<td>LAPACK_SGELSB</td>
<td>LAPACK_DGELSB</td>
<td></td>
</tr>
<tr>
<td>SGEVSFb</td>
<td>DGESVFb</td>
<td>“SGESVF and DGESVF (Singular Value Decomposition for a General Matrix)” on page 919</td>
</tr>
<tr>
<td>SGEVSFs</td>
<td>DGESVSs</td>
<td>“SGESVS and DGESVS (Linear Least Squares Solution for a General Matrix Using the Singular Value Decomposition)” on page 927</td>
</tr>
</tbody>
</table>

Table 130. List of Non–LAPACK Linear Least Squares Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGESVFb</td>
<td>DGESVFb</td>
<td></td>
</tr>
<tr>
<td>SGEVSFs</td>
<td>DGESVSs</td>
<td></td>
</tr>
</tbody>
</table>

§ LAPACK
Table 130. List of Non–LAPACK Linear Least Squares Subroutines (continued)

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGELLS$^+$</td>
<td>DGELLS$^+$</td>
<td>&quot;SGELLS and DGELLS (Linear Least Squares Solution for a General Matrix with Column Pivoting)&quot; on page 932</td>
</tr>
</tbody>
</table>

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

Dense and Banded Linear Algebraic Equation Considerations

This provides some key points about using the dense and banded linear algebraic equation subroutines.

Use Considerations

To solve a system of equations, you have two choices:

- Use the combined factorization-and-solve subroutine for the type of matrix you have.
- Use both the factorization subroutine and the solve subroutine for the type of matrix you have. When doing so, note the following:
  - 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.
  - 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.

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 when the Altivec or VSX unit is not used. 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 64.

Sparse Matrix Direct Solver Considerations

This 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 \( l\text{na} > 2nz \) 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 \( l\text{na} \) 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 \( j\text{opt} \) (such as \( j\text{opt} = 10 \) 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 (\( \text{RPARM}(1) \)) 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 (\( \text{RPARM}(2) \)) 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 \( l\text{na} \).

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 64.

**Sparse Matrix Skyline Solver Considerations**

This 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 RPARAM(10) through RPARAM(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 RPARAM(11) through RPARAM(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 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 116.

   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.

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, \((A+A^\top)\)/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 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 \(AX=\mathbf{B}\), 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 when the AltiVec or VSX unit is not used. 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 64.
Dense Linear Algebraic Equation Subroutines

This contains the dense linear algebraic equation subroutine descriptions.
SGESV, DGESV, CGESV, ZGESV (General Matrix Factorization and Multiple Right-Hand Side Solve)

Purpose

These subroutines solve the system of linear equations $AX = B$ for $X$, where $A$, $B$, and $X$ are general matrices.

The matrix $A$ is factored using Gaussian elimination with partial pivoting.

<table>
<thead>
<tr>
<th>A, B</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGESV</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGESV</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGESV</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGESV</td>
</tr>
</tbody>
</table>

.syntax

Fortran

CALL SGESV | DGESV | CGESV | ZGESV (n, nrhs, a, ipvt, bx, ldb, info )

C and C++

sgesv | dgesv | cgesv | zgesv (n, nrhs, a, lda, ipvt, bx, ldb, info );

LAPACK

info = LAPACKE_sgesv | LAPACKE_dgesv | LAPACKE_cgesv | LAPACKE_zgesv

(matrix_layout, n, nrhs, a, lda, ipvt, bx, ldb);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

• If matrix_layout = LAPACK_ROW_MAJOR, the matrices are stored in row major order.

• If matrix_layout = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. It must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR

n is the order $n$ of matrix $A$ and the number of rows of matrix $B$.

Specified as: an integer; $n \geq 0$, $n \leq lda$, and $n \leq ldb$.

nrhs is the number of right-hand sides; that is, the number of columns of matrix $B$.

Specified as: an integer; $nrhs \geq 0$.

a is the general matrix $A$ to be factored.

Specified as: an lda by (at least) $n$ array, containing numbers of the data type indicated in Table 131

lda is the leading dimension of the array specified for $A$.

Specified as: an integer; $lda > 0$ and $lda \geq n$. 

530 ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
$ipvt$
See On Return.

$bx$ is the general 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 131 on page 530.

$ldb$
is the leading dimension of the array specified for $B$.
Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

$info$
See On Return.

On Return

$a$ is the transformed matrix $A$ of order $n$, containing the results of the factorization.
Returned as: an $lda$ by (at least) $n$ array, containing numbers of the data type indicated in Table 131 on page 530. See “Function” on page 532.

$ipvt$
is the integer vector of length $n$, containing the pivot indices.
Returned as: a one-dimensional array of (at least) length $n$, containing integers, where $1 \leq ipvt(i) \leq n$.

$bx$ is the matrix $X$, containing the $nrhs$ solutions to the system. The solutions, each of length $n$, reside in the columns of $X$.
Returned as: an $ldb$ by (at least) $nrhs$ array, containing numbers of the data type indicated in Table 131 on page 530.

$info$
has the following meaning:
If $info = 0$, the subroutine completed successfully.
If $info > 0$, the factorization was unsuccessful and the solution was not computed. $info$ is set equal to the first $i$ where $U_{ii}$ is singular and its inverse could not be computed.

Returned as:
- For SGESV, DGESV, CGESV, and ZGESV returned as: an integer; $info \geq 0$.
- For LAPACKE_sgesv, LAPACKE_dgesv, LAPACKE_cgesv, and LAPACKE_zgesv returned as an integer function value; $info \geq 0$.

Notes
1. In your C program, argument $info$ must be passed by reference.
2. The matrices and vector used in this computation must have no common elements; otherwise, results are unpredictable.
3. The way these subroutines handle singularity differs from LAPACK. Like LAPACK, these subroutines use the $info$ argument to provide information about the singularity of $A$, but they also provide an error message.
4. On both input and output, matrices $A$ and $B$ conform to LAPACK format.
Function

These subroutines solve the system of linear equations $AX = B$ for $X$, where $A$, $B$, and $X$ are general matrices.

The matrix $A$ is factored using Gaussian elimination with partial pivoting to compute the $LU$ factorization of $A$, where:

$$A = PLU$$

and

- $L$ is a unit lower triangular matrix.
- $U$ is an upper triangular matrix.
- $P$ is the permutation matrix.

If $n$ is 0, no computation is performed and the subroutine returns after doing some parameter checking. If $n > 0$ and $nrhs$ is 0, no solutions are computed and the subroutine returns after factoring the matrix.

See references [8 on page 1363], [44 on page 1366], and [73 on page 1367].

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 $U_{ii} = 0$ 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. $n < 0$
2. $nrhs < 0$
3. $n > lda$
4. $lda \leq 0$
5. $n > ldb$
6. $ldb \leq 0$

Examples

Example 1

This example shows how to solve the system $AX = B$, where:

Matrix $A$ is the same used as input in Example 1 for DGETRF.
Matrix $B$ is the same used as input in Example 1 for DGETRS.

Call Statement and Input:

```plaintext
CALL DGESV( 9 , 5 , A , 9 , IPVT , BX , 9 , INFO)
```
Example 2

This example shows how to solve the system \( AX = B \), where:

Matrix \( A \) is the same used as input in Example 2 for ZGETRF.
Matrix \( B \) is the same used as input in Example 2 for ZGETRS.

Call Statement and Input:

\[
\begin{array}{cccccccc}
\text{CALL ZGESV}\left( 9, 5, A, 9, IPVT, BX, 9, \text{INFO} \right)
\end{array}
\]

\( A = \text{(same as input } A \text{ in Example 2)} \)
\( IPVT = \text{(same as input } IPVT \text{ in Example 2)} \)
\( BX = \text{(same as input } BX \text{ in Example 2)} \)

Output:

\[
\begin{bmatrix}
(1.0,1.0) & (1.0,2.0) & (1.0,3.0) & (1.0,4.0) & (1.0,5.0) \\
(2.0,1.0) & (2.0,2.0) & (2.0,3.0) & (2.0,4.0) & (2.0,5.0) \\
(3.0,1.0) & (3.0,2.0) & (3.0,3.0) & (3.0,4.0) & (3.0,5.0) \\
(4.0,1.0) & (4.0,2.0) & (4.0,3.0) & (4.0,4.0) & (4.0,5.0) \\
(5.0,1.0) & (5.0,2.0) & (5.0,3.0) & (5.0,4.0) & (5.0,5.0) \\
(6.0,1.0) & (6.0,2.0) & (6.0,3.0) & (6.0,4.0) & (6.0,5.0) \\
(7.0,1.0) & (7.0,2.0) & (7.0,3.0) & (7.0,4.0) & (7.0,5.0) \\
(8.0,1.0) & (8.0,2.0) & (8.0,3.0) & (8.0,4.0) & (8.0,5.0) \\
(9.0,1.0) & (9.0,2.0) & (9.0,3.0) & (9.0,4.0) & (9.0,5.0)
\end{bmatrix}
\]

\( BX = \)

\[
\begin{bmatrix}
1.0 & 2.0 & 3.0 & 4.0 & 5.0 \\
2.0 & 4.0 & 6.0 & 8.0 & 10.0 \\
3.0 & 6.0 & 9.0 & 12.0 & 15.0 \\
4.0 & 8.0 & 12.0 & 16.0 & 20.0 \\
5.0 & 10.0 & 15.0 & 20.0 & 25.0 \\
6.0 & 12.0 & 18.0 & 24.0 & 30.0 \\
7.0 & 14.0 & 21.0 & 28.0 & 35.0 \\
8.0 & 16.0 & 24.0 & 32.0 & 40.0 \\
9.0 & 18.0 & 27.0 & 36.0 & 45.0
\end{bmatrix}
\]

\( INFO = 0 \)
SGETRF, DGETRF, CGETRF and ZGETRF (General Matrix Factorization)

Purpose

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 SGETRI, DGETRI, CGETRI, or ZGETRI, respectively.

To estimate the reciprocal of the condition number of matrix \( A \), follow the call to these subroutines with a call to SGECON, DGECON, CGECON, or ZGECON, respectively.

Table 132. Data Types

<table>
<thead>
<tr>
<th>( A )</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGETRF(^\d)</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGETRF(^\d)</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGETRF(^\d)</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGETRF(^\d)</td>
</tr>
</tbody>
</table>

\(^\d\) LAPACK

Note: The output from each of these subroutines should be used only as input for specific other subroutines, as shown in the table below.

<table>
<thead>
<tr>
<th>Output from this subroutine:</th>
<th>Should be used only as input to the following subroutines:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solve</td>
<td>Inverse</td>
</tr>
<tr>
<td>SGETRF</td>
<td>SGETRS</td>
</tr>
<tr>
<td>DGETRF</td>
<td>DGETRS</td>
</tr>
<tr>
<td>CGETRF</td>
<td>CGETRS</td>
</tr>
<tr>
<td>ZGETRF</td>
<td>ZGETRS</td>
</tr>
</tbody>
</table>

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);

LAPACK

info = LAPACKE_sgetrf | LAPACKE_dgetrf | LAPACKE_cgetrf | LAPACKE_zgetrf (matrix_layout, m, n, a, lda, ipvt);

On Entry

\( matrix\_layout \)

indicates whether the input and output matrices are stored in row major order or column major order, where:
• If \( \text{matrix\_layout} = \text{LAPACK\_ROW\_MAJOR} \), the matrices are stored in row major order.
• If \( \text{matrix\_layout} = \text{LAPACK\_COL\_MAJOR} \), the matrices are stored in column major order.

Specified as: an integer. It must be LAPACK\_ROW\_MAJOR or LAPACK\_COL\_MAJOR

\( m \) the number of rows in general matrix \( A \) used in the computation.  
Specified as: an integer; \( 0 \leq m \leq \text{lda} \).

\( n \) the number of columns in general matrix \( A \) used in the computation.  
Specified as: an integer; \( n \geq 0 \).

\( a \) is the \( m \) by \( n \) general matrix \( A \) to be factored.  
Specified as: an \( \text{lda} \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 132 on page 534.

\( \text{lda} \)  
is the leading dimension of matrix \( A \).  
Specified as: an integer; \( \text{lda} > 0 \) and \( \text{lda} \geq m \).

\( \text{ipvt} \)  
See On Return

\( \text{info} \)  
See On Return

On Return  
\( a \) is the \( m \) by \( n \) transformed matrix \( A \), containing the results of the factorization.  
See “Function” on page 536. Returned as: an \( \text{lda} \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 132 on page 534.

\( \text{ipvt} \)  
is the integer vector \( \text{ipvt} \) of length \( \min(m,n) \), containing the pivot indices.  
Returned as: a one-dimensional array of (at least) length \( \min(m,n) \), containing integers, where \( 1 \leq \text{ipvt}(i) \leq m \).

\( \text{info} \) has the following meaning:  
If \( \text{info} = 0 \), the factorization of general matrix \( A \) completed successfully.

If \( \text{info} > 0 \), \( \text{info} \) is set equal to the first \( i \) where \( U_{ii} \) is singular and its inverse could not be computed.

Returned as:
• For SGETRF, DGETRF, CGETRF and ZGETRF returned as: an integer; \( \text{info} \geq 0 \).
• For LAPACK\_sgetrf, LAPACK\_dgetrf, LAPACK\_cgetrf and LAPACK\_zgetrf, returned as an integer function value; \( \text{info} \geq 0 \).

Returned as: an integer; \( \text{info} \geq 0 \).

Notes  
1. In your C program, argument \( \text{info} \) must be passed by reference.
2. The matrix \( A \) and vector \( \text{ipvt} \) must have no common elements; otherwise results are unpredictable.
The way these subroutines handle singularity differs from LAPACK. Like LAPACK, these subroutines use the info argument to provide information about the singularity of A, but they also provide an error message.

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:

\[
A = PLU
\]

and

\( L \) is a unit lower triangular matrix.
\( U \) is an upper triangular matrix.
\( P \) is the permutation matrix.

On output, the transformed matrix A contains \( U \) in the upper triangle (if \( m \geq n \)) or upper trapezoid (if \( m < n \)) and \( L \) in the strict lower triangle (if \( m \leq n \)) or lower trapezoid (if \( m > n \)). \( ipvt \) contains the pivots representing permutation \( P \), such that \( A = PLU \).

If \( m \) or \( n \) is 0, no computation is performed and the subroutine returns after doing some parameter checking. See references \[8 on page 1363\], \[44 on page 1366\], and \[73 on page 1367\].

**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 \( U_{ii} = 0 \) 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. \( lda \leq 0 \)

**Examples**

**Example 1**

This example shows a factorization of a real general matrix A of order 9.

Call Statement and Input:

\[
\begin{align*}
M & \quad N & \quad A & \quad LDA & \quad IPVT & \quad INFO \\
\end{align*}
\]

\[
\text{CALL DGTRF( 9, 9, A, 9, IPVT, INFO )}
\]
Example 2

This example shows a factorization of a complex general matrix $A$ of order 9.

Call Statement and Input:

```
CALL ZGETRF( 9 , 9 , A , 9 , IPVT , INFO )
```

Output:

```
(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) (2.8, 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) (5.6, 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) (5.6, 1.0) (6.0, 1.0)
A = (3.2, 1.0) (3.6, 1.0) (4.0, 1.0) (4.4, 1.0) (4.8, 1.0) (5.2, 1.0) (5.6, 1.0) (6.0, 1.0) (6.4, 1.0)
(3.6, 1.0) (4.0, 1.0) (4.4, 1.0) (4.8, 1.0) (5.2, 1.0) (5.6, 1.0) (6.0, 1.0) (6.4, 1.0) (6.8, 1.0)
(4.0, 1.0) (4.4, 1.0) (4.8, 1.0) (5.2, 1.0) (5.6, 1.0) (6.0, 1.0) (6.4, 1.0) (6.8, 1.0) (7.2, 1.0)
(4.4, 1.0) (4.8, 1.0) (5.2, 1.0) (5.6, 1.0) (6.0, 1.0) (6.4, 1.0) (6.8, 1.0) (7.2, 1.0) (7.6, 1.0)
(4.8, 1.0) (5.2, 1.0) (5.6, 1.0) (6.0, 1.0) (6.4, 1.0) (6.8, 1.0) (7.2, 1.0) (7.6, 1.0) (8.0, 1.0)
(5.2, 1.0) (5.6, 1.0) (6.0, 1.0) (6.4, 1.0) (6.8, 1.0) (7.2, 1.0) (7.6, 1.0) (8.0, 1.0) (8.4, 1.0)
(5.6, 1.0) (6.0, 1.0) (6.4, 1.0) (6.8, 1.0) (7.2, 1.0) (7.6, 1.0) (8.0, 1.0) (8.4, 1.0) (8.8, 1.0)
```

IPVT = (9, 9, 9, 9, 9, 9, 9, 9, 9)
INFO = 0

Example 3

This example shows a factorization of a real general matrix $A$ of order 9.

Output:

```
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
```

IPVT = (9, 9, 9, 9, 9, 9, 9, 9, 9)
INFO = 0
Call Statement and Input:

```
CALL SGETRF( 9 , 9 , A , 9 , IPVT , INFO )
```

\[
A = \begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
4.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 5.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 6.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 7.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 8.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 9.0 & 1.0 & 1.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 10.0 & 11.0 & 12.0
\end{bmatrix}
\]

Output:

\[
A = \begin{bmatrix}
4.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\
0.0000 & 5.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 & 0.0000 & 0.0000 & 0.0000 \\
0.0000 & 0.0000 & 6.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 & 0.0000 & 0.0000 \\
0.0000 & 0.0000 & 0.0000 & 7.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 \\
0.0000 & 0.0000 & 0.0000 & 0.0000 & 8.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 \\
0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 9.0000 & 1.0000 & 1.0000 & 1.0000 \\
0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 10.0000 & 11.0000 & 12.0000 \\
0.2500 & 0.1500 & 0.1000 & 0.0714 & 0.0536 & -0.0694 & -0.0306 & 0.1806 & 0.3111 \\
0.2500 & 0.1500 & 0.1000 & 0.0714 & -0.0714 & -0.0556 & -0.0194 & 0.9385 & -0.0031
\end{bmatrix}
\]

\[
IPVT = ( 3 , 4 , 5 , 6 , 7 , 8 , 9 , 8 , 9 )
\]
SGETRS, DGETRS, CGETRS, and ZGETRS (General Matrix Multiple Right-Hand Side Solve)

Purpose

SGETRS and DGETRS solve one of the following systems of equations for multiple right-hand sides:
1. \( AX = B \)
2. \( A^T X = B \)

CGETRS and ZGETRS solve one of the following systems of equations for multiple right-hand sides:
1. \( AX = B \)
2. \( A^T X = B \)
3. \( A^H X = B \)

In the formulas above:
• \( A \) represents the general matrix containing the \( LU \) factorization.
• \( B \) represents the general matrix containing the right-hand sides in its columns.
• \( X \) represents the general matrix 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 133. Data Types

<table>
<thead>
<tr>
<th></th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGETRS(^\circ)</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGETRS(^\circ)</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGETRS(^\circ)</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGETRS(^\circ)</td>
</tr>
<tr>
<td>(^\circ) LAPACK</td>
<td></td>
</tr>
</tbody>
</table>

Note: The input to these solve subroutines must be the output from the factorization subroutines SGETRF, DGETRF, CGETRF and ZGETRF, respectively.

Syntax

Fortran  
\text{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);

LAPACK  
info = LAPACK\_sgetrs | LAPACK\_dgetrs | LAPACK\_cgetrs | LAPACK\_zgetrs (matrix\_layout, transa, n, nrhs, a, lda, ipvt, bx, ldb);

On Entry

\text{matrix\_layout}  
indicates whether the input and output matrices are stored in row major order or column major order, where:
• If \( \text{matrix\_layout} = \text{LAPACK\_ROW\_MAJOR} \), the matrices are stored in row major order.
If \( \text{matrix\_layout} = \text{LAPACK\_COL\_MAJOR} \), the matrices are stored in column major order.
Specified as: an integer. It must be \( \text{LAPACK\_ROW\_MAJOR} \) or \( \text{LAPACK\_COL\_MAJOR} \).

\text{transa} \ indicates the form of matrix \( A \) to use in the computation, where:
- If \( \text{transa} = 'N' \), \( A \) is used in the computation, resulting in solution 1.
- If \( \text{transa} = 'T' \), \( A^T \) is used in the computation, resulting in solution 2.
- If \( \text{transa} = 'C' \), \( A^H \) is used in the computation, resulting in solution 3.
Specified as: a single character; \( \text{transa} = 'N' \), 'T', or 'C'.

\( n \) is the order of factored matrix \( A \) and the number of rows in matrix \( B \).
Specified as: an integer; \( n \geq 0 \).

\( \text{nrhs} \) is the number of right-hand sides—that is, the number of columns in matrix \( B \) used in the computation.
Specified as: an integer; \( \text{nrhs} \geq 0 \).

\( a \) is the factorization of matrix \( A \), produced by a preceding call to \text{SGGETRF}, \text{DGETRF}, \text{CGETRF}, or \text{ZGETRF}, respectively.
Specified as: an \( \text{lda} \) by (at least) \( n \) array, containing numbers of the data type indicated in \text{Table 133 on page 539}.

\( \text{lda} \) is the leading dimension of the array specified for \( a \).
Specified as: an integer; \( \text{lda} > 0 \) and \( \text{lda} \geq n \).

\( \text{ipvt} \) is the integer vector \( \text{ipvt} \) of length \( n \), containing the pivot indices produced by a preceding call to \text{SGGETRF}, \text{DGETRF}, \text{CGETRF}, or \text{ZGETRF}, respectively.
Specified as: a one-dimensional array of (at least) length \( n \), containing integers, where \( 1 \leq \text{ipvt}(i) \leq n \).

\( bx \) is the general matrix \( B \), containing the \( \text{nrhs} \) right-hand sides of the system. The right-hand sides, each of length \( n \), reside in the columns of matrix \( B \).
Specified as: an \( \text{ldb} \) by (at least) \( \text{nrhs} \) array, containing numbers of the data type indicated in \text{Table 133 on page 539}.

\( \text{ldb} \) is the leading dimension of the array specified for \( B \).
Specified as: an integer; \( \text{ldb} > 0 \) and \( \text{ldb} \geq n \).

\text{info} \ See \text{On Return}.

\text{On Return}

\( bx \) is the matrix \( X \), containing the \( \text{nrhs} \) solutions to the system. The solutions, each of length \( n \), reside in the columns of \( X \).
Returned as: an \( \text{ldb} \) by (at least) \( \text{nrhs} \) array, containing numbers of the data type indicated in \text{Table 133 on page 539}. 

540 ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
info

info has the following meaning:

If info = 0, the solve of general matrix A completed successfully.

Returned as:

- For SGETRS, DGETRS, CGETRS, and ZGETRS, returned as: an integer; info ≥ 0.
- For LAPACKE_sgetrs, LAPACKE_dgetrs, LAPACKE_cgetrs, and LAPACKE_zgetrs, returned as an integer function value; info ≥ 0.

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 75.
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. $AX = B$
2. $A^TX = B$
3. $A^{\text{H}}X = B$ (only for CGETRS and ZGETRS)

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 SGETRF, DGETRF, CGETRF or ZGETRF, respectively. For details on the factorization, see "SGETRF, DGETRF, CGETRF and ZGETRF (General Matrix Factorization)" on page 534.

If $n = 0$ or $nrhs = 0$, no computation is performed and the subroutine returns after doing some parameter checking. See references 8 on page 1363, 44 on page 1366, and 73 on page 1367.

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. \( transa \neq 'N', 'T', \text{ or } 'C' \)
2. \( n < 0 \)
3. \( nrhs < 0 \)
4. \( n > lda \)
5. \( lda \leq 0 \)
6. \( n > ldb \)
7. \( ldb \leq 0 \)

Examples

Example 1
This example shows how to solve the system \( AX = B \), where matrix \( A \) is the same matrix factored in the Example 1 for DGETRF.

Call Statement and Input:

\[
\begin{array}{cccccccccc}
\text{TRANS} & \text{N} & \text{NRHS} & \text{A} & \text{LDA} & \text{IPIV} & \text{BX} & \text{LDB} & \text{INFO} \\
\hline
\end{array}
\]
\[
\text{CALL DGETRS('N', 9, 5, A, 9, IPVT, BX, 9, INFO)}
\]

\[
\begin{array}{c}
\text{IPVT} = (9, 9, 9, 9, 9, 9, 9, 9, 9) \\
\text{A} = \text{(same as output A in Example 1)} \\
\end{array}
\]

\[
\begin{bmatrix}
93.0 & 186.0 & 279.0 & 372.0 & 465.0 \\
84.4 & 168.8 & 253.2 & 337.6 & 422.0 \\
76.6 & 153.2 & 229.8 & 306.4 & 383.0 \\
70.0 & 140.0 & 210.0 & 280.0 & 350.0 \\
\end{bmatrix}
\]

\[
\begin{array}{c}
\text{BX} = \\
\end{array}
\]

\[
\begin{bmatrix}
65.0 & 130.0 & 195.0 & 260.0 & 325.0 \\
62.0 & 124.0 & 186.0 & 248.0 & 310.0 \\
61.4 & 122.8 & 184.2 & 245.6 & 307.0 \\
63.6 & 127.2 & 190.8 & 254.4 & 318.0 \\
69.0 & 138.0 & 207.0 & 276.0 & 345.0 \\
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
1.0 & 2.0 & 3.0 & 4.0 & 5.0 \\
2.0 & 4.0 & 6.0 & 8.0 & 10.0 \\
3.0 & 6.0 & 9.0 & 12.0 & 15.0 \\
4.0 & 8.0 & 12.0 & 16.0 & 20.0 \\
\end{bmatrix}
\]

\[
\begin{array}{c}
\text{BX} = \\
\end{array}
\]

\[
\begin{bmatrix}
5.0 & 10.0 & 15.0 & 20.0 & 25.0 \\
6.0 & 12.0 & 18.0 & 24.0 & 30.0 \\
7.0 & 14.0 & 21.0 & 28.0 & 35.0 \\
8.0 & 16.0 & 24.0 & 32.0 & 40.0 \\
9.0 & 18.0 & 27.0 & 36.0 & 45.0 \\
\end{bmatrix}
\]

\[
\text{INFO} = 0
\]

Example 2
This example shows how to solve the system \( AX = b \), where matrix \( A \) is the same matrix factored in the Example 2 for ZGETRF.

Call Statement and Input:

\[
\begin{array}{cccccccccc}
\text{TRANS} & \text{N} & \text{NRHS} & \text{A} & \text{LDA} & \text{IPIV} & \text{BX} & \text{LDB} & \text{INFO} \\
\hline
\end{array}
\]
\[
\text{CALL ZGETRS('N', 9, 5, A, 9, IPVT, BX, 9, INFO)}
\]

\[
\begin{array}{c}
\text{IPVT} = (9, 9, 9, 9, 9, 9, 9, 9, 9) \\
\text{A} = \text{(same as output A in Example 2)} \\
\end{array}
\]

\[
\begin{bmatrix}
\end{bmatrix}
\]

\[
\begin{array}{c}
\text{INFO} = 0
\end{array}
\]
\[
\begin{bmatrix}
(193.0, -10.6) & (200.0, 21.8) & (207.0, 54.2) & (214.0, 86.6) & (221.0, 119.0) \\
(173.8, -9.4) & (178.8, 20.2) & (183.8, 49.8) & (188.8, 79.4) & (193.8, 109.0) \\
(156.2, -5.4) & (159.2, 22.2) & (162.2, 49.8) & (165.2, 77.4) & (168.2, 105.0) \\
(141.0, 1.4) & (142.0, 27.8) & (143.0, 54.2) & (144.0, 80.6) & (145.0, 107.0) \\
(129.0, 11.0) & (128.0, 37.0) & (127.0, 63.0) & (126.0, 89.0) & (125.0, 115.0) \\
(121.0, 23.4) & (118.0, 49.8) & (115.0, 76.2) & (112.0, 102.6) & (109.0, 129.0) \\
(117.8, 38.6) & (112.8, 66.2) & (107.8, 93.8) & (102.8, 121.4) & (97.8, 149.0) \\
(120.2, 56.6) & (113.2, 86.2) & (106.2, 115.8) & (99.2, 145.4) & (92.2, 175.0) \\
(129.0, 77.4) & (120.0, 109.8) & (111.0, 142.2) & (102.0, 174.6) & (93.0, 207.0)
\end{bmatrix}
\]

\[
\text{BX} = 
\begin{bmatrix}
(1.0, 1.0) & (1.0, 2.0) & (1.0, 3.0) & (1.0, 4.0) & (1.0, 5.0) \\
(2.0, 1.0) & (2.0, 2.0) & (2.0, 3.0) & (2.0, 4.0) & (2.0, 5.0) \\
(3.0, 1.0) & (3.0, 2.0) & (3.0, 3.0) & (3.0, 4.0) & (3.0, 5.0) \\
(4.0, 1.0) & (4.0, 2.0) & (4.0, 3.0) & (4.0, 4.0) & (4.0, 5.0) \\
(5.0, 1.0) & (5.0, 2.0) & (5.0, 3.0) & (5.0, 4.0) & (5.0, 5.0) \\
(6.0, 1.0) & (6.0, 2.0) & (6.0, 3.0) & (6.0, 4.0) & (6.0, 5.0) \\
(7.0, 1.0) & (7.0, 2.0) & (7.0, 3.0) & (7.0, 4.0) & (7.0, 5.0) \\
(8.0, 1.0) & (8.0, 2.0) & (8.0, 3.0) & (8.0, 4.0) & (8.0, 5.0) \\
(9.0, 1.0) & (9.0, 2.0) & (9.0, 3.0) & (9.0, 4.0) & (9.0, 5.0)
\end{bmatrix}
\]

\[
\text{Output:}
\begin{bmatrix}
(1.0, 1.0) & (1.0, 2.0) & (1.0, 3.0) & (1.0, 4.0) & (1.0, 5.0) \\
(2.0, 1.0) & (2.0, 2.0) & (2.0, 3.0) & (2.0, 4.0) & (2.0, 5.0) \\
(3.0, 1.0) & (3.0, 2.0) & (3.0, 3.0) & (3.0, 4.0) & (3.0, 5.0) \\
(4.0, 1.0) & (4.0, 2.0) & (4.0, 3.0) & (4.0, 4.0) & (4.0, 5.0) \\
(5.0, 1.0) & (5.0, 2.0) & (5.0, 3.0) & (5.0, 4.0) & (5.0, 5.0) \\
(6.0, 1.0) & (6.0, 2.0) & (6.0, 3.0) & (6.0, 4.0) & (6.0, 5.0) \\
(7.0, 1.0) & (7.0, 2.0) & (7.0, 3.0) & (7.0, 4.0) & (7.0, 5.0) \\
(8.0, 1.0) & (8.0, 2.0) & (8.0, 3.0) & (8.0, 4.0) & (8.0, 5.0) \\
(9.0, 1.0) & (9.0, 2.0) & (9.0, 3.0) & (9.0, 4.0) & (9.0, 5.0)
\end{bmatrix}
\]

\[
\text{INFO} = 0
\]
SGEF, DGEF, CGEF, and ZGEF (General Matrix Factorization)

Purpose

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 134. Data Types

<table>
<thead>
<tr>
<th>( A )</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGEF</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGEF</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGEF</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGEF</td>
</tr>
</tbody>
</table>

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

\[
\text{CALL SGEF | DGEF | CGEF | ZGEF (a, lda, n, ipvt)}
\]

C and C++

\[
\text{sgef | dgef | cgef | zgef (a, lda, n, ipvt);}
\]

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 134.

\( lda \)

is the leading dimension of the array specified for \( a \).

Specified as: an integer; \( lda > 0 \) and \( lda \geq n \).

\( n \)

is the order of matrix \( A \).

Specified as: an 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 545. Returned as: an \( lda \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 134.

\( ipvt \)

is the integer vector \( ipvt \) of length \( n \), containing the pivot indices. Returned as: a one-dimensional array of (at least) length \( n \), containing integers.
Notes
1. Calling SGEFCD or DGEFCD with \( \text{iopt} = 0 \) is equivalent to calling SGEF or DGEF.
2. On both input and output, matrix \( A \) conforms to LAPACK format.

Function

The matrix \( A \) is factored using Gaussian elimination with partial pivoting (ipvt) to compute the \( LU \) factorization of \( A \), where \((A = PLU)\):

- \( L \) is a unit lower triangular matrix.
- \( U \) is an upper triangular matrix.
- \( P \) is the permutation matrix.

On output, the transformed matrix \( A \) contains \( U \) in the upper triangle and \( L \) in the strict lower triangle where \( ipvt \) contains the pivots representing permutation \( P \), such that \( A = PLU \).

If \( n \) is 0, no computation is performed. See references [44 on page 1366] and [46 on page 1366].

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 68.

Input-Argument Errors
1. \( \text{lda} \leq 0 \)
2. \( n < 0 \)
3. \( n > \text{lda} \)

Examples

Example 1

This example shows a factorization of a real general matrix \( A \) of order 9.

Call Statement and Input:

\[
\begin{array}{c}
A \quad \text{LDA} \quad N \quad \text{IPVT} \\
\end{array}
\]

\[
\begin{array}{c}
\text{CALL SGEF(} \ A \ , \ 9 \ , \ 9 \ , \ \text{IPVT} \ ) \\
\end{array}
\]

\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
4.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 \\
\end{bmatrix}
\]
A =
\[
\begin{bmatrix}
0.0 & 5.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 6.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 7.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 8.0 & 1.0 & 1.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 9.0 & 1.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 10.0 & 11.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 12.0
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
4.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 & 0.0000 & 0.0000 & 0.0000 \\
0.0000 & 5.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 & 0.0000 \\
0.0000 & 0.0000 & 6.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 & 0.0000 \\
0.0000 & 0.0000 & 0.0000 & 7.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 \\
0.0000 & 0.0000 & 0.0000 & 0.0000 & 8.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 \\
0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 9.0000 & 1.0000 & 1.0000 & 1.0000 \\
0.2500 & 0.1500 & 0.1000 & 0.0714 & 0.0536 & -0.0694 & -0.0306 & 0.1806 & 0.3111 \\
0.2500 & 0.1500 & 0.1000 & 0.0714 & -0.0714 & -0.0556 & -0.0194 & 0.9385 & -0.0031
\end{bmatrix}
\]

\[
\text{IPVT} = (3, 4, 5, 6, 7, 8, 9, 8, 9)
\]

**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 )

\[
\begin{align*}
A &= \\
&= \begin{bmatrix}
(1.0, 2.0) & (1.0, 7.0) & (2.0, 4.0) & (3.0, 1.0) \\
(2.0, 0.0) & (1.0, 3.0) & (4.0, 4.0) & (2.0, 3.0) \\
(2.0, 1.0) & (5.0, 0.0) & (3.0, 6.0) & (0.0, 0.0) \\
(8.0, 5.0) & (1.0, 9.0) & (6.0, 6.0) & (8.0, 1.0)
\end{bmatrix}
\end{align*}
\]

Output:

\[
A = \\
\begin{bmatrix}
(8.0000, 5.0000) & (1.0000, 9.0000) & (6.0000, 6.0000) & (8.0000, 1.0000) \\
(0.2022, 0.1236) & (1.9101, 5.0562) & (1.5281, 2.0449) & (1.5056, -0.1910) \\
(0.2360, -0.0225) & (-0.0654, -0.9269) & (-0.3462, 6.2692) & (-1.6346, 1.3269) \\
(0.1798, -0.1124) & (0.2462, 0.1308) & (0.4412, -0.3655) & (0.2900, 2.3864)
\end{bmatrix}
\]

\[
\text{IPVT} = (4, 4, 3, 4)
\]
SGES, DGES, CGES, and ZGES (General Matrix, Its Transpose, or Its Conjugate Transpose Solve)

Purpose

These subroutines solve the system $Ax = b$ for $x$, where $A$ is a general matrix and $x$ and $b$ are vectors. Using the $iopt$ argument, they can also solve the real system $A^T x = b$ or the complex system $A^H x = b$ 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.

Table 135. Data Types

<table>
<thead>
<tr>
<th>$A$, $b$, $x$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGES</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGES</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGES</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGES</td>
</tr>
</tbody>
</table>

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);

On Entry

$a$ 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 135.

$lda$ is the leading dimension of the array specified for $a$.

Specified as: an integer; $lda > 0$ and $lda \equiv n$.

$n$ is the order of matrix $A$.

Specified as: an integer; $0 \leq n \leq lda$.

$ipvt$ is the integer vector $ipvt$ of length $n$, containing the pivot indices produced by a preceding call to SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, or ZGEF, respectively.

Specified as: a one-dimensional array of (at least) length $n$, containing 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 135.

$iopt$ determines the type of computation to be performed, where:
If \( iopt = 0 \), \( A \) is used in the computation.

If \( iopt = 1 \), \( A^T \) is used in SGES and DGES. \( A^H \) is used in CGES and ZGES.

**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: an 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 135 on page 547.

**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 75.

**Function**

The system \( Ax = b \) 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 \( A^T x = b \) or the complex system \( A^H x = b \) 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. For a description of how \( A \) is factored, see “SGEF, DGEF, CGEF, and ZGEF (General Matrix Factorization)” on page 544.

If \( n \) is 0, no computation is performed. See references 44 on page 1366 and 46 on page 1366.

**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. \( iopt \neq 0 \) or \( 1 \)

**Examples**

**Example 1**

Part 1
This part of the example shows how to solve the system \( Ax = b \), where matrix \( A \) is the same matrix factored in the Example 1 for SGEF and DGEF.

Call Statement and Input:

\[
\begin{align*}
\text{CALL SGES(} & \text{ A, 9, 9, IPVT, BX, 0 )} \\
\text{IPVT} & = (3, 4, 5, 6, 7, 8, 9, 8, 9) \\
\text{BX} & = (4.0, 5.0, 9.0, 10.0, 11.0, 12.0, 12.0, 12.0, 33.0) \\
\text{A} & = \text{same as output A in Example 1} \\
\end{align*}
\]

Output:

\[
\begin{align*}
\text{BX} & = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0) \\
\end{align*}
\]

Part 2

This part of the example shows how to solve the system \( A^T x = b \), where matrix \( A \) is the input matrix factored in Example 1 for SGEF and DGEF. Most of the input is the same in Part 2 as in Part 1.

Call Statement and Input:

\[
\begin{align*}
\text{CALL SGES(} & \text{ A, 9, 9, IPVT, BX, 1 )} \\
\text{IPVT} & = (3, 4, 5, 6, 7, 8, 9, 8, 9) \\
\text{BX} & = (6.0, 8.0, 10.0, 12.0, 13.0, 14.0, 15.0, 15.0, 15.0) \\
\text{A} & = \text{same as output A in Example 1} \\
\end{align*}
\]

Output:

\[
\begin{align*}
\text{BX} & = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0) \\
\end{align*}
\]

Example 2

Part 1

This part of the example shows how to solve the system \( Ax = b \), where matrix \( A \) is the same matrix factored in the Example 2 for CGEF and ZGEF.

Call Statement and Input:

\[
\begin{align*}
\text{CALL CGES(} & \text{ A, 4, 4, IPVT, BX, 0 )} \\
\text{IPVT} & = (4, 4, 3, 4) \\
\text{BX} & = ((-10.0, 85.0), (-6.0, 61.0), (10.0, 38.0), \\
&\quad (58.0, 168.0)) \\
\text{A} & = \text{same as output A in Example 1} \\
\end{align*}
\]

Output:

\[
\begin{align*}
\text{BX} & = ((9.0, 0.0), (5.0, 1.0), (1.0, 6.0), (3.0, 4.0)) \\
\end{align*}
\]

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 for CGEF and ZGEF. Most of the input is the same in Part 2 as in Part 1.

Call Statement and Input:

\[
\begin{align*}
\text{CALL CGES(} & \text{ A, 4, 4, IPVT, BX, 1 )} \\
\end{align*}
\]
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)
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)

Purpose

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 SGESM/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, or ZGEF, respectively.

Table 136. Data Types

<table>
<thead>
<tr>
<th>A, B, X</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGESM</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGESM</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGESM</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGESM</td>
</tr>
</tbody>
</table>

Note: The input to these solve subroutines must be the output from the factorization subroutines SGESM/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, and ZGEF, respectively.

Syntax

Fortran

CALL SGESM | DGESM | CGESM | ZGESM (trans, a, lda, n, ipvt, bx, ldb, nrhs)

C and C++

sgesm | dgesm | cgesm | zgesm (trans, a, lda, n, ipvt, bx, 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 SGESM/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 136.
**lda**

is the leading dimension of the array specified for a.

Specified as: an integer; lda > 0 and lda ≥ n.

**n**

is the order of matrix A.

Specified as: an integer; 0 ≤ n ≤ lda.

**ipvt**

is the integer vector ipvt of length n, containing the pivot indices produced by a preceding call to SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, or ZGEF, respectively.

Specified as: a one-dimensional array of (at least) length n, containing integers.

**bx**

is the general 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 136 on page 551.

**ldb**

is the leading dimension of the array specified for b.

Specified as: an integer; ldb > 0 and ldb ≥ n.

**nrhs**

is the number of right-hand sides in the system to be solved.

Specified as: an integer; nrhs ≥ 0.

**On Return**

**bx** is the matrix X, containing the nrhs solutions to the system. The solutions, each of length n, reside in the columns of X.

Specified as: an ldb by (at least) nrhs array, containing numbers of the data type indicated in Table 136 on page 551.

**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 75.

**Function**

One of the following systems of equations is solved for multiple right-hand sides:

1. \( AX = B \)
2. \( A^T X = B \)
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. For a description of how \( A \) is factorized, see “SGEF, DGEF, CGEF, and ZGEF (General Matrix Factorization)” on page 544.

If \( n \) or \( nrhs \) is 0, no computation is performed. See references [44 on page 1366] and [46 on page 1366].

**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. \( \text{trans} \neq 'N', 'T', \) or 'C'
2. \( \text{lda}, \text{ldb} \leq 0 \)
3. \( n < 0 \)
4. \( n > \text{lda}, \text{ldb} \)
5. \( nrhs < 0 \)

**Examples**

**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 factorized in the [Example 1] for SGEF and DGEF.

Call Statement and Input:

```plaintext
TRANS A LDA N IPVT BX LDB NRHS
CALL SGESM( 'N' , A , 9 , 9 , IPVT , BX , 9 , 2 )
```

\[ \text{IPVT} = (3, 4, 5, 6, 7, 8, 9, 8, 9) \]
\[ \text{A} = \text{(same as output A in Example 1)} \]

\[
\begin{bmatrix}
4.0 & 10.0 \\
5.0 & 15.0 \\
9.0 & 24.0 \\
10.0 & 35.0 \\
\end{bmatrix}
\]

\[ \text{BX} = \begin{bmatrix}
11.0 & 48.0 \\
12.0 & 63.0 \\
12.0 & 70.0 \\
12.0 & 78.0 \\
33.0 & 266.0 \\
\end{bmatrix} \]

Output:

\[
\begin{bmatrix}
1.0 & 1.0 \\
1.0 & 2.0 \\
1.0 & 3.0 \\
1.0 & 4.0 \\
\end{bmatrix}
\]

\[ \text{BX} = \begin{bmatrix}
1.0 & 5.0 \\
1.0 & 6.0 \\
\end{bmatrix} \]
Part 2
This part of the example shows how to solve the system $A^T X = B$ for two right-hand sides, where matrix $A$ is the input matrix factored in Example 1 for SGEF and DGEF.

Call Statement and Input:

```
TRANS A LDA N IPVT BX LDB NRHS
| | | | | | |
CALL SGESM('T', A, 9, 9, IPVT, BX, 9, 2)
```

\[
\begin{align*}
\text{IPVT} & = (3, 4, 5, 6, 7, 8, 9, 8, 9) \\
\text{A} & = \text{(same as output A in Example 1)} \\
& = \begin{bmatrix} 6.0 & 15.0 \\ 8.0 & 26.0 \\ 10.0 & 40.0 \\ 12.0 & 57.0 \end{bmatrix} \\
\text{BX} & = \begin{bmatrix} 13.0 & 76.0 \\ 14.0 & 97.0 \\ 15.0 & 120.0 \\ 15.0 & 125.0 \\ 15.0 & 129.0 \end{bmatrix}
\end{align*}
\]

Output:

\[
\begin{align*}
\text{BX} & = \begin{bmatrix} 1.0 & 5.0 \\ 1.0 & 6.0 \\ 1.0 & 7.0 \\ 1.0 & 8.0 \\ 1.0 & 9.0 \end{bmatrix}
\end{align*}
\]

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 Example 2 for CGEF and ZGEF.

Call Statement and Input:

```
TRANS A LDA N IPVT BX LDB NRHS
| | | | | | |
CALL CGESM('N', A, 4, 4, IPVT, BX, 4, 2)
```

\[
\begin{align*}
\text{IPVT} & = (4, 4, 3, 4) \\
\text{A} & = \text{(same as output A in Example 2)} \\
& = \begin{bmatrix} -10.0, 85.0 & -11.0, 53.0 \\ -6.0, 61.0 & -6.0, 54.0 \\ 10.0, 38.0 & 2.0, 40.0 \\ 58.0, 168.0 & 15.0, 105.0 \end{bmatrix} \\
\text{BX} & = \begin{bmatrix} 10.0, 38.0 & 2.0, 40.0 \\ 58.0, 168.0 & 15.0, 105.0 \end{bmatrix}
\end{align*}
\]
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 2 for CGEF and ZGEF.

Call Statement and Input:

```c
CALL CGESM( 'T', A, 4, 4, IPVT, BX, 4, 2 )
```

$IPVT = (4, 4, 3, 4)$

$A = \begin{bmatrix}
    (71.0, 12.0) & (18.0, 68.0) \\
    (61.0, -70.0) & (-27.0, 71.0) \\
    (123.0, -34.0) & (-11.0, 97.0) \\
    (68.0, 7.0) & (28.0, 50.0)
\end{bmatrix}$

Output:

$BX = \begin{bmatrix}
    (9.0, 0.0) & (1.0, 1.0) \\
    (5.0, 1.0) & (2.0, 2.0) \\
    (1.0, 6.0) & (3.0, 3.0) \\
    (3.0, 4.0) & (4.0, 4.0)
\end{bmatrix}$

Part 3:
This part of the example shows how to solve the system $A^HX = B$ for two right-hand sides, where matrix $A$ is the input matrix factored in Example 2 for CGEF and ZGEF.

Call Statement and Input:

```c
CALL CGESM( 'C', A, 4, 4, IPVT, BX, 4, 2 )
```

$IPVT = (4, 4, 3, 4)$

$A = \begin{bmatrix}
    (58.0, -3.0) & (45.0, 20.0) \\
    (68.0, -31.0) & (83.0, -20.0) \\
    (89.0, -22.0) & (98.0, 1.0) \\
    (53.0, 15.0) & (45.0, 25.0)
\end{bmatrix}$

Output:

$BX = \begin{bmatrix}
    (1.0, 4.0) & (4.0, 5.0) \\
    (2.0, 3.0) & (3.0, 4.0) \\
    (3.0, 2.0) & (2.0, 3.0) \\
    (4.0, 1.0) & (1.0, 2.0)
\end{bmatrix}$
SGECON, DGECON, CGECON, and ZGECON (Estimate the Reciprocal of the Condition Number of a General Matrix)

Purpose

SGECON, DGECON, CGECON, and ZGECON estimate the reciprocal of the condition number of general matrix $A$. These subroutines use the results of the factorization of matrix $A$ produced by a preceding call to SGETRF, DGETRF, CGETRF, or ZGETRF, respectively. For details on the factorization, see “SGETRF, DGETRF, CGETRF and ZGETRF (General Matrix Factorization)” on page 534.

<table>
<thead>
<tr>
<th>Table 137. Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$, work</td>
</tr>
<tr>
<td>anorm, rcond, rwork</td>
</tr>
<tr>
<td>Subroutine</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>SGECON</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
<tr>
<td>DGECON</td>
</tr>
<tr>
<td>Short-precision complex</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>CGECON</td>
</tr>
<tr>
<td>Long-precision complex</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
<tr>
<td>ZGECON</td>
</tr>
</tbody>
</table>

Syntax

Fortran

CALL SGECON | DGECON (norm, n, a, lda, anorm, rcond, work, iwork, info)

C and C++

sgecon | dgecon (norm, n, a, lda, anorm, rcond, work, iwork, info);

cgecon | zgecon (norm, n, a, lda, anorm, rcond, work, rwork, info);

LAPACK

info = LAPACKE_sgecon | LAPACKE_dgecon (matrix_layout, norm, n, a, lda, anorm, rcond);

info = LAPACKE_cgecon | LAPACKE_zgecon (matrix_layout, norm, n, a, lda, anorm, rcond);

On Entry

- $matrix_layout$
  - indicates whether the input and output matrices are stored in row major or column major order, where:
    - If $matrix_layout = \text{LAPACK\_ROW\_MAJOR}$, the matrices are stored in row major order.
    - If $matrix_layout = \text{LAPACK\_COL\_MAJOR}$, the matrices are stored in column major order.
  - Specified as: an integer. It must be \text{LAPACK\_ROW\_MAJOR} or \text{LAPACK\_COL\_MAJOR}

- $norm$
  - specifies whether the estimate of the condition number is computed using the one norm or the infinity norm; where:
    - If $norm = \text{’O’}$ or \text{’1’}, the one norm is used in the computation.
    - If $norm = \text{’I’}$, the infinity norm is used in the computation.
  - Specified as: a single character; $norm = \text{’O’}$, \text{’1’}, or \text{’I’}.

- $n$
  - the order of the factored general matrix $A$ used in the computation.
  - Specified as: an integer; $n \geq 0$.

- $a$
  - is the general matrix $A$, containing 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 137 on page 556.

*lda* is the leading dimension of matrix *A*.

Specified as: an integer; *lda* > 0 and *lda* ≥ *n*.

*anorm* has the following meaning:

If *norm* = ‘O’ or ‘1’, then *anorm* is the one norm of the original matrix.

If *norm* = ‘I’, then *anorm* is the infinity norm of the original matrix.

**Note:** You may obtain the value of *anorm* by a preceding call to SLANGE, DLANGE, CLANGE, or ZLANGE, respectively. Refer to “SLANGE, DLANGE, CLANGE, and ZLANGE (General Matrix Norm)” on page 572.

Specified as: a number ≥ 0.0, of the data type indicated in Table 137 on page 556.

*rcond* See On Return.

*work* is the work area used by this subroutine, where:

For SGECON and DGECON

The size of *work* is (at least) of length 4*n*.

For CGECON and ZGECON

The size of *work* is (at least) of length 2*n*.

Specified as: an area of storage containing numbers of data type indicated in Table 137 on page 556.

*iwork* is a work area used by this subroutine whose size is (at least) of length *n*.

Specified as: an area of storage containing integers.

*rwork* is a work area used by this subroutine whose size is (at least) of length 2*n*.

Specified as: an area of storage containing numbers of the data type indicated in Table 137 on page 556.

On Return

*rcond* has the following meaning:

If *info* = 0, an estimate of the reciprocal of the condition number of general matrix *A* is returned; i.e., rcond = 1.0/(NORM(A) × NORM(A⁻¹)).

If *n* = 0, the subroutines return with rcond = 1.0.

If *n* ≠ 0 and anorm = 0.0, the subroutines return with rcond = 0.0.

Returned as: a number ≥ 0.0, of the data type indicated in Table 137 on page 556.
info has the following meaning:

If info = 0, the computation completed normally.

Returned as:

- For SGECON, DGECON, CGECON, and ZGECON returned as: an integer; info ≥ 0.
- For LAPACKE_sgecon, LAPACKE_dgecon, LAPACKE_cgecon, and LAPACKE_zgecon returned as an integer function value; info ≥ 0.

Notes
1. In your C program, arguments rcond and info must be passed by reference.
2. This subroutine accepts lowercase letters for the norm argument.
3. The scalar data specified for input argument n must be the same for SLANGE/DLANGE/CLANGE/ZLANGE, SGETRF/DGETRF/CGETRF/ZGETRF, and SGECON/DGECON/CGECON/ZGECON. In addition, the scalar data specified for input argument m in SLANGE/DLANGE/CLANGE/ZLANGE and SGETRF/DGETRF/CGETRF/ZGETRF must be the same as input argument n in SLANGE/DLANGE/CLANGE/ZLANGE, SGETRF/DGETRF/CGETRF/ZGETRF, and SGECON/DGECON/CGECON/ZGECON.
4. The matrix A input to SLANGE/DLANGE/CLANGE/ZLANGE must be the same as the corresponding input argument for SGETRF/DGETRF/CGETRF/ZGETRF.
5. The matrix A input to SGECON/DGECON/CGECON/ZGECON must be the same as the corresponding output argument for SGETRF/DGETRF/CGETRF/ZGETRF.
6. On both input and output, matrix A conforms to LAPACK format.

Function

The reciprocal of the condition number of general matrix A is estimated, using the results of the factorization of matrix A produced by a preceding call of SGETRF, DGETRF, CGETRF, or ZGETRF.

\[
\text{rcond} = 1.0/(\text{NORM}(A) \times \text{NORM}(A^{-1})).
\]

If n = 0, the subroutines return with rcond = 1.0.

If n ≠ 0 and anorm = 0.0, the subroutines return with rcond = 0.0.

See reference \[82 on page 1368\].

Error conditions

Resource Errors
None.

Computational Errors
None.

Input-Argument Errors
1. norm ≠ 'O', '1', or 'I'
2. n < 0
3. n > lda
4. lda ≤ 0
5. \( \text{anorm} < 0 \)
6. \( \text{anorm} \neq 0 \) and \( \text{anorm} > \text{big} \) or \( \text{anorm} < \text{tiny} \)

Where:

For SGECON and CGECON

\( \text{big} \) and \( \text{tiny} \) have the following values:

\[ \text{big} = 2^{127} \times (1 - \text{ULP}) \]
\[ \text{tiny} = 2^{-126} \times (2^{21}) \]

For DGECON

\( \text{big} \) and \( \text{tiny} \) have the following values:

\[ \text{big} = 2^{1023} \times (1 - \text{ULP}) \]
\[ \text{tiny} = 2^{-1022} \times (2^{49}) \]

For ZGECON

\( \text{big} \) and \( \text{tiny} \) have the following values:

\[ \text{big} = 2^{1023} \times (1 - \text{ULP}) \]
\[ \text{tiny} = 2^{-1022} \times (2^{50}) \]

Where \( \text{ULP} \) = unit in last place.

Note: To avoid this error, scale matrix \( A \) so that \( \text{tiny} \leq \text{anorm} \leq \text{big} \).

Examples

Example 1

This example estimates the reciprocal of the condition number of real general matrix \( A \). The input matrix \( A \) to DLANGE and DGETRF is the same as input matrix \( A \) in Example 3.

Call Statements and Input:

\[
\begin{align*}
\text{NORM} & \quad \text{M} \quad \text{N} \quad \text{A} \quad \text{LDA} \quad \text{WORK} \\
\text{ANORM} &= \text{DLANGE}( '1', 9, 9, A, 9, \text{WORK} ) \\
\text{M} & \quad \text{N} \quad \text{A} \quad \text{LDA} \quad \text{IPVT} \quad \text{INFO} \\
\text{CALL} & \quad \text{DGETRF}( 9, 9, A, 9, \text{IPVT}, \text{INFO} ) \\
\text{NORM} & \quad \text{N} \quad \text{A} \quad \text{LDA} \quad \text{ANORM} \quad \text{RCOND} \quad \text{WORK} \quad \text{IWORK} \quad \text{INFO} \\
\text{CALL} & \quad \text{DGECON}( '1', 9, \text{A}, 9, \text{ANORM}, \text{RCOND}, \text{WORK}, \text{IWORK}, \text{INFO} )
\end{align*}
\]

\( A \) = (same as output \( A \) in Example 3)

ANORM = (same as output ANORM in Example 1)

Output:

\( \text{RCOND} = 5.44 \times 10^{-5} \)

\( \text{INFO} = 0 \)

Example 2

This example estimates the reciprocal of the condition number of complex general matrix \( A \). The input matrix \( A \) to ZLANGE and ZGETRF is the same as input matrix \( A \) in Example 2.

Call Statements and Input:

\[
\begin{align*}
\text{NORM} & \quad \text{M} \quad \text{N} \quad \text{A} \quad \text{LDA} \quad \text{WORK} \\
\text{ANORM} &= \text{ZLANGE}( '1', 4, 4, A, 4, \text{RWORK} )
\end{align*}
\]
CALL ZGETRF( 4, 4, A, 4, IPVT, INFO )

CALL ZGECON( '1', 4, A, 4, ANORM, RCOND, WORK, RWORK, INFO )

A = (same as output A in Example 2)
ANORM = (same as output ANORM in Example 2)

Output:
RCOND = $3.66 \times 10^{-2}$
INFO = 0
SGEFCD and DGEFCD (General Matrix Factorization, Condition Number Reciprocal, and Determinant)

Purpose

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 138. Data Types

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Short-precision real</th>
<th>Long-precision real</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGEFCD</td>
<td></td>
<td>DGEFCD</td>
</tr>
</tbody>
</table>

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

```fortran
CALL SGEFCD | DGEFCD (a, lda, n, ipvt, iopt, rcond, det, aux, naux)
```

C and C++

```c
sgefcd | dgefc (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 138.

- $lda$ is the leading dimension of the array specified for $a$.
  Specified as: an integer; $lda > 0$ and $lda \geq n$.

- $n$ is the order of matrix $A$.
  Specified as: an 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: an integer; $iopt = 0, 1, 2,$ or $3$. 

\texttt{rcond}

See \textit{On Return}

\texttt{det}

See \textit{On Return}

\texttt{aux}

has the following meaning:

If \texttt{naux} = 0 and error 2015 is unrecoverable, \texttt{aux} is ignored.

Otherwise, it is a storage work area used by this subroutine. Its size is specified by \texttt{naux}.

Specified as: an area of storage, containing numbers of the data type indicated in \textit{Table 138 on page 561}

\texttt{naux}

is the size of the work area specified by \texttt{aux}—that is, the number of elements in \texttt{aux}.

Specified as: an integer, where:

If \texttt{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, \texttt{naux} \geq n.

\textbf{On Return}

\texttt{a} is the transformed matrix \textit{A} of order \textit{n}, containing the results of the factorization. See \textit{“Function” on page 563}. Returned as: an \textit{lda} by (at least) \textit{n} array, containing numbers of the data type indicated in \textit{Table 138 on page 561}

\texttt{ipvt}

is the integer vector \textit{ipvt} of length \textit{n}, containing the pivot indices. Returned as: a one-dimensional array of (at least) length \textit{n}, containing integers.

\texttt{rcond}

is an estimate of the reciprocal of the condition number, \textit{rcond}, of matrix \textit{A}. Returned as: a number of the data type indicated in \textit{Table 138 on page 561}.

\texttt{det}

is the vector \textit{det}, containing the two components, \textit{det}_1 and \textit{det}_2, of the determinant of matrix \textit{A}. The determinant is:

\[ \textit{det}_1 \left(10^{\textit{det}_2}\right) \]

where \(1 \leq \textit{det}_1 < 10\). Returned as: an array of length 2, containing numbers of the data type indicated in \textit{Table 138 on page 561}.

\textbf{Notes}

1. In your C program, argument \texttt{rcond} must be passed by reference.

2. When \texttt{iop} = 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 \texttt{naux} dynamically returned to your program. For details, see \textit{“Using Auxiliary Storage in ESSL” on page 51}.
4. On both input and output, matrix $A$ conforms to LAPACK format.

**Function**

Matrix $A$ is factored using Gaussian elimination with partial pivoting ($ipvt$) to compute the $LU$ factorization of $A$, where ($A=PLU$):

$L$ is a unit lower triangular matrix.
$U$ is an upper triangular matrix.
$P$ is the permutation matrix.

On output, the transformed matrix $A$ contains $U$ in the upper triangle and $L$ in the strict lower triangle where $ipvt$ contains the pivots representing permutation $P$, such that $A = PLU$.

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 [81 on page 1368] and [82 on page 1368].

If $n$ is 0, no computation is performed. See reference [44 on page 1366].

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 68.

**Input-Argument Errors**

1. $lda \leq 0$
2. $n < 0$
3. $n > lda$
4. $iopt \neq 0, 1, 2, \text{or} 3$
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.
Examples

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. 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:

$\| A \|_1 = \max(6.0, 8.0, 10.0, 12.0, 13.0, 14.0, 15.0, 15.0, 15.0) = 15.0$

Estimate of $\| A^{-1} \|_1 = 1091.87$

This estimate is equal to the actual $rcond$ of $5.436 \times 10^{-5}$, which is computed by SGEICD and DGEICD. (See Example 3.) On output, the value in $det$, $|A|$, is equal to 336.

Call Statement and Input:

```
CALL DGEFCD( A, 9, 9, IPVT, 3, RCOND, DET, AUX, 9 )
```

$A$ = (same as input $A$ in Example 1)

Output:

$A$ = (same as output $A$ in Example 1)

$IPVT = (3, 4, 5, 6, 7, 8, 9, 8, 9)$

$RCOND = 0.00005436$

$DET = (3.36, 2.00)$
SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD (General Matrix Inverse, Condition Number Reciprocal, and Determinant)

Purpose

These subroutines find the inverse of general matrix $A$.

Subroutines SGEICD and DGEICD also find the reciprocal of the condition number and the determinant of general matrix $A$.

Table 139. Data Types

<table>
<thead>
<tr>
<th>$A$, aux, rcond, det, work</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGETRI$^\dagger$ and SGEICD</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGETRI$^\dagger$ and DGEICD</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGETRI$^\dagger$</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGETRI$^\dagger$</td>
</tr>
</tbody>
</table>

$^\dagger$LAPACK

Note: The input to SGETRI, DGETRI, CGETRI, and ZGETRI must be the output from the factorization subroutines SGETRF, DGETRF, CGETRF, and ZGETRF, respectively.

If you call subroutines SGEICD and DGEICD with $iopt = 4$, the input must be the output from the factorization subroutines SGEF/SGEFCD/SGETRF or DGEF/DGEFCD/DGEFP/DGETRF, respectively.

Syntax

Fortran

<table>
<thead>
<tr>
<th>CALL SGETRI</th>
<th>DGETRI</th>
<th>CGETRI</th>
<th>ZGETRI ($n, a, lda, ipvt, work, lwork, info$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL SGEICD</td>
<td>DGEICD ($a, lda, n, iopt, rcond, det, aux, naux$)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C and C++

<table>
<thead>
<tr>
<th>sgetri</th>
<th>dgetri</th>
<th>cgetri</th>
<th>zgetri ($n, a, lda, ipvt, work, lwork, info$);</th>
</tr>
</thead>
<tbody>
<tr>
<td>sgeicd</td>
<td>dgeicd ($a, lda, n, iopt, rcond, det, aux, naux$);</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LAPACK

$\text{info} = \text{LAPACKE}_\text{sgetri} \mid \text{LAPACKE}_\text{dgetri} \mid \text{LAPACKE}_\text{cgetri} \mid \text{LAPACKE}_\text{zgetri}$

(\text{matrix\_layout}, n, a, lda, ipvt);

On Entry

(matrix\_layout)

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If matrix\_layout = LAPACK\_ROW\_MAJOR, the matrices are stored in row major order.
- If matrix\_layout = LAPACK\_COL\_MAJOR, the matrices are stored in column major order.

Specified as: an integer. It must be LAPACK\_ROW\_MAJOR or LAPACK\_COL\_MAJOR

$a$ has the following meaning, where:

For subroutines SGETRI, DGETRI, CGETRI, and ZGETRI:
It is the transformed matrix $A$ of order $n$, resulting from the factorization performed in a previous call to SGETRF, DGETRF, CGETRF, or ZGETRF, respectively, whose inverse is computed.

**For subroutines SGEICD and DGEICD:**

If $iopt = 0, 1, 2, 3$, 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 139 on page 565.

$l da$

is the leading dimension of the array specified for $a$.

Specified as: an integer; $lda > 0$ and $lda ≤ n$.

$n$

is the order of matrix $A$.

Specified as: an integer; $0 ≤ n ≤ lda$.

$i opt$

indicates the type of computation to be performed, where:

- If $i opt = 0$, the inverse is computed for matrix $A$.
- If $i opt = 1$, the inverse and the reciprocal of the condition number are computed for matrix $A$.
- If $i opt = 2$, the inverse and the determinant are computed for matrix $A$.
- If $i opt = 3$, the inverse, the reciprocal of the condition number, and the determinant are computed for matrix $A$.
- If $i opt = 4$, the inverse is computed using the factored matrix $A$.

Specified as: an integer; $i opt = 0, 1, 2, 3, 4$.

$r con d$

See [On Return](#).

det

See [On Return](#).

$aux$

has the following meaning, and its size is specified by $n aux$:

- If $i opt = 0, 1, 2, 3$, then if $n aux = 0$ and error 2015 is unrecoverable, $aux$ is ignored. Otherwise, it is the storage work area used by this subroutine.

- If $i opt = 4$, $aux$ has the following meaning:
  - For SGEICD, the first $n$ (32-bit integer arguments) or $2n$ (64-bit integer arguments) locations in $aux$ must contain the $ip vt$ integer vector of length $n$, resulting from a previous call to SGEF, SGETRF, or SGEFCD.
  - For DGEICD, the first ceiling($n/2$) (32-bit integer arguments) or $n$ (64-bit integer arguments) locations in $aux$ must contain the $ip vt$ 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 139 on page 565.
\( naux \)

is the size of the work area specified by \( aux \); that is, the number of elements in \( aux \).

Specified as: an integer, where:

If \( iopt \neq 4 \), 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, \( naux \geq 100n \).

\( ipvt \)

is the integer vector \( ipvt \) of length \( n \), containing the pivot indices resulting from a previous call to SGETRF, DGETRF, CGETRF, or ZGETRF.

Specified as: a one-dimensional array of (at least) length \( n \), containing integers, where \( 1 \leq ipvt(i) \leq n \).

\( work \)

has the following meaning:

If \( lwork = 0 \), \( work \) is ignored.

If \( lwork \neq 0 \), \( work \) is the work area used by this subroutine, where:

• If \( lwork \neq -1 \), its size is (at least) of length \( lwork \).

• If \( lwork = -1 \), its size is (at least) of length 1.

Specified as: an area of storage containing numbers of data type indicated in Table 139 on page 565.

\( lwork \)

is the number of elements in array \( WORK \).

Specified as: an integer; where:

• If \( lwork = 0 \), SGETRI/DGETRI/CGETRI/ZGETRI dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program. This option is an extension to the LAPACK standard.

• If \( lwork = -1 \), SGETRI/DGETRI/CGETRI/ZGETRI performs a work area query and returns the optimal size of \( work \) in \( work1 \). No computation is performed and the subroutine returns after error checking is complete.

• Otherwise, it must be:

\[ lwork \geq \max(1, n) \]

• For optimal performance, \( lwork \geq 100n \).

\( info \)

See On Return.

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 139 on page 565.

\( rcond \)

is the reciprocal of the condition number, \( rcond \), of matrix \( A \). Returned as: a real number of the data type indicated in Table 139 on page 565; \( rcond \geq 0 \).

\( det \)

is the vector \( det \), containing the two components \( det_1 \) and \( det_2 \) of the

Chapter 10. Linear Algebraic Equations 567
determinant of matrix $A$. The determinant is:

$$\text{det}_1(10^{\text{det}_2})$$

where $1 \leq \text{det}_1 < 10$. Returned as: an array of length 2, containing numbers of the data type indicated in Table 139 on page 565.

work is the work area used by this subroutine if $lwork \neq 0$, where:

- If $lwork \neq 0$ and $lwork \neq -1$, its size is (at least) of length $lwork$.
- If $lwork = -1$, its size is (at least) of length 1.

Returned as: an area of storage, where:

- If $lwork \geq 1$ or $lwork = -1$, then $work_i$ is set to the optimal $lwork$ value and contains numbers of the data type indicated in Table 139 on page 565. Except for $work_i$, the contents of $work$ are overwritten on return.

info has the following meaning:

- If $info = 0$, the inverse completed successfully.
- If $info > 0$, $info$ is set equal to the first $i$ where $U_{ii}$ is exactly zero. The matrix is singular, and its inverse could not be computed.

Returned as:

- For SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD returned as: an integer; $info \geq 0$.
- For LAPACKE_sgetri, LAPACKE_dgetri, LAPACKE_cgetri, and LAPACKE_zgetri, returned as an integer function value; $info \geq 0$.

Notes

1. In your C program, arguments $rcond$ and $info$ must be passed by reference.
2. The input scalar arguments for SGETRI, DGETRI, CGETRI, and ZGETRI must be set to the same values as the corresponding input arguments in the previous call to SGETRF, DGETRF, CGETRF, and ZGETRF, respectively.
   - If $iopt = 4$, the input scalar arguments for SGEICD and DGEICD must be set to the same values as the corresponding input arguments in the previous call to SGEF/SGEFC or DGEF/DGEFCD/DGEFP, respectively.
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 51.
4. The way _GETRI subroutines handle computational errors differs from LAPACK. Like LAPACK, these subroutines use the $info$ argument to provide information about the computational error, but they also provide an error message.
5. On both input and output, matrix $A$ conforms to LAPACK format.
6. For best performance, specify $lwork = 0$.

Function

These subroutines compute the inverse of general square matrix $A$, where:

- $A^{-1}$ is the inverse of matrix $A$, where $AA^{-1} = A^{-1}A = I$, and $I$ is the identity matrix.
Additionally, the subroutines SGEICD and DGEICD compute the reciprocal of the condition number and the determinant of a general square matrix $A$, using partial pivoting to preserve accuracy, where:

- $1/(\| A \| _1(\| A^{-1} \| _1))$ is the reciprocal of the condition number, where $\| A \| _1$ is the one-norm of matrix $A$.
- $|A|$ is the determinant of matrix $A$, where $|A|$ is expressed as:

$$det(A)\left(10^{det_r}\right)$$

The $iopt$ argument is used to determine the combination of output items produced by SGEICD and DGEICD: the inverse, the reciprocal of the condition number, and the determinant.

If $n$ is 0, no computation is performed. See references [44 on page 1366], [46 on page 1366], and [52 on page 1366].

**Error conditions**

**Resource Errors**

1. Unable to allocate internal work area.
2. If $iopt = 0, 1, 2, or 3$, then error 2015 is unrecoverable, $naux = 0$, and unable to allocate work area.

**Computational Errors**

Matrix $A$ is singular or nearly singular.

For SGETRI, DGETRI, CGETRI, and ZGETRI:

- The index $i$ of the first pivot element having a value equal to zero 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 2149 is set to be unlimited in the ESSL error option table.

For SGEICD and DGEICD:

- 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 68.

**Input-Argument Errors**

1. $lda \leq 0$
2. $n < 0$
3. $n > lda$
4. $iopt \neq 0, 1, 2, 3, or 4$
5. $lwork \neq 0, lwork \neq -1$, and $lwork < \max(1, n)$
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.
Examples

Example 1

This example computes the inverse of matrix \( A \), where matrix \( A \) is the transformed matrix factored by \( \text{SGETRF} \) in Example 3 and the input contents of \( \text{IPVT} \) are the same as the output contents of \( \text{IPVT} \) in Example 3.

**Note:** Because \( \text{lwork} \) is 0, \( \text{ZGETRI} \) dynamically allocates the work area used by this subroutine.

**Call Statement and Input:**

```plaintext
CALL \text{SGETRI}( 9, A, 9, \text{IPVT}, \text{WORK}, 0, \text{INFO} )
```

**Output:**

\[
\begin{bmatrix}
0.333 & -0.667 & 0.333 & 0.000 & 0.000 & 0.000 & 0.042 & -0.042 & 0.000 \\
56.833 & -52.167 & -1.167 & -0.500 & -0.500 & -0.357 & 6.836 & -0.479 & -0.500 \\
-55.167 & 51.833 & 0.833 & 0.500 & 0.500 & 0.214 & -6.735 & 0.521 & 0.500 \\
-1.000 & 1.000 & 0.000 & 0.000 & 0.000 & 0.143 & -0.143 & 0.000 & 0.000 \\
-1.000 & 1.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 \\
-1.000 & 1.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 \\
-226.000 & 206.000 & 5.000 & 3.000 & 2.000 & 1.429 & -0.125 & 0.125 & 0.000 \\
-350.000 & 305.000 & 5.000 & 3.000 & 2.000 & 1.429 & -0.125 & 0.125 & 0.000 \\
\end{bmatrix}
\]

**INFO = 0**

Example 2

This example computes the inverse of matrix \( A \), where \( A \) is the transformed matrix factored by \( \text{ZGETRF} \) in Example 2 and the input contents of \( \text{IPVT} \) are the same as the output contents of \( \text{IPVT} \) in Example 2.

**Note:** Because \( \text{lwork} \) is 0, \( \text{ZGETRI} \) dynamically allocates the work area used by this subroutine.

**Call Statement and Input:**

```plaintext
CALL \text{ZGETRI}( 9, A, 9, \text{IPVT}, \text{WORK}, 0, \text{INFO} )
```

**Output:**

\[
\begin{bmatrix}
(-0.2, -0.4) & (-0.1, 0.1) & (-0.1, 0.1) & (0.0, 0.1) & (0.1, 0.1) & (0.1, 0.1) & (0.1, 0.1) & (0.1, 0.1) & (0.1, 0.1) \\
(0.2, 0.4) & (0.0, -0.6) & (-0.1, 0.0) & (-0.1, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (0.2, 0.4) & (0.0, -0.6) & (-0.1, 0.0) & (-0.1, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (0.0, 0.0) & (0.2, 0.4) & (0.0, -0.6) & (-0.1, 0.0) & (-0.1, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.2, 0.4) & (0.0, -0.6) & (-0.1, 0.0) & (-0.1, 0.0) & (-0.1, 0.0) \\
(0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.2, 0.4) & (0.0, -0.6) & (-0.1, 0.0) & (-0.1, 0.0) \\
(0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.2, 0.4) & (-0.2, -0.4) & (-0.2, -0.4) \\
\end{bmatrix}
\]
INFO = 0

Example 3

This example computes the inverse, the reciprocal of the condition number, and the determinant of matrix \( A \). The values used to compute the reciprocal of the condition number in this example are obtained with the following values:

\[
\| A \|_1 = \max(6.0, 8.0, 10.0, 12.0, 13.0, 14.0, 15.0, 15.0, 15.0) = 15.0 \\
\| A^{-1} \|_1 = 1226.33
\]

On output, the value in \( \det |A| \), is equal to 336.

Call Statement and Input:

\[
\begin{aligned}
&\text{CALL DGEICD( A, 9, 9, 4, RCOND, DET, AUX, 293 )} \\
&\begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
4.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 5.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 6.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 7.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 8.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 9.0 & 1.0 & 1.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 10.0 & 11.0 & 12.0
\end{bmatrix}
\end{aligned}
\]

Output:

\[
\begin{aligned}
&\begin{bmatrix}
0.333 & -0.667 & 0.333 & 0.000 & 0.000 & 0.000 & 0.042 & -0.042 & 0.000 \\
-56.833 & -52.167 & -1.167 & -0.500 & -0.500 & -0.357 & 6.836 & -0.479 & -0.500 \\
-55.167 & 51.833 & 0.833 & 0.500 & 0.500 & 0.214 & -6.735 & 0.521 & 0.500 \\
-1.000 & 1.000 & 0.000 & 0.000 & 0.000 & 0.143 & -0.143 & 0.000 & 0.000 \\
-1.000 & 1.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 \\
-1.000 & 1.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 \\
-226.000 & 206.000 & 5.000 & 3.000 & 2.000 & 1.429 & -27.179 & 1.750 & 2.000 \\
560.000 & -520.000 & -10.000 & -6.000 & -4.000 & -2.857 & 67.857 & -5.000 & -5.000 \\
-325.000 & 305.000 & 5.000 & 3.000 & 2.000 & 1.429 & -39.554 & 3.125 & 3.000
\end{bmatrix}
\end{aligned}
\]

\[
\begin{aligned}
&\text{RCOND = 0.00005436} \\
&\text{DET = (3.36, 2.00)}
\end{aligned}
\]

Example 4

This example computes the inverse of matrix \( A \), where: \( iopt = 4 \); matrix \( A \) is the transformed matrix factored by SGEF in [Example 1] and the input contents of \( AUX \) are the same as the output contents of \( IPVT \) in [Example 1].

Call Statement and Input:

\[
\begin{aligned}
&\text{CALL SGEICO( A, 9, 9, 4, RCOND, DET, AUX, 300 )} \\
&\begin{bmatrix}
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0
\end{bmatrix}
\end{aligned}
\]

Output:
SLANGE, DLANGE, CLANGE, and ZLANGE (General Matrix Norm)

Purpose

SLANGE, DLANGE, CLANGE, and ZLANGE compute the norm of general matrix \( A \).

Table 140. Data Types

<table>
<thead>
<tr>
<th>( A )</th>
<th>( \text{work, Result} )</th>
<th>( \text{Subprogram} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SLANGE&lt;sup&gt;®&lt;/sup&gt;</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DLANGE&lt;sup&gt;®&lt;/sup&gt;</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>CLANGE&lt;sup&gt;®&lt;/sup&gt;</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>ZLANGE&lt;sup&gt;®&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

Syntax

Fortran

\[
\text{SLANGE | DLANGE | CLANGE | ZLANGE (norm, m, n, a, lda, work)}
\]

C and C++

\[
\text{slange | dlange | clange | zlange (norm, m, n, a, lda, work);}
\]

LAPACK

\[
\text{LAPACK\_slange | LAPACK\_dlange | LAPACK\_clange | LAPACK\_zlange (matrix\_layout, norm, m, n, a, lda)};
\]

On Entry

- **\( \text{matrix\_layout} \)**
  - indicates whether the input and output matrices are stored in row major order or column major order, where:
  - If \( \text{matrix\_layout} = \text{LAPACK\_ROW\_MAJOR} \), the matrices are stored in row major order.
  - If \( \text{matrix\_layout} = \text{LAPACK\_COL\_MAJOR} \), the matrices are stored in column major order.

- **\( \text{norm} \)**
  - specifies the type of computation, where:
  - If \( \text{norm} = 'O' \) or '1', the one norm of \( A \) is computed.
  - If \( \text{norm} = 'I' \), the infinity norm of \( A \) is computed.
  - If \( \text{norm} = 'F' \) or 'E', the Frobenius or Euclidean norm of \( A \) is computed.
  - If \( \text{norm} = 'M' \), the absolute value of the matrix element having the largest absolute value, i.e., \( \max (|A|) \), is returned.

\[
\begin{bmatrix}
0.333 & -0.667 & 0.333 & 0.000 & 0.000 & 0.000 & 0.042 & -0.042 & 0.000 \\
56.833 & -52.167 & -1.167 & -0.500 & -0.500 & -0.357 & 6.836 & -0.479 & -0.500 \\
-55.167 & 51.833 & 0.833 & 0.500 & 0.500 & 0.214 & -6.735 & 0.521 & 0.500 \\
-1.000 & 1.000 & 0.000 & 0.000 & 0.000 & 0.143 & -0.143 & 0.000 & 0.000 \\
-1.000 & 1.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 \\
-1.000 & 1.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 \\
-226.000 & 206.000 & 5.000 & 3.000 & 2.000 & 1.429 & -27.179 & 1.750 & 2.000 \\
560.000 & -520.000 & -10.000 & -6.000 & -4.000 & -2.857 & 67.857 & -5.000 & -5.000 \\
-325.000 & 305.000 & 5.000 & 3.000 & 2.000 & 1.429 & -39.554 & 3.125 & 3.000
\end{bmatrix}
\]
Specified as: a single character; \( \text{norm} = \text{'O', '1', 'T', 'F', 'E', or 'M'} \).

\( m \) the number of rows in matrix \( A \).
Specified as: an integer; \( m \geq 0 \).

\( n \) the number of columns in matrix \( A \).
Specified as: an integer; \( n \geq 0 \).

\( a \) is the general matrix \( A \), with \( m \) rows and \( n \) columns.
Specified as: an \( \text{lda} \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 140 on page 572.

\( \text{lda} \)
is the leading dimension of matrix \( A \).
Specified as: an integer; \( \text{lda} > 0 \) and \( \text{lda} \geq m \).

\( \text{work} \)
is the work area used by this subroutine, where:

- When \( \text{norm} = \text{T} \), the size of \( \text{work} \) is (at least) of length \( m \).
- Otherwise, \( \text{work} \) is not referenced.

Specified as: an area of storage containing numbers of data type indicated in Table 140 on page 572.

**On Return**

**Function value**
is the result of the norm computation, returned as a number of the data type indicated in Table 140 on page 572.

If \( \text{norm} = \text{'O'} \) or \( \text{'1'} \), the one norm of \( A \) is returned.

If \( \text{norm} = \text{'T'} \), the infinity norm of \( A \) is returned.

If \( \text{norm} = \text{'F'} \) or \( \text{'E'} \), the Frobenius or Euclidean norm of \( A \) is returned.

If \( \text{norm} = \text{'M'} \), the absolute value of the matrix element having the largest absolute value, i.e., \( \max (|A|) \), is returned.

If \( m = 0 \) or \( n = 0 \), the function returns zero.

**Notes**

1. Declare this function in your program as returning a value of the data type indicated in Table 140 on page 572.

2. This function accepts lowercase letters for the \( \text{norm} \) argument.

**Function**

One of the following computations is performed on general matrix \( A \), depending on the value specified for \( \text{norm} \):

<table>
<thead>
<tr>
<th>Value specified for ( \text{norm} )</th>
<th>Type of computation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>'O' or '1'</td>
<td>one norm</td>
</tr>
<tr>
<td>'T'</td>
<td>infinity norm</td>
</tr>
<tr>
<td>'F' or 'E'</td>
<td>Frobenius or Euclidean norm</td>
</tr>
<tr>
<td>'M'</td>
<td>absolute value of the matrix element having the largest absolute value, i.e., ( \max (</td>
</tr>
</tbody>
</table>
If \( m = 0 \) or \( n = 0 \), the function returns zero.

**Error conditions**

**Resource Errors**
None.

**Computational Errors**
None.

**Input-Argument Errors**
1. \( \text{norm} \neq 'O', '1', 'T', 'F', 'E', \) or 'M'
2. \( m < 0 \)
3. \( n < 0 \)
4. \( m > \text{lda} \)
5. \( \text{lda} \leq 0 \)

**Examples**

**Example 1**

This example computes the one norm of real general matrix \( A \).

Call Statements and Input:

```plaintext
NORM M N A LDA WORK
ANORM = DLANGE( '1', 9, 9, A, 9, WORK )
```

\( A = \) (same as input matrix \( A \) in Example 3)

Output:

ANORM = 15.0

**Example 2**

This example computes the one norm of complex general matrix \( A \).

Call Statements and Input:

```plaintext
NORM M N A LDA WORK
ANORM = ZLANGE( '1', 4, 4, A, 4, WORK )
```

\( A = \) (same as input matrix \( A \) in Example 2)

Output:

ANORM = 25.32
SPPSV, DPPSV, CPPSV, and ZPPSV (Positive Definite Real Symmetric and Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)

Purpose

These subroutines solve the system of linear equations $AX = B$ for $X$, where $X$ and $B$ are general matrices and:

- for SPPSV and DPPSV, $A$ is a positive definite real symmetric matrix.
- for CPPSV and ZPPSV, $A$ is a positive definite complex Hermitian matrix.

The matrix $A$, stored in upper- or lower-packed storage mode, is factored using Cholesky factorization.

Table 141. Data Types

<table>
<thead>
<tr>
<th>$A, B$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SPPSV$^c$</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DPPSV$^c$</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CPPSV$^c$</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZPPSV$^c$</td>
</tr>
<tr>
<td>$^c$ LAPACK</td>
<td></td>
</tr>
</tbody>
</table>

Syntax

For Fortran:

```
CALL SPPSV | DPPSV | CPPSV | ZPPSV (uplo, n, nrhs, ap, bx, ldb, info)
```

For C and C++:

```
sppsv | dppsv | cppsv | zppsv (uplo, n, nrhs, ap, bx, ldb, info);
```

For LAPACK:

```
info = LAPACKE_sppsv | LAPACKE_dppsv | LAPACKE_cppsv | LAPACKE_zppsv
(matrix_layout, uplo, n, nrhs, ap, bx, ldb);
```

On Entry

- **matrix_layout** indicates whether the input and output matrices are stored in row major order or column major order, where:
  - If `matrix_layout = LAPACK_ROW_MAJOR`, the matrices are stored in row major order.
  - If `matrix_layout = LAPACK_COL_MAJOR`, the matrices are stored in column major order.

  Specified as: an integer. It must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR

- **uplo** indicates whether matrix $A$ is stored in upper- or lower-packed storage mode, where:
  - If `uplo = 'U'`, $A$ is stored in upper-packed storage mode.
  - If `uplo = 'L'`, $A$ is stored in lower-packed storage mode.

  Specified as: a single character. It must be 'U' or 'L'.

- **n** is the order $n$ of matrix $A$ and the number of rows of matrix $B$.

  Specified as: an integer; $n \geq 0$. 

Chapter 10. Linear Algebraic Equations 575
nrhs

is the number of right-hand sides; that is, the number of columns of matrix $B$.
Specified as: an integer; $nrhs \geq 0$.

$ap$ is an array, referred to as $A_p$, in which matrix $A$, to be factored, is stored in upper- or 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 141 on page 575.

$bx$ is the general matrix $B$, containing the $nrhs$ right-hand sides of the system. The right-hand sides, each of length $n$, reside in the columns of $B$.
Specified as: an $ldb$ by (at least) $nrhs$ array, containing numbers of the data type indicated in Table 141 on page 575.

$ldb$ is the leading dimension of the array specified for $B$.
Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

$info$

See On Return.

On Return

$ap$ is an array, referred to as $A_p$, in which the transformed matrix $A$ of order $n$, containing the results of the factorization, is stored in upper- or 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 141 on page 575. See “Function” on page 577.

$bx$ is the general matrix $X$, containing the $nrhs$ solutions to the system. The solutions, each of length $n$, reside in the columns of $X$.
Returned as: an $ldb$ by (at least) $nrhs$ array, containing numbers of the data type indicated in Table 141 on page 575.

$info$

has the following meaning:
If $info = 0$, the subroutine completed successfully.
If $info > 0$, the factorization was unsuccessful. $B$ is overwritten; that is, the original input is not preserved. $info$ is set equal to the order $i$ of the first minor encountered having a nonpositive determinant.
Returned as:
- For SPPSV, DPPSV, CPPSV, and ZPPSV, returned as: an integer; $info \geq 0$.
- For LAPACKE_sppsv, LAPACKE_dppsv, LAPACKE_cppsv, and LAPACKE_zppsv, returned as an integer function value; $info \geq 0$.

Notes

1. These subroutines accept lowercase letters for the $uplo$ argument.
2. In your C program, argument $info$ must be passed by reference.
3. The matrices used in this computation must have no common elements; otherwise, results are unpredictable. See “Concepts” on page 75.
4. 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.
5. For a description of the storage modes used for the matrices, see:
   - For positive definite real symmetric matrices, see "Positive Definite or Negative Definite Symmetric Matrix" on page 89.
   - For positive definite complex Hermitian matrices, see "Positive Definite or Negative Definite Complex Hermitian Matrix" on page 91.

6. On both input and output, matrices $A$, $B$, and $X$ conform to LAPACK format.

7. The way these subroutines handle computational errors differs from LAPACK. Like LAPACK, these subroutines use the info argument to provide information about the computational error, but they also provide an error message.

**Function**

The system $AX = B$ is solved for $X$, where $X$ and $B$ are general matrices and:
- for SPPSV and DPPSV, $A$ is a positive definite real symmetric matrix.
- for CPPSV and ZPPSV, $A$ is a positive definite complex Hermitian matrix.

The matrix $A$, stored in upper- or lower-packed storage mode, is factored using the Cholesky factorization method, where $A$ is expressed as:

- $A = LL^T$ or $A = U^U$
  - for SPPSV and DPPSV
- $A = LL^H$ or $A = U^H$
  - for CPPSV and ZPPSV

where:

- $L$ is a lower triangular matrix.
- $U$ is an upper triangular matrix.

If $n$ is 0, no computation is performed and the subroutine returns after doing some parameter checking. If $n > 0$ and $nrhs$ is 0, no solutions are computed and the subroutine returns after factoring the matrix.

See references [8 on page 1363], [44 on page 1366], and [46 on page 1366].

**Error conditions**

**Resource Errors**

Unable to allocate internal work area.

**Computational Errors**

Matrix $A$ is not positive definite.
- The order $i$ of the first minor encountered having a nonpositive determinant 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 2148 is set to be unlimited in the ESSL error option table.

**Input-Argument Errors**

1. $uplo \neq \text{'U'} \text{ or 'L'}$
2. $n < 0$
3. $nrhs < 0$
4. $n > \text{ldb}$
5. $\text{ldb} \leq 0$
Examples

Example 1

This example shows how to solve the system $AX = B$, where matrix $A$ is a positive definite real symmetric matrix of order 9, stored in lower-packed storage mode.

On input, matrix $A$ is:

$$
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 \\
1.0 & 2.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 4.0 & 4.0 & 4.0 & 4.0 & 4.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 5.0 & 5.0 & 5.0 & 5.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 6.0 & 6.0 & 6.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 7.0 & 7.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 8.0 & 8.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 8.0 & 9.0 \\
\end{bmatrix}
$$

On output, all elements of this matrix $A$ are 1.0.

Note: The $AP$ array is formatted in a triangular arrangement for readability; however, it is stored in lower-packed storage mode.

Call Statement and Input:

```fortran
CALL SPPSV ( 'L', 9, 2, AP, BX, 9, INFO )
```

$AP = \text{(same as input } AP \text{ in Example 5)}$

$BX = \text{(same as input } BX \text{ in Example 5)}$

Output:

$AP = \begin{bmatrix}
1.0 & 1.0 \\
1.0 & 2.0 \\
1.0 & 3.0 \\
1.0 & 4.0 \\
1.0 & 5.0 \\
1.0 & 6.0 \\
1.0 & 7.0 \\
1.0 & 8.0 \\
1.0 & 9.0 \\
\end{bmatrix}$

$BX = \begin{bmatrix}
1.0 \\
1.0 \\
1.0 \\
1.0 \\
1.0 \\
1.0 \\
1.0 \\
1.0 \\
1.0 \\
\end{bmatrix}$

$INFO = 0$

Example 2

This example shows how to solve the system $AX = B$, where matrix $A$ is a positive definite real symmetric matrix of order 9, stored in upper-packed storage mode.

On input, matrix $A$ is:
On output, all elements of this matrix $A$ are 1.0.

Note: The $AP$ array is formatted in a triangular arrangement for readability; however, it is stored in upper-packed storage mode.

Call Statement and Input:

```plaintext
CALL SPPSV ( 'U', 9, 2, AP, BX, 9, INFO )
```

$AP = $ (same as input AP in Example 6)

$BX = $ (same as input BX in Example 6)

Output:

$AP = (\begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 \\
1.0 & 2.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 4.0 & 4.0 & 4.0 & 4.0 & 4.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 5.0 & 5.0 & 5.0 & 5.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 6.0 & 6.0 & 6.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 7.0 & 7.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 8.0 & 8.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 8.0 & 9.0 \\
\end{bmatrix})$

$BX = \begin{bmatrix}
1.0 \\
1.0 \\
1.0 \\
1.0 \\
1.0 \\
1.0 \\
1.0 \\
1.0 \\
1.0 \\
\end{bmatrix}$

$INFO = 0$

Example 3

This example shows how to solve the system $AX = B$, where matrix $A$ is a positive definite complex Hermitian matrix of order 3, stored in lower-packed storage mode.

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:

```
CALL ZPPSV ( 'L', 3, 2, AP, BX, 3, INFO )
```

\[
\begin{align*}
AP & = (\text{same as input } AP \text{ in Example 7}) \\
BX & = (\text{same as input } BX \text{ in Example 7})
\end{align*}
\]

Output:

\[
\begin{align*}
AP & = ((5.0, 0.0), (-1.0, 1.0), (2.0, -1.0), (7.0, 0.0), (1.0, 1.0), (8.0, 0.0)) \\
BX & = \\
& \begin{bmatrix}
(2.0, -1.0) & (2.0, 0.0) \\
(1.0, 1.0) & (-1.0, 2.0) \\
(0.0, -2.0) & (1.0, 1.0)
\end{bmatrix}
\end{align*}
\]

\[
INFO = 0
\]

**Example 4**

This example shows how to solve the system \( AX = B \), where matrix \( A \) is a positive definite complex Hermitian matrix of order 3, stored in upper-packed storage mode.

On input, matrix \( A \) is:

\[
\begin{bmatrix}
(9.0, 0.0) & (3.0, 3.0) & (3.0, -3.0) \\
(3.0, -3.0) & (18.0, 0.0) & (8.0, -6.0) \\
(3.0, 3.0) & (8.0, 6.0) & (43.0, 0.0)
\end{bmatrix}
\]

**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:

```
CALL ZPPSV ( 'U', 3, 2, AP, BX, 3, INFO )
```

\[
\begin{align*}
AP & = (\text{same as input } AP \text{ in Example 8}) \\
BX & = (\text{same as input } BX \text{ in Example 8})
\end{align*}
\]

Output:

\[
\begin{align*}
AP & = ((3.0, 0.0), (1.0, 1.0), (4.0, 0.0), (1.0, -1.0), (2.0, -1.0), (6.0, 0.0)) \\
BX & = \\
& \begin{bmatrix}
(2.0, -1.0) & (2.0, 0.0) \\
(1.0, -1.0) & (0.0, 1.0) \\
(3.0, 0.0) & (1.0, -1.0)
\end{bmatrix}
\end{align*}
\]

\[
INFO = 0
\]
SPOSV, DPOSV, CPOSV, and ZPOSV (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)

Purpose

These subroutines solve the system of linear equations $AX = B$ for $X$, where $X$ and $B$ are general matrices and:
- for SPOSV and DPOSV, $A$ is a positive definite real symmetric matrix.
- for CPOSV and ZPOSV, $A$ is a positive definite complex Hermitian matrix.

The matrix $A$, stored in upper- or lower-storage mode, is factored using Cholesky factorization.

Table 142. Data Types

<table>
<thead>
<tr>
<th>$A, B$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SPOSV</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DPOSV</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CPOSV</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZPOSV</td>
</tr>
<tr>
<td>LAPACK</td>
<td></td>
</tr>
</tbody>
</table>

Syntax

Fortran

CALL SPOSV | DPOSV | CPOSV | ZPOSV (uplo, n, nrhs, a, lda, bx, ldb, info)

C and C++

spsov | dposv | cposv | zposv (uplo, n, nrhs, a, lda, bx, ldb, info);

LAPACK

info = LAPACKE_sposv | LAPACKE_dposv | LAPACKE_cposv | LAPACKE_zposv
(matrix_layout, uplo, n, nrhs, a, lda, bx, ldb);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:
- If matrix_layout = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If matrix_layout = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. It must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR

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'.

$n$

is the order $n$ of matrix $A$ and the number of rows of matrix $B$.

Specified as: an integer; $n \geq 0$. 
nrhs
is the number of right-hand sides; that is, the number of columns of matrix B.
Specified as: an integer; nrhs ≥ 0.
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 142 on page 581] See "Notes “
lda
is the leading dimension of the array specified for A.
Specified as: an integer; lda > 0 and lda ≥ n.

bx
is the general matrix B, containing the nrhs right-hand sides of the system. The right-hand sides, each of length n, reside in the columns of B.
Specified as: an ldb by (at least) nrhs array, containing numbers of the data type indicated in [Table 142 on page 581]

ldb
is the leading dimension of the array specified for B.
Specified as: an integer; ldb > 0 and ldb ≥ n.

info
See [On Return]

On Return

a
is the transformed matrix A of order n, containing the results of the factorization.
Returned as: an lda by (at least) n array, containing numbers of the data type indicated in [Table 142 on page 581] See "Function” on page 583.

bx
is the general matrix X, containing the nrhs solutions to the system. The solutions, each of length n, reside in the columns of X.
Returned as: an ldb by (at least) nrhs array, containing numbers of the data type indicated in [Table 142 on page 581]

info
has the following meaning:
If info = 0, the subroutine completed successfully.
If info > 0, the factorization was unsuccessful and the solution was not computed. info is set equal to the order i of the first minor encountered having a nonpositive determinant.

Returned as:
• For SPOSV, DPOSV, CPOSV, and ZPOSV, returned as: an integer; info ≥ 0.
• For LAPACKE_sposv, LAPACKE_dposv, LAPACKE_cposv, and
  LAPACKE_zposv, returned as an integer function value; info ≥ 0.

Notes
1. In your C program, argument info must be passed by reference.
2. All subroutines accept lowercase letters for the uplo argument.
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. The way these subroutines handle computational errors differs from LAPACK. Like LAPACK, these subroutines use the info argument to provide information about the computational error, but they also provide an error message.

5. On both input and output, matrices $A$, $B$, and $X$ conform to LAPACK format.

6. For a description of the storage modes used for the matrices, see:
   - For positive definite real symmetric matrices, see “Positive Definite or Negative Definite Symmetric Matrix” on page 89.
   - For positive definite complex Hermitian matrices, see “Positive Definite or Negative Definite Complex Hermitian Matrix” on page 91.

7. The matrices used in this computation must have no common elements; otherwise, results are unpredictable. See “Concepts” on page 75.

**Function**

These subroutines solve the system of linear equations $AX = B$ for $X$, where $X$ and $B$ are general matrices and:
- for SPOSV and DPOSV, $A$ is a positive definite real symmetric matrix.
- for CPOSV and ZPOSV, $A$ is a positive definite complex Hermitian matrix.

The matrix $A$ is factored using Cholesky factorization, where $A$ is expressed as:

对于 SPOSV 和 DPOSV

$$A = LL^T \text{ or } A = U^T U$$

对于 CPOSV 和 ZPOSV

$$A = LL^H \text{ or } A = U^H U$$

where:

$L$ is a unit lower triangular matrix.
$U$ is an upper triangular matrix.

If $n$ is 0, no computation is performed and the subroutine returns after doing some parameter checking. If $n > 0$ and $nrhs$ is 0, no solutions are computed and the subroutine returns after factoring the matrix.

See references 8 on page 1363, 44 on page 1366, and 82 on page 1368.

**Error conditions**

**Resource Errors**

Unable to allocate internal work area.

**Computational Errors**

Matrix $A$ is not positive definite.

The order $i$ of the first minor encountered having a nonpositive determinant 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 2148 is set to be unlimited in the ESSL error option table.

**Input-Argument Errors**

1. $\text{uplo} \neq 'U' \text{ or } 'L'$
2. $n < 0$
3. $nrhs < 0$
4. $n > lda$
5. $lda \leq 0$
6. $n > ldb$
7. $ldb \leq 0$

**Examples**

**Example 1**

This example shows how to solve the system $AX = B$, where:

Matrix $A$ is the same used as input in Example 1 for SPOTRF.
Matrix $B$ is the same used as input in Example 1 for SPOTRS.

Call Statement and Input:

```
CALL SPOSV( 'L', 9, 2, A, 9, BX, 9, INFO)
```

$A = (\text{same as input } A \text{ in Example 1})$

$BX = (\text{same as input } BX \text{ in Example 1})$

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 1.0 1.0 1.0  .  .  
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0

BX =
 1.0 5.0
1.0 6.0
1.0 7.0
1.0 8.0
1.0 9.0
```

INFO = 0

**Example 2**

This example shows how to solve the system $AX = B$, where:

Matrix $A$ is the same used as input in Example 2 for SPOTRF.
Matrix $B$ is the same used as input in Example 2 for SPOTRS.

Call Statement and Input:

```
CALL SPOSV( 'U', 9, 2, A, 9, BX, 9, INFO)
```

$A = (\text{same as input } A \text{ in Example 2})$

$BX = (\text{same as input } BX \text{ in Example 2})$

Output:
Example 3

This example shows how to solve the system $AX = B$, where:

Matrix $A$ is the same used as input in Example 3 for CPOTRF.
Matrix $BX$ is the same used as input in Example 3 for CPOTRS.

Call Statement and Input:

```
CALL CPOSV('L', 3, 2, A, 3, BX, 3, INFO)
```

Output:

```
A = 
[ ( 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) ]
BX = 
[ ( 60.0,-55.0) ( 70.0, 10.0) ]  
[ ( 34.0, 58.0) (-51.0, 110.0) ]  
[ (13.0,-152.0) ( 75.0, 63.0) ]
```

INFO = 0

Example 4

This example shows how to solve the system $AX = B$, where:
Matrix $A$ is the same used as input in Example 4 for CPOTRF.
Matrix $BX$ is the same used as input in Example 4 for CPOTRS.

Call Statement and Input:

$$\begin{align*}
\text{UPLO} & \quad \text{N} \quad \text{NRHS} \quad \text{A} \quad \text{LDA} \quad \text{BX} \quad \text{LDB} \quad \text{INFO} \\
\text{CALL} & \quad \text{CPOSV}('U', 3, 2, \ A, 3, \ BX, 3, \ \text{INFO})
\end{align*}$$

\[
A = \begin{bmatrix}
(9.0, 0.0) & (3.0, 3.0) & (3.0, -3.0) \\
(3.0, -3.0) & (18.0, 0.0) & (8.0, -6.0) \\
(3.0, 3.0) & (8.0, 6.0) & (43.0, 0.0)
\end{bmatrix}
\]

\[
BX = \begin{bmatrix}
(33.0, -18.0) & (15.0, -3.0) \\
(45.0, -45.0) & (8.0, -2.0) \\
(152.0, 1.0) & (43.0, -29.0)
\end{bmatrix}
\]

Output:

**Note:** The strict lower part of $A$ is not referenced.

\[
A = \begin{bmatrix}
(3.0, 0.0) & (1.0, 1.0) & (1.0, -1.0) \\
(3.0, -3.0) & (4.0, 0.0) & (2.0, -1.0) \\
(3.0, 3.0) & (8.0, 6.0) & (6.0, 0.0)
\end{bmatrix}
\]

\[
BX = \begin{bmatrix}
(2.0, -1.0) & (2.0, 0.0) \\
(1.0, -1.0) & (0.0, 1.0) \\
(3.0, 0.0) & (1.0, -1.0)
\end{bmatrix}
\]

INFO = 0
SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPP, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)

Purpose

These subroutines factor matrix $A$ as explained below:

**SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, and ZPOF**

The SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, and ZPOF subroutines factor matrix $A$ stored in upper or lower storage mode, where:

- For SPOTRF, DPOTRF, SPOF, and DPOF, $A$ is a positive definite real symmetric matrix.
- For CPOTRF, ZPOTRF, CPOF, and ZPOF, $A$ is a positive definite complex Hermitian matrix.

Matrix $A$ is factored using Cholesky factorization.

To solve the system of equations with one or more right-hand sides, follow the call to SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, or DPOF with a call to SPOTRS, DPOTRS, CPOTRS, ZPOTRS, SPOSM, DPOSM, CPOSM, or ZPOSM, respectively.

To find the inverse of matrix $A$, follow the call to SPOTRF, DPOTRF, CPOTRF, or ZPOTRF with a call to SPOTRI, DPOTRI, CPOTRI, ZPOTRI, SPOICD, or DPOICD, respectively.

To estimate the reciprocal of the condition number of matrix $A$, follow the call to SPOTRF, DPOTRF, CPOTRF, or ZPOTRF with a call to SPOCON, DPOCON, CPOCON, or ZPOCON, respectively.

**SPPTRF, DPPTRF, CPPTRF, and ZPPTRF**

The SPPTRF, DPPTRF, CPPTRF, and ZPPTRF subroutines factor matrix $A$, stored in upper- or lower-packed storage mode, where:

- For SPPTRF and DPPTRF, $A$ is a positive definite real symmetric matrix.
- For CPPTRF and ZPPTRF, $A$ is a positive definite complex Hermitian matrix.

Matrix $A$ is factored using Cholesky factorization.

To solve the system of equations with one or more right-hand sides, follow the call to SPPTRF, DPPTRF, CPPTRF, or ZPPTRF with a call to SPPTRS, DPPTRS, CPPTRS, or ZPPTRS, respectively.

To find the inverse of matrix $A$, follow the call to SPPTRF, DPPTRF, CPPTRF, or ZPPTRF with a call to SPPTRI, DPPTRI, CPPTRI, or ZPPTRI, respectively.

To estimate the reciprocal of the condition number of matrix $A$, follow the call to SPPTRF, DPPTRF, CPPTRF, or ZPPTRF with a call to SPPCON, DPPCON, CPPCON, or ZPPCON, respectively.

**SPPF and DPPF**

The SPPF and DPPF subroutines factor positive definite real symmetric matrix $A$, stored in lower-packed storage mode, using Gaussian elimination ($LDL^T$) or Cholesky factorization. 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.
Table 143. Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SPOTRF, SPOF, SPPTRF, and SPPF</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DPOTRF, DPOF, DPPTRF, and DPPF</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CPOTRF, CPOF, and CPPTRF</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZPOTRF, ZPOF, and ZPPTRF</td>
</tr>
<tr>
<td>LAPACK</td>
<td></td>
</tr>
</tbody>
</table>

Note: The output from each of these subroutines should be used only as input for specific other subroutines, as shown in the table below.

<table>
<thead>
<tr>
<th>Output from this subroutine:</th>
<th>Should be used only as input to the following subroutines:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solve</td>
<td>Inverse</td>
</tr>
<tr>
<td>SPOTRF</td>
<td>SPOTRS</td>
</tr>
<tr>
<td>DPOTRF</td>
<td>DPOTRS</td>
</tr>
<tr>
<td>CPOTRF</td>
<td>CPOTRS</td>
</tr>
<tr>
<td>ZPOTRF</td>
<td>ZPOTRS</td>
</tr>
<tr>
<td>SPOF</td>
<td>SPOSM</td>
</tr>
<tr>
<td>DPOF</td>
<td>DPOSM</td>
</tr>
<tr>
<td>CPOF</td>
<td>CPOSM</td>
</tr>
<tr>
<td>ZPOF</td>
<td>ZPOSM</td>
</tr>
<tr>
<td>SPPTRF</td>
<td>SPPTRS</td>
</tr>
<tr>
<td>DPPTRF</td>
<td>DPPTRS</td>
</tr>
<tr>
<td>CPPTRF</td>
<td>CPPTRS</td>
</tr>
<tr>
<td>ZPPTRF</td>
<td>ZPPTRS</td>
</tr>
<tr>
<td>SPPF</td>
<td>SPS</td>
</tr>
<tr>
<td>DPPF</td>
<td>DPPS</td>
</tr>
</tbody>
</table>

Syntax

Fortran

CALL SPOTRF | DPOTRF | CPOTRF | ZPOTRF (uplo, n, a, lda, info)
CALL SPOF | DPOF | CPOF | ZPOF (uplo, a, lda, n)
CALL SPPTRF | DPPTRF | CPPTRF | ZPPTRF (uplo, n, ap, info)
CALL SPPF | DPPF (ap, n, iopt)

C and C++

spotrf | dpotrf | cpotrf | zpotrf (uplo, n, a, lda, info);
spof | dpoof | cpoof | zpoof (uplo, a, lda, n);
spptrf | dpptrf | cpptrf | zppptrf (uplo, n, ap, info);
sppf | dppf (ap, n, iopt);

LAPACK

info = LAPACKE_spotrf | LAPACKE_dpotrf | LAPACKE_cpotrf | LAPACKE_zpotrf
info = LAPACKE_spptrf | LAPACKE_dpptrf | LAPACKE_cpptrf | LAPACKE_zpptrf

588  ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
On Entry

matrix_layout
indicates whether the input and output matrices are stored in row major order or column major order, where:
- If matrix_layout = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If matrix_layout = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. It must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR

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 an array, referred to as AP, in which matrix A, to be factored, is stored as follows:
- **SPPTRF, DPPTRF, CPPTRF, and ZPPTRF**: Upper-packed or lower-packed storage mode
- **SPPF and DPPF**: Lower-packed storage mode

Specified as: a one-dimensional array, containing numbers of the data type indicated in Table 143 on page 588. See "Notes" on page 591.

**For SPPTRF, DPPTRF, CPPTRF, and ZPPTRF:**
The array must have at least \( n(n+1)/2 \) elements.

**For SPPF and DPPF:**
- If iopt = 0 or 10, the array must have at least \( n(n+1)/2+n \) elements.
- If iopt = 1 or 11, 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 143 on page 588

lda
is the leading dimension of the array specified for a.

Specified as: an integer; lda > 0 and lda ≥ n.

n is the order n of matrix A.

Specified as: an integer; n ≥ 0.

iopt determines the type of computation to be performed, where:
- If iopt = 0, the matrix is factored using the \( LDL^T \) method, and the output is stored in an internal format.
- If iopt = 1, the matrix is factored using Cholesky factorization, and the output is stored in an internal format.
If $iopt = 10$, the matrix is factored using the $LDL^T$ method, and the output is stored in lower-packed storage mode.

If $iopt = 11$, the matrix is factored using Cholesky factorization, and the output is stored in lower-packed storage mode.

Specified as: an integer; $iopt = 0, 1, 10, \text{ or } 11$.

$\text{info}$

See On Return

On Return

$ap$ is an array, referred to as $AP$, in which the transformed matrix $A$ of order $n$, containing the results of the factorization, is stored.

For $\text{SPPTRF, DPTRF, CPPTRF, and ZPPTRF}$:

The transformed matrix is stored in upper-packed or lower-packed storage mode.

For $\text{SPPF and DPPF}$:

If $iopt$ is 0 or 1, the transformed matrix is stored in an internal format and should only be used as input to the corresponding solve or inverse subroutine.

If $iopt$ is 10 or 11, the transformed matrix is stored in lower-packed storage mode.

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 143 on page 588

For $\text{SPPTRF, DPTRF, CPPTRF, and ZPPTRF}$:

The array contains at least $n(n+1)/2$ elements.

For $\text{SPPF and DPPF}$:

If $iopt = 0$ or 10, the array contains $n(n+1)/2+n$ elements.

If $iopt = 1$ or 11, the array contains $n(n+1)/2$ elements.

See “Notes” on page 591 and see “Function” on page 591.

$a$ is the transformed matrix $A$ of order $n$, containing the results of the factorization. See “Function” on page 591.

Returned as: a two-dimensional array, containing numbers of the data type indicated in Table 143 on page 588

$\text{info}$

has the following meaning:

If $info = 0$, the factorization completed successfully.

If $info > 0$, $info$ is set equal to the order $i$ of the first minor encountered having a nonpositive determinant.

Returned as:

- For $\text{SPPTRF, DPTRF, CPPTRF, ZPPTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPPF, and DPPF}$, returned as: an integer; $info \geq 0$.

- For $\text{LAPACKE_spptrf, LAPACKE_dpptrf, LAPACKE_cpptrf, LAPACKE_zpopt, LAPACKE_sppptrf, LAPACKE_dpptrf, LAPACKE_cpptrf, and LAPACKE_zppptrf}$, returned as an integer function value; $info \geq 0$.

Specified as: an integer; $info \geq 0$.
Notes
1. In your C program, argument info must be passed by reference.
2. All subroutines accept lowercase letters for the uplo argument.
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. 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 or 10, are used for working storage by this subroutine and should not be altered between calls to the factorization and solve subroutines.
5. If iopt = 0 or 1, SPPF and DPPF in some cases utilize algorithms based on recursive packed storage format. As a result, on output, if iopt = 0 or 1, the array specified for AP may be stored in this new format rather than the conventional lower packed format. (See references [61 on page 1367], [77 on page 1368], and [79 on page 1368]). The array specified for AP should not be altered between calls to the factorization and solve subroutines; otherwise unpredictable results may occur.
6. The way _POTRF and _PPTRF subroutines handle computational errors differs from LAPACK. Like LAPACK, these subroutines use the info argument to provide information about the computational error, but they also provide an error message.
7. On both input and output, matrix A conforms to LAPACK format.
8. 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 89.
   - For positive definite complex Hermitian matrices, see “Positive Definite or Negative Definite Complex Hermitian Matrix” on page 91.

Function

The functions for these subroutines are described.

For SPOTRF, DPOTRF, CPOTRF, ZPOTRF, 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:

\[ A = LL^T \text{ or } A = U^T U \]
for SPOTRF, DPOTRF, SPOF, and DPOF

\[ A = LL^H \text{ or } A = U^H U \]
for CPOTRF, ZPOTRF, CPOF, and ZPOF

where:

L is a lower triangular matrix.
U is an upper triangular matrix.

If n is 0, no computation is performed. See references [8 on page 1363] and [44 on page 1366].

For SPPTRF, DPPTRF, CPPTRF, and ZPPTRF:
The positive definite matrix $A$, stored in upper-packed or lower-packed storage mode, is factored using Cholesky factorization, where $A$ is expressed as:

$$ A = LL^T \text{ or } A = U^U $$

for SPPTRF and DPPTRF

$$ A = LL^H \text{ or } A = U^U $$

for CPPTRF and ZPPTRF

where:

$L$ is a lower triangular matrix.
$U$ is an upper triangular matrix.

If $n$ is 0, no computation is performed. See references 8 on page 1363, 44 on page 1366, and 78 on page 1368.

For SPPF and DPPF:

If $iopt = 0$ or 10, the positive definite symmetric matrix $A$, stored in lower-packed storage mode, is factored using Gaussian elimination, where $A$ is expressed as:

$$ A = LDL^T $$

where:

$L$ is a unit lower triangular matrix.
$D$ is a diagonal matrix.

If $iopt = 1$ or 11, the positive definite symmetric matrix $A$, stored in lower-packed storage mode, is factored using Cholesky factorization, where $A$ is expressed as:

$$ A = LL^T $$

where $L$ is a lower triangular matrix.

If $n$ is 0, no computation is performed. See references 8 on page 1363 and 44 on page 1366.

**Error conditions**

**Resource Errors**

Unable to allocate internal work area.

**Computational Errors**

1. Matrix $A$ is not positive definite (for SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPPTRF, DPPTRF, CPPTRF, and ZPPTRF).
   - The order $i$ of the first minor encountered having a nonpositive determinant 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 2148 is set to be unlimited in the ESSL error option table.

2. Matrix $A$ is not positive definite (for SPPF and DPPF when $iopt = 0$ or 10).
   - 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.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
• The return code is set to 1.
Example 2

This example shows a factorization of the same positive definite symmetric matrix $A$ of order 9 used in Example 9, but stored in upper storage mode.

Call Statement and Input:

```
CALL SPOTRF('U', 9, A, 9, INFO)
```

```
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 1.0 1.0 1.0 1.0 1.0 1.0 1.0 . . . . . .
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 . . . . . .
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0]
```

```
INFO = 0
```

Example 3

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:

```
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
. . . . . 1.0 1.0 1.0 1.0
. . . . . . 1.0 1.0 1.0
. . . . . . . 1.0 1.0
. . . . . . . . 1.0
. . . . . . . . . 1.0]
```

```
INFO = 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.
Example 4

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:

$$
A = \begin{bmatrix}
(25.0, 5.0) & (51.0, 5.0) & (71.0, 5.0) \\
(-5.0, 5.0) & (10.0, -5.0) & (4.0, 6.0) \\
(10.0, -5.0) & (4.0, 6.0) & (71.0, 5.0)
\end{bmatrix}
$$

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:

```
CALL CPOTRF( 'L', 3, A, 3, INFO )
```

or

```
CALL CPOF( 'L', A, 3, INFO )
```

Output:

$$
A = \begin{bmatrix}
(5.0, 0.0) & (1.0, 1.0) & (1.0, -1.0) \\
(-1.0, 1.0) & (1.0, 1.0) & (1.0, -1.0) \\
(2.0, -1.0) & (1.0, 1.0) & (1.0, -1.0)
\end{bmatrix}
$$

INFO = 0

Example 5

Chapter 10. Linear Algebraic Equations 595
This example shows a factorization (using the Cholesky factorization method) of the same positive definite symmetric matrix $A$ of order 9 used in Example 9, but 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:

```fortran
CALL SPTRF( 'L', 9, AP, INFO )
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, 2.0,
       3.0, 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, 4.0,
       5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0,
       6.0, 6.0, 6.0, 6.0, 6.0, 6.0, 6.0, 6.0, 6.0,
       7.0, 7.0, 7.0, 7.0, 7.0, 7.0, 7.0, 7.0, 7.0,
       8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0,
       9.0 )
```

Output:

```fortran
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,
       1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
       1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
       1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0 )
```

INFO = 0

**Example 6**

This example shows a factorization (using the Cholesky factorization method) of the same positive definite symmetric matrix $A$ of order 9 used in Example 9, but stored in upper-packed storage mode.

**Note:** The AP arrays are formatted in a triangular arrangement for readability; however, they are stored in upper-packed storage mode.

Call Statement and Input:

```fortran
CALL SPTRF( 'U', 9, AP, INFO )
AP = ( 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,
       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:

```fortran
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,
```

INFO = 0
Example 7

This example shows a factorization (using the Cholesky factorization method) of the same positive definite complex Hermitian matrix $A$ of order 3 used in Example 3, but stored in lower-packed storage mode.

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:

```
CALL ZPPTRF('L', 3, AP, INFO)
```

$AP = ((25.0, .), (-5.0, 5.0), (10.0, 10.0), (25.0, .), (4.0, 6.0), (71.0, .))$

Output:

$AP = ((5.0, 0.0), (-1.0, 1.0), (2.0, -1.0), (7.0, 0.0), (1.0, 1.0), (8.0, 0.0))$

INFO = 0

Example 8

This example shows a factorization (using the Cholesky factorization method) of the same positive definite complex Hermitian matrix $A$ of order 3 used in Example 4, but stored in upper-packed storage mode.

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:

```
CALL ZPPTRF('U', 3, AP, INFO)
```

$AP = ((8.0, .), (3.0, 3.0), (18.0, .), (3.0, -3.0), (8.0, -6.0), (2.0, 2.0), (6.0, 0.0))$

Output:

$AP = ((3.0, 0.0), (1.0, 1.0), (4.0, 0.0), (1.0, -1.0), (2.0, -1.0), (6.0, 0.0))$

INFO = 0

Example 9

This example shows a factorization (using the Gaussian elimination method) of positive definite symmetric matrix $A$ of order 9, stored in lower-packed storage mode, where on input matrix $A$ is:

$$
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 \\
1.0 & 2.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 4.0 & 4.0 & 4.0 & 4.0 & 4.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 5.0 & 5.0 & 5.0 & 5.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 6.0 & 6.0 & 6.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 7.0 & 7.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 8.0 & 8.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 8.0 & 9.0 \\
\end{bmatrix}
$$

INFO = 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:**

```fortran
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, 2.0,
     3.0, 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, 4.0,
     5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0,
     6.0, 6.0, 6.0, 6.0, 6.0, 6.0, 6.0, 6.0, 6.0,
     7.0, 7.0, 7.0, 7.0, 7.0, 7.0, 7.0, 7.0, 7.0,
     8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 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 10**

This example shows a factorization (using the Cholesky factorization method) of the same positive definite symmetric matrix $A$ of order 9 used in Example 9, 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:**

```fortran
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, 2.0,
     3.0, 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, 4.0,
     5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0,
     6.0, 6.0, 6.0, 6.0, 6.0, 6.0, 6.0, 6.0, 6.0,
     7.0, 7.0, 7.0, 7.0, 7.0, 7.0, 7.0, 7.0, 7.0,
     8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 8.0, 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,
     ..., ..., ..., ..., ..., ..., ..., ..., ...) )
```
**SPOTRS, DPOTRS, CPOTRS, ZPOTRS, SPOSM, DPOSM, CPOSM, ZPOSM, SPPTRS, DPPTRS, CPPTRS, and ZPPTRS** (Positive Definite Real Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)

### Purpose

These subroutines solve the system $AX = B$ for $X$, where $X$ and $B$ are general matrices and:
- For SPOTRS, DPOTRS, SPOSM, DPOSM, SPPTRS, and DPPTRS, $A$ is a positive definite real symmetric matrix.
- For CPOTRS, ZPOTRS, CPOSM, ZPOSM, CPPTRS, and ZPPTRS, $A$ is a positive definite complex Hermitian matrix.

SPOTRS, DPOTRS, CPOTRS, and ZPOTRS use the results of the factorization of matrix $A$, produced by a preceding call to SPOTRF, DPOTRF, CPOTRF, or ZPOTRF, respectively.

SPOSM, DPOSM, CPOSM, and ZPOSM use the results of the factorization of matrix $A$, produced by a preceding call to SPOF/SPOFCD, DPOF/DPOFCD, CPOF, or ZPOF, respectively.

SPPTRS, DPPTRS, CPPTRS, and ZPPTRS use the results of the factorization of matrix $A$, produced by a preceding call to SPPTRF, DPPTRF, CPPTRF, or ZPPTRF, respectively.

### Table 144. Data Types

<table>
<thead>
<tr>
<th>$A$, $B$, $X$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SPOTRS, SPOSM, and SPPTRS</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DPOTRS, DPOSM, and DPPTRS</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CPOTRS, CPOSM, and CPPTRS</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZPOTRS, ZPOSM, and ZPPTRS</td>
</tr>
</tbody>
</table>

^LAPACK

**Note:** The input to these solve subroutines must be the output from the corresponding factorization subroutines.

### Syntax

**Fortran**

```fortran
CALL SPOTRS | DPOTRS | CPOTRS | ZPOTRS (uplo, n, nrhs, a, lda, bx, ldb, info)
CALL SPOSM | DPOSM | CPOSM | ZPOSM (uplo, a, lda, n, bx, ldb, nrhs)
CALL SPPTRS | DPPTRS | CPPTRS | ZPPTRS (uplo, n, nrhs, ap, bx, ldb, info)
```

**C and C++**

```c
spotrs | dpotrs | cpotrs | zpotrs (uplo, n, nrhs, a, lda, bx, ldb, info);
sposm | dposm | cposm | zposm (uplo, a, lda, n, bx, ldb, nrhs);
spptrs | dpptrs | cpptrs | zpptrs (uplo, n, nrhs, ap, bx, ldb, info);
```
On Entry

**matrix_layout**

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If **matrix_layout** = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If **matrix_layout** = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. It must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR

**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 SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF/SPOFCD, DPOF/DPOFCD, CPOF, or ZPOF.

Specified as: an lda by (at least) n array, containing numbers of the data type indicated in Table 144 on page 600

**ap**

is an array, referred to as AP, in which the factorization of positive definite matrix A, produced by a preceding call to SPPTRF, DPPTRF, CPPTRF, or ZPPTRF, is stored in upper-packed or lower-packed storage mode.

Specified as: a one-dimensional array, containing numbers of the data type indicated in Table 144 on page 600

**lda**

is the leading dimension of the array specified for a.

Specified as: an integer; lda > 0 and lda ≥ n.

**n**

is the order of matrix A and the number of rows of matrix B.

Specified as: an integer; n ≥ 0.

**bx**

is the general matrix B, containing the nrhs right-hand sides of the system. The right-hand sides, each of length n, reside in the columns of B.

Specified as: an ldb by (at least) nrhs array, containing numbers of the data type indicated in Table 144 on page 600

**ldb**

is the leading dimension of the array specified for b.

Specified as: an integer; ldb > 0 and ldb ≥ n.

**nrhs**

is the number of right-hand sides; that is, the number of columns of matrix B.
Specified as: an integer; \( nrhs \geq 0 \).

**info**

See On Return

On Return

**bx** is the general matrix \( X \), containing the \( nrhs \) solutions to the system. The solutions, each of length \( n \), reside in the columns of \( X \).

Returned as: an \( ldb \) by (at least) \( nrhs \) array, containing numbers of the data type indicated in Table 144 on page 600

**info**

**info** has the following meaning:

If \( info = 0 \), the solve completed successfully.

Returned as:

- For SPOTRS, DPOTRS, CPOTRS, ZPOTRS, SPOSM, DPOSM, CPOSM, ZPOM, SPPTRS, DPPTRS, CPPTRS, and ZPPTR, returned as: an integer; \( info \geq 0 \).
- For LAPACKE_spotrs, LAPACKE_dpotrs, LAPACKE_cpotrs, LAPACKE_zpotrs, LAPACKE_spptrs, LAPACKE_dpptrs, LAPACKE_cpptrs, and LAPACKE_zpptr, returned as an integer function value; \( info \geq 0 \).

**Notes**

1. In your C program, argument \( info \) must be passed by reference.
2. All subroutines accept lowercase letters for the \( uplo \) argument.
3. 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 SPOTRF/SPOF/SPOFCD/SPPTTF, DPOTRF/DPOF/DPOFCD/DPPTTF, CPOTRF/CPOF/CPPTF, and ZPOTRF/ZPOF/ZPPTF, respectively.
4. The array data specified for input argument \( a \) for these subroutines must be the same as the corresponding output arguments for SPOTRF/SPOF/SPOFCD, DPOTRF/DPOF/DPOFCD, CPOTRF/CPOF, and ZPOTRF/ZPOF, respectively.
5. The array data specified for input argument \( ap \) for these subroutines must be the same as the corresponding output arguments for SPPTTF, DPPTTF, CPPTF, and ZPPTF, respectively.
6. The matrices used in this computation must have no common elements; otherwise, results are unpredictable. See “Concepts” on page 75.
7. For a description of how the matrices are stored:
   - For positive definite real symmetric matrices, see “Positive Definite or Negative Definite Symmetric Matrix” on page 89.
   - For positive definite complex Hermitian matrices, see “Positive Definite or Negative Definite Complex Hermitian Matrix” on page 91.

**Function**

The system \( AX = B \) is solved for \( X \), where \( X \) and \( B \) are general matrices and \( A \) is a positive definite real symmetric matrix for SPOTRS/SPOSM/SPPTTF and DPOTRS/DPOSM/DPPTTF, and a positive definite complex Hermitian matrix for CPOTRF/CPOF/CPPTF and ZPOTRF/ZPOF/ZPPTF. These subroutines use the results of the factorization of matrix \( A \), produced by a preceding call to SPOTRF/SPOF/SPOFCD/SPPTTF, DPOTRF/DPOF/DPOFCD/DPPTTF, CPOTRF/CPOF/CPPTF, or ZPOTRF/ZPOF/ZPPTF, respectively. For a description of how \( A \) is factored, see SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF,
DPOF, CPOF, ZPOF, SPPTRE, DPPTRE, CPPTRE, ZPPTRE, SPPF, and DPPF

(Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)” on page 587.

If \( n \) or \( nrhs \) is 0, no computation is performed. See references [8 on page 1363] and [44 on page 1366].

Error conditions

Computational Errors
None

Note: If the factorization performed by _POTRF, _POF, _POFCD, or _PPTRF failed because matrix \( A \) 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. \( \text{uplo} \neq 'U' \) or 'L'
2. \( n < 0 \)
3. \( nrhs < 0 \)
4. \( n > lda \)
5. \( lda \leq 0 \)
6. \( n > ldb \)
7. \( ldb \leq 0 \)

Examples

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 1 for SPOTRF and SPOF.

Call Statement and Input:

\[
\begin{align*}
\text{UPLO} & \quad N & \quad NRHS & \quad A & \quad LDA & \quad BX & \quad LDB & \quad INFO \\
& | & | & | & | & | & | \\
\end{align*}
\]

CALL SPOTRS( 'L', 9, 2, A, 9, BX, 9, INFO )

or

\[
\begin{align*}
\text{UPLO} & \quad A & \quad LDA & \quad N & \quad BX & \quad LDB & \quad NRHS \\
& | & | & | & | & | & | \\
\end{align*}
\]

CALL SPOSM( 'L', A, 9, 9, BX, 9, 2 )

\[
A = \begin{bmatrix}
9.0 & 45.0 \\
17.0 & 89.0 \\
24.0 & 131.0 \\
30.0 & 170.0 \\
\end{bmatrix}
\]

\[
BX = \begin{bmatrix}
35.0 & 205.0 \\
39.0 & 235.0 \\
42.0 & 259.0 \\
44.0 & 276.0 \\
45.0 & 285.0 \\
\end{bmatrix}
\]

Output:
Example 2

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 2 for SPOTRF and SPOF.

Call Statement and Input:

\[
\begin{array}{c}
\text{UPLO N NRHS A LDA BX LDB INFO} \\
\text{CALL SPOTRS( 'U', 9, 2, A, 9, BX, 9, INFO )}
\end{array}
\]

or

\[
\begin{array}{c}
\text{UPLO A LDA N BX LDB NRHS} \\
\text{CALL SPOSM( 'U', A, 9, 9, BX, 9, 2 )}
\end{array}
\]

\[
A = \begin{bmatrix}
9.0 & 45.0 \\
17.0 & 89.0 \\
24.0 & 131.0 \\
30.0 & 170.0
\end{bmatrix}
\]

\[
BX = \begin{bmatrix}
35.0 & 205.0 \\
39.0 & 235.0 \\
42.0 & 259.0 \\
44.0 & 276.0 \\
45.0 & 285.0
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
1.0 & 1.0 \\
1.0 & 2.0 \\
1.0 & 3.0 \\
1.0 & 4.0
\end{bmatrix}
\]

\[
BX = \begin{bmatrix}
1.0 & 5.0 \\
1.0 & 6.0 \\
1.0 & 7.0 \\
1.0 & 8.0 \\
1.0 & 9.0
\end{bmatrix}
\]

INFO = 0

Example 3

This example shows how to solve the system \( AX = B \) for two right-hand sides, where matrix \( A \) is the same matrix factored in Example 3 for CPOTRF and CPOF.

Call Statement and Input:
CALL CPOTRS('L', 3, 2, A, 3, BX, 3, INFO)

or

CALL CPOSM('L', A, 3, 3, BX, 3, 2)

A = (same as output A in Example 3)

\[
\begin{pmatrix}
60.0 & -55.0 \\
34.0 & 58.0 \\
13.0 & -152.0
\end{pmatrix}
\begin{pmatrix}
70.0 & 10.0 \\
-51.0 & 110.0 \\
75.0 & 63.0
\end{pmatrix}
\]

BX =

\[
\begin{pmatrix}
34.0 & 58.0 \\
-51.0 & 110.0 \\
13.0 & -152.0
\end{pmatrix}
\begin{pmatrix}
2.0 & -1.0 \\
7.0 & 2.0
\end{pmatrix}
\begin{pmatrix}
2.0 & 0.0 \\
-51.0 & 110.0 \\
75.0 & 63.0
\end{pmatrix}
\]

Output:

\[
\begin{pmatrix}
34.0 & 58.0 \\
-51.0 & 110.0 \\
13.0 & -152.0
\end{pmatrix}
\begin{pmatrix}
3.0 & 0.0 \\
-1.0 & 3.0
\end{pmatrix}
\begin{pmatrix}
2.0 & -1.0 \\
7.0 & 2.0
\end{pmatrix}
\begin{pmatrix}
2.0 & 0.0 \\
-51.0 & 110.0 \\
75.0 & 63.0
\end{pmatrix}
\]

INFO = 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 4 for CPOTRF and CPOF.

Call Statement and Input:

CALL CPOTRS('U', 3, 2, A, 3, BX, 3, INFO)

or

CALL CPOSM('U', A, 3, 3, BX, 3, 2)

A = (same as output A in Example 4)

\[
\begin{pmatrix}
33.0 & -18.0 \\
45.0 & -45.0 \\
152.0 & 1.0
\end{pmatrix}
\begin{pmatrix}
15.0 & -3.0 \\
8.0 & -2.0 \\
43.0 & -29.0
\end{pmatrix}
\]

Output:

\[
\begin{pmatrix}
33.0 & -18.0 \\
45.0 & -45.0 \\
152.0 & 1.0
\end{pmatrix}
\begin{pmatrix}
2.0 & -1.0 \\
1.0 & -1.0 \\
3.0 & 0.0
\end{pmatrix}
\begin{pmatrix}
2.0 & 0.0 \\
1.0 & -1.0
\end{pmatrix}
\begin{pmatrix}
3.0 & 0.0 \\
-1.0 & 5.0
\end{pmatrix}
\begin{pmatrix}
2.0 & 0.0 \\
-51.0 & 110.0 \\
75.0 & 63.0
\end{pmatrix}
\]

INFO = 0

Example 5

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 5 for SPPTF and DPPTF.

Call Statement and Input:
CALL SPPTRS( 'L', 9, 2, AP, BX, 9, INFO )

\[ A = (\text{same as output } A \text{ in Example 5}) \]

\[
\begin{bmatrix}
9.0 & 45.0 \\
17.0 & 89.0 \\
24.0 & 131.0 \\
30.0 & 170.0 \\
\end{bmatrix}
\]

\[ BX = \begin{bmatrix}
35.0 & 205.0 \\
39.0 & 235.0 \\
42.0 & 259.0 \\
44.0 & 276.0 \\
45.0 & 285.0 \\
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
1.0 & 1.0 \\
1.0 & 2.0 \\
1.0 & 3.0 \\
1.0 & 4.0 \\
\end{bmatrix}
\]

\[ BX = \begin{bmatrix}
1.0 & 5.0 \\
1.0 & 6.0 \\
1.0 & 7.0 \\
1.0 & 8.0 \\
1.0 & 9.0 \\
\end{bmatrix}
\]

INFO = 0

**Example 6**

This example shows how to solve the system \( AX = B \) for two right-hand sides, where matrix \( A \) is the same matrix factored in Example 6 for SPPTRF.

Call Statement and Input:

\[
\begin{bmatrix}
9.0 & 45.0 \\
17.0 & 89.0 \\
24.0 & 131.0 \\
30.0 & 170.0 \\
\end{bmatrix}
\]

\[ BX = \begin{bmatrix}
35.0 & 205.0 \\
39.0 & 235.0 \\
42.0 & 259.0 \\
44.0 & 276.0 \\
45.0 & 285.0 \\
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
1.0 & 1.0 \\
1.0 & 2.0 \\
1.0 & 3.0 \\
1.0 & 4.0 \\
\end{bmatrix}
\]

\[ BX = \begin{bmatrix}
1.0 & 5.0 \\
1.0 & 6.0 \\
\end{bmatrix}
\]
Example 7

This example shows how to solve the system \( AX = B \) for two right-hand sides, where matrix \( A \) is the same matrix factored in Example 7 for ZPPTRF.

Call Statement and Input:

```
CALL ZPPTRS('L', 3, 2, AP, BX, 3, INFO)
```

\[
\begin{pmatrix}
1.0 & 7.0 \\
1.0 & 8.0 \\
1.0 & 9.0 \\
\end{pmatrix}
\]

INFO = 0

Output:

```
\begin{pmatrix}
(2.0, -1.0) & (2.0, 0.0) \\
(1.0, 1.0) & (-1.0, 2.0) \\
(0.0, -2.0) & (1.0, 1.0) \\
\end{pmatrix}
```

INFO = 0

Example 8

This example shows how to solve the system \( AX = B \) for two right-hand sides, where matrix \( A \) is the same matrix factored in Example 8 for ZPPTRF.

Call Statement and Input:

```
CALL ZPPTRS('U', 3, 2, AP, BX, 3, INFO)
```

\[
\begin{pmatrix}
33.0 & -18.0 \\
45.0 & -45.0 \\
152.0 & 1.0 \\
\end{pmatrix}
\]

INFO = 0
SPPS and DPPS (Positive Definite Real Symmetric Matrix Solve)

Purpose

These subroutines solve the system $Ax = b$ for $x$, where $A$ is a positive definite symmetric matrix, and $x$ and $b$ are vectors. The subroutines use the results of the factorization of matrix $A$, produced by a preceding call to SPPF/SPPFCD or DPPF/DPPFP/DPPFCD, respectively.

Table 145. Data Types

<table>
<thead>
<tr>
<th>A, b, x</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SPPS</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DPPS</td>
</tr>
</tbody>
</table>

Note: The input to these solve subroutines must be the output from the factorization subroutines SPPF/SPPFCD and DPPF/DPPFP/DPPFCD, respectively.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SPPS</th>
<th>DPPS (ap, n, bx, iopt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>spps</td>
<td>dpps (ap, n, bx, iopt);</td>
</tr>
</tbody>
</table>

On Entry

$ap$ is the factorization of matrix $A$, 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 145, 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 $A$ used in the factorization, and the lengths of vectors $b$ and $x$.

Specified as: an integer; $n \geq 0$.

$bx$ 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 145.

$iopt$ indicates the type of factorization that was performed on matrix $A$, where:

- If $iopt = 0$, the matrix was factored using the $LDL^T$ method.
- If $iopt = 1$, the matrix was factored using Cholesky factorization.

Specified as: an 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 145.
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 as follows:

<table>
<thead>
<tr>
<th>SPPF/DPPF Input <em>iopt</em></th>
<th>SPPS/DPPS Input <em>iopt</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 or 10</td>
<td>0</td>
</tr>
<tr>
<td>1 or 11</td>
<td>1</td>
</tr>
</tbody>
</table>

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 75.
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 85.

Function
The system \( Ax = b \) is solved for \( x \), where \( A \) is a positive definite symmetric matrix, stored in lower-packed storage mode in array *AP*, and \( x \) and \( b \) are vectors. These subroutines use the results of the factorization of matrix \( A \), produced by a preceding call to SPPF/SPPFCD or DPPF/DPPFP/DPPFCD, respectively.

If *n* is 0, no computation is performed. See references [44 on page 1366] and [46 on page 1366].

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

Examples
Example 1
This example shows how to solve the system \( Ax = b \), where matrix \( A \) is the same matrix factored in the [Example 9] for SPPF and DPPF.
Call Statement and Input:

\[
\begin{array}{c|c|c|c}
\text{AP} & \text{N} & \text{BX} & \text{IOPT} \\
\hline
\text{CALL SPPS (} & \text{AP} & , & 9 , \text{ BX , } 0 \text{ )}
\end{array}
\]

\[
\begin{align*}
\text{AP} &= \text{(same as output AP in Example 9 for SPPF and DPPF)} \\
\text{BX} &= (9.0, 17.0, 24.0, 30.0, 35.0, 39.0, 42.0, 44.0, 45.0)
\end{align*}
\]

Output:

\[
\begin{align*}
\text{BX} &= (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
\end{align*}
\]

Example 2

This example shows how to solve the same system as in Example 1, where matrix \( A \) is the same matrix factored in Example 10 for SPPF and DPPF.

Call Statement and Input:

\[
\begin{array}{c|c|c|c}
\text{AP} & \text{N} & \text{BX} & \text{IOPT} \\
\hline
\text{CALL SPPS(} & \text{AP} & , & 9 , \text{ BX , } 1 \text{ )}
\end{array}
\]

\[
\begin{align*}
\text{AP} &= \text{(same as output AP in Example 10 for SPPF and DPPF)} \\
\text{BX} &= (9.0, 17.0, 24.0, 30.0, 35.0, 39.0, 42.0, 44.0, 45.0)
\end{align*}
\]

Output:

\[
\begin{align*}
\text{BX} &= (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
\end{align*}
\]
SPOCON, DPOCON, CPOCON, ZPOCON, SPPCON, DPPCON, CPPCON, and ZPPCON (Estimate the Reciprocal of the Condition Number of a Positive Definite Real Symmetric or Complex Hermitian Matrix)

Purpose

These subroutines estimate the reciprocal of the condition number of matrix $A$ as explained below:

SPOCON, DPOCON, CPOCON, and ZPOCON

The SPOCON, DPOCON, CPOCON, and ZPOCON subroutines estimate the reciprocal of the condition number of matrix $A$, stored in upper or lower storage mode, where:

- For SPOCON and DPOCON, $A$ is a positive definite real symmetric matrix.
- For CPOCON and ZPOCON, $A$ is a positive definite complex Hermitian matrix.

These subroutines use the results of the factorization of matrix $A$ produced by a preceding call to SPOTRF, DPOTRF, CPOTRF, or ZPOTRF, respectively.

SPPCON, DPPCON, CPPCON, and ZPPCON

The SPPCON, DPPCON, CPPCON, and ZPPCON subroutines estimate the reciprocal of the condition number of matrix $A$, stored in upper-packed or lower-packed storage mode, where:

- For SPPCON and DPPCON, $A$ is a positive definite real symmetric matrix.
- For CPPCON and ZPPCON, $A$ is a positive definite complex Hermitian matrix.

These subroutines use the results of the factorization of matrix $A$ produced by a preceding call to SPPTRF, DPPTRF, CPPTRF, or ZPPTRF, respectively.

For details about the factorization subroutines, see “SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPPEF, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)” on page 587.

Table 146. Data Types

<table>
<thead>
<tr>
<th>$A$, work</th>
<th>anorm, rcond, rwork</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SPOCON, SPPCON</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DPOCON, DPPCON</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>CPOCON, CPPCON</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>ZPOCON, ZPPCON</td>
</tr>
</tbody>
</table>

Syntax

Fortran

| CALL SPOCON | DPOCON (uplo, n, a, lda, anorm, rcond, work, iwork, info) |
| CALL CPOCON | ZPOCON (uplo, n, a, lda, anorm, rcond, work, rwork, info) |
| CALL SPPCON | DPPCON (uplo, n, ap, anorm, rcond, work, iwork, info)    |
| CALL CPPCON | ZPPCON (uplo, n, ap, anorm, rcond, work, rwork, info)    |
C and C++

```
spocon | dpocon (uplo, n, a, lda, anorm, rcond, work, iwork, info);
cpocon | zpocon (uplo, n, a, lda, anorm, rcond, work, rwork, info);
sppcon | dppcon (uplo, n, ap, anorm, rcond, work, iwork, info);
cppcon | zppcon (uplo, n, ap, anorm, rcond, work, rwork, info);
```

```
info = LAPACKE_spocon | LAPACKE_dpocon (matrix_layout, uplo, n, a, lda, anorm, rcond);
info = LAPACKE_cpocon | LAPACKE_zpocon (matrix_layout, uplo, n, a, lda, anorm, rcond);
info = LAPACKE_sppcon | LAPACKE_dppcon (matrix_layout, uplo, n, ap, anorm, rcond);
info = LAPACKE_cppcon | LAPACKE_zppcon (matrix_layout, uplo, n, ap, anorm, rcond);
```

**On Entry**

- **matrix_layout**
  - Indicates whether the input and output matrices are stored in row major order or column major order, where:
    - If `matrix_layout = LAPACK_ROW_MAJOR`, the matrices are stored in row major order.
    - If `matrix_layout = LAPACK_COL_MAJOR`, the matrices are stored in column major order.
  - Specified as: an integer. It must be `LAPACK_ROW_MAJOR` or `LAPACK_COL_MAJOR`.

- **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'.

- **n**
  - The order of the factored matrix `A` used in the computation.
  - Specified as: an integer; `n ≥ 0`.

- **ap**
  - Is an array, referred to as `AP`, containing the factorization of the positive definite matrix `A` produced by a preceding call to `SPPTFR`, `DPPTFR`, `CPPTFR`, or `ZPPTFR`, respectively, stored in upper-packed or lower-packed storage mode.
  - Specified as: a one-dimensional array, containing numbers of the data type indicated in Table 146 on page 611. See "Notes" on page 614.

- **a**
  - The factorization of positive definite matrix `A` produced by a preceding call to `SPOTRF`, `DPOTRF`, `CPOTRF`, or `ZPOTRF`, respectively, 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 146 on page 611.

- **lda**
  - Is the leading dimension of matrix `A`.
  - Specified as: an integer; `lda > 0` and `lda ≥ n`.

- **anorm**
  - Is the one norm of the original matrix `A`.

**For SPOCON, DPOCON, CPOCON, and ZPOCON**

To obtain the value of `anorm`, make a preceding call to `SLANSY`, `DLANSY`, `CLANHE`, or `ZLANHE`, respectively.
For SPPCON, DPPCON, CPPCON, and ZPPCON

To obtain the value of anorm, make a preceding call to SLANSP, DLANSP, CLANHP, or ZLANHP, respectively.

Refer to “SLANSY, DLANSY, CLANHE, ZLANHE, SLANSP, DLANSP, CLANHP, and ZLANHP (Real Symmetric or Complex Hermitian Matrix Norm)” on page 637.

Specified as: a number \( \geq 0.0 \), of the data type indicated in Table 146 on page 611.

rcond

See On Return.

work

is the work area used by this subroutine, where:

For SPOCON, DPOCON, SPPCON, and DPPCON

The size of work is (at least) of length \( 3n \).

For CPOCON, ZPOCON, CPPCON, and ZPPCON

The size of work is (at least) of length \( 2n \).

Specified as: an area of storage containing numbers of data type indicated in Table 146 on page 611.

iwork

is a work area used by this subroutine whose size is (at least) of length \( n \).

Specified as: an area of storage containing integers.

rwork

is a work area used by this subroutine whose size is (at least) of length \( n \).

Specified as: an area of storage containing numbers of the data type indicated in Table 146 on page 611.

info

See On Return.

On Return

rcond

has the following meaning:

If \( info = 0 \), an estimate of the reciprocal of the condition number of matrix \( A \) is returned; i.e., \( rcond = 1.0 / (\text{NORM}(A) \times \text{NORM}(A^{-1})) \).

If \( n = 0 \), the subroutines return with \( rcond = 1.0 \).

If \( n \neq 0 \) and \( \text{anorm} = 0.0 \), the subroutines return with \( rcond = 0.0 \).

Returned as: a number \( \geq 0.0 \), of the data type indicated in Table 146 on page 611.

info

has the following meaning:

If \( info = 0 \), the computation completed normally.

Returned as:

- For SPOCON, DPOCON, CPOCON, ZPOCON, SPPCON, DPPCON, CPPCON, and ZPPCON, returned as: an integer; \( info \geq 0 \).
For LAPACKE_spocon, LAPACKE_dpocon, LAPACKE_cpocon, LAPACKE_zpocon, LAPACKE_sppcon, LAPACKE_dppcon, LAPACKE_cppcon, and LAPACKE_zppcon, returned as an integer function value; info ≥ 0.

Returned as: an integer; info = 0.

Notes
1. In your C program, arguments rcond and info must be passed by reference.
2. This subroutine accepts lowercase letters for the uplo argument.
3. For input arguments uplo, lda, and n, the following must be true:
   - For SPOCON/DPOCON/CPOCON/ZPOCON
     The scalar data specified for uplo, lda, and n must be the same as the scalar data specified for SLANSY/DLANSY/CLANHE/ZLANHE and SPOTRF/DPOTRF/CPOTRF/ZPOTRF.
   - For SPPCON/DPPCON/CPPCON/ZPPCON
     The scalar data specified for uplo and n must be the same as the scalar data specified for SLANSP/DLANSP/CLANHP/ZLANHP and SPTRF/DPTRF/CPTRF/ZPTRF.
4. For matrix A, the following must be true:
   - For SPOCON/DPOCON/CPOCON/ZPOCON
     The matrix A input to SLANSY/DLANSY/CLANHE/ZLANHE must be the same as the corresponding input argument for SPOTRF/DPOTRF/CPOTRF/ZPOTRF.
   - For SPPCON/DPPCON/CPPCON/ZPPCON
     The matrix A input to SLANSP/DLANSP/CLANHP/ZLANHP must be the same as the corresponding input argument for SPTRF/DPTRF/CPTRF/ZPTRF.
5. On both input and output, matrix A conforms to LAPACK format.
6. 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 89.
   - For positive definite complex Hermitian matrices, see “Positive Definite or Negative Definite Complex Hermitian Matrix” on page 91.

Function
The reciprocal of the condition number of general matrix A is estimated, using the results of the factorization of matrix A produced by a preceding factorization call.

\[ rcond = 1.0/(\text{NORM}(A) \times \text{NORM}(A^{-1})). \]

If \( n = 0 \), the subroutines return with \( rcond = 1.0 \).

If \( n \neq 0 \) and \( anorm = 0.0 \), the subroutines return with \( rcond = 0.0 \).

See reference [82 on page 1368].

Error conditions
Resource Errors
None.
Computational Errors
None.

Input-Argument Errors
1. \textit{uplo} ≠ 'U' or 'L'
2. \( n < 0 \)
3. \( n > lda \)
4. \( lda \leq 0 \)
5. \( anorm < 0 \)
6. \( anorm \neq 0 \) and \( anorm > big \) or \( anorm < tiny \)

Where:

For \texttt{SPOCON}, \texttt{SPPCON}, \texttt{CPOCON}, and \texttt{CPPCON}

\( big \) and \( tiny \) have the following values:
\[
\begin{align*}
big &= 2^{127} \times (1 - \text{ULP}) \\
tiny &= 2^{-126} \times 2^{21}
\end{align*}
\]

For \texttt{DPOCON}, \texttt{DPPCON}, \texttt{ZPOCON}, and \texttt{ZPPCON}

\( big \) and \( tiny \) have the following values:
\[
\begin{align*}
big &= 2^{1023} \times (1 - \text{ULP}) \\
tiny &= 2^{-1022} \times 2^{50}
\end{align*}
\]

Where ULP = unit in last place.

Note: To avoid this error, scale matrix \( A \) so that \( tiny \leq anorm \leq big \).

Examples

Example 1

This example estimates the reciprocal of the condition number of positive definite real symmetric matrix \( A \) stored in lower storage mode. The input matrix \( A \) to \texttt{DLANSY} and \texttt{DPOTRF} is the same as input matrix \( A \) in Example 1.

Call Statements and Input:

\begin{verbatim}
NORM UPLO N A LDA WORK
\end{verbatim}

\[
\begin{align*}
\text{ANORM} &= \text{DLANSY( '1', 'L', 9, A, 9, WORK )} \\
\text{UPLO} &= \text{N} \quad \text{A} \quad \text{LDA} \quad \text{INFO} \\
\text{CALL DPOTRF( 'L', 9, A, 9, INFO )}
\end{align*}
\]

\[
\begin{align*}
\text{UPLO} &= \text{N} \quad \text{A} \quad \text{LDA} \quad \text{ANORM} \quad \text{RCOND} \quad \text{WORK} \quad \text{IWORK} \quad \text{INFO} \\
\text{CALL DPOCON( 'L', 9, A, 9, ANORM, RCOND, WORK, IWORK, INFO )}
\end{align*}
\]

\[
\begin{align*}
\text{A} &= \text{(same as output A in Example 1)} \\
\text{ANORM} &= \text{(same as output ANORM in Example 1)}
\end{align*}
\]

Output:

\[
\begin{align*}
\text{RCOND} &= 5.56 \times 10^{-3} \\
\text{INFO} &= 0
\end{align*}
\]

Example 2

This example estimates the reciprocal of the condition number of positive definite real symmetric matrix \( A \) stored in upper storage mode. The input matrix \( A \) to \texttt{DLANSY} and \texttt{DPOTRF} is the same as input matrix \( A \) in Example 2.

\begin{verbatim}
NORM UPLO N A LDA WORK
\end{verbatim}

\[
\begin{align*}
\text{ANORM} &= \text{DLANSY( '1', 'L', 9, A, 9, WORK )} \\
\text{UPLO} &= \text{N} \quad \text{A} \quad \text{LDA} \quad \text{ANORM} \quad \text{RCOND} \quad \text{WORK} \quad \text{IWORK} \quad \text{INFO} \\
\text{CALL DPOCON( 'L', 9, A, 9, ANORM, RCOND, WORK, IWORK, INFO )}
\end{align*}
\]

\[
\begin{align*}
\text{A} &= \text{(same as output A in Example 2)} \\
\text{ANORM} &= \text{(same as output ANORM in Example 2)}
\end{align*}
\]
Call Statements and Input:

\[
\begin{align*}
\text{NORM} & \quad \text{UPLO} \quad \text{N} \quad \text{A} \quad \text{LDA} \quad \text{WORK} \\
\text{ANORM} = \text{DLANSY}( '1', 'U', 9, A, 9, \text{WORK} ) \\
\text{UPLO} & \quad \text{N} \quad \text{A} \quad \text{LDA} \quad \text{INFO} \\
\text{CALL} & \quad \text{DPOTRF}( 'U', 9, A, 9, \text{INFO} ) \\
\text{UPLO} & \quad \text{N} \quad \text{A} \quad \text{LDA} \quad \text{ANORM} \quad \text{RCOND} \quad \text{WORK} \quad \text{IWORK} \quad \text{INFO} \\
\text{CALL} & \quad \text{DPOCON}( 'U', 9, A, 9, \text{ANORM}, \text{RCOND}, \text{WORK}, \text{IWORK}, \text{INFO} ) \\
A & = (\text{same as output A in Example 2}) \\
\text{ANORM} & = (\text{same as output ANORM in Example 2}) \\
\end{align*}
\]

Output:

\[
\begin{align*}
\text{RCOND} & = 5.56 \times 10^{-3} \\
\text{INFO} & = 0 \\
\end{align*}
\]

Example 3

This example estimates the reciprocal of the condition number of positive definite complex Hermitian matrix \(A\) stored in lower storage mode. The input matrix \(A\) to ZLANHE and ZPOTRF is the same as input matrix \(A\) in Example 2.

Call Statements and Input:

\[
\begin{align*}
\text{NORM} & \quad \text{UPLO} \quad \text{N} \quad \text{A} \quad \text{LDA} \quad \text{WORK} \\
\text{ANORM} = \text{ZLANHE}( '1', 'L', 3, A, 3, \text{RWORK} ) \\
\text{UPLO} & \quad \text{N} \quad \text{A} \quad \text{LDA} \quad \text{INFO} \\
\text{CALL} & \quad \text{ZPOTRF}( 'L', 3, A, 3, \text{INFO} ) \\
\text{UPLO} & \quad \text{N} \quad \text{A} \quad \text{LDA} \quad \text{ANORM} \quad \text{RCOND} \quad \text{WORK} \quad \text{RWORK} \quad \text{INFO} \\
\text{CALL} & \quad \text{ZPOCON}( 'L', 3, A, 3, \text{ANORM}, \text{RCOND}, \text{WORK}, \text{RWORK}, \text{INFO} ) \\
A & = (\text{same as output A in Example 3}) \\
\text{ANORM} & = (\text{same as output ANORM in Example 3}) \\
\end{align*}
\]

Output:

\[
\begin{align*}
\text{RCOND} & = 1.85 \times 10^{-1} \\
\text{INFO} & = 0 \\
\end{align*}
\]

Example 4

This example estimates the reciprocal of the condition number of positive definite complex Hermitian matrix \(A\) stored in upper storage mode. The input matrix \(A\) to ZLANHE and ZPOTRF is the same as input matrix \(A\) in Example 4.

Call Statements and Input:

\[
\begin{align*}
\text{NORM} & \quad \text{UPLO} \quad \text{N} \quad \text{A} \quad \text{LDA} \quad \text{WORK} \\
\text{ANORM} = \text{ZLANHE}( '1', 'U', 3, A, 3, \text{RWORK} ) \\
\end{align*}
\]
Example 5

This example estimates the reciprocal of the condition number of positive definite real symmetric matrix $A$ stored in lower-packed storage mode. The input matrix $A$ to DLANSF and DPPTRF is the same as input matrix $A$ in Example 5.

Call Statements and Input:

```fortran
CALL ZPOTRF( 'U', 3, A, 3, INFO )
CALL ZPOCON( 'U', 3, A, 3, ANORM, RCOND, WORK, RWORK, INFO )
```

Output:

$\text{RCOND} = 1.01 \times 10^{-1}$

$\text{INFO} = 0$

Example 6

This example estimates the reciprocal of the condition number of positive definite real symmetric matrix $A$ stored in upper-packed storage mode. The input matrix $A$ to DLANSF and DPPTRF is the same as input matrix $A$ in Example 6.

Call Statements and Input:

```fortran
CALL ZPOTRF( 'U', 3, A, 3, INFO )
CALL ZPOCON( 'U', 3, A, 3, ANORM, RCOND, WORK, RWORK, INFO )
```

Output:

$\text{RCOND} = 5.56 \times 10^{-3}$

$\text{INFO} = 0$
CALL DPPCON( 'U', 9, AP, ANORM, RCOND, WORK, IWORK, INFO )

AP = (same as output AP in Example 6)
ANORM = (same as output ANORM in Example 6)

Output:
RCOND = 5.56 \times 10^{-3}
INFO = 0

Example 7
This example estimates the reciprocal of the condition number of positive definite complex Hermitian matrix $A$ stored in lower-packed storage mode. The input matrix $A$ to ZLANHP and ZPPTRF is the same as input matrix $A$ in Example 7.

Call Statements and Input:

\[
\begin{align*}
\text{ANORM} &= \text{ZLANHP( '1', 'L', 3, AP, RWORK )} \\
\text{UPLO} \ N \ AP \ \text{INFO} &= \text{ZPPTRF( 'L', 3, AP, INFO )}\\
\text{UPLO} \ N \ AP \ \text{ANORM} \ RCOND \ WORK \ RWORK \ INFO &= \text{ZPPCON( 'L', 3, AP, ANORM, RCOND, WORK, RWORK, INFO )}
\end{align*}
\]

AP = (same as output AP in Example 7)
ANORM = (same as output ANORM in Example 7)

Output:
RCOND = 1.85 \times 10^{1}
INFO = 0

Example 8
This example estimates the reciprocal of the condition number of positive definite complex Hermitian matrix $A$ stored in upper-packed storage mode. The input matrix $A$ to ZLANHP and ZPPTRF is the same as input matrix $A$ in Example 8.

Call Statements and Input:

\[
\begin{align*}
\text{ANORM} &= \text{ZLANHP( '1', 'U', 3, AP, RWORK )} \\
\text{UPLO} \ N \ AP \ \text{INFO} &= \text{ZPPTRF( 'U', 3, AP, INFO )}\\
\text{UPLO} \ N \ AP \ \text{ANORM} \ RCOND \ WORK \ RWORK \ INFO &= \text{ZPPCON( 'U', 3, AP, ANORM, RCOND, WORK, RWORK, INFO )}
\end{align*}
\]
$\text{AP} = \text{(same as output AP in Example 8)}$

$\text{ANORM} = \text{(same as output ANORM in Example 8)}$

**Output:**

$\text{RCOND} = 1.01 \times 10^{-1}$

$\text{INFO} = 0$
SPPFCD, DPPFCD, SPOFCD, and DPOFCD (Positive Definite Real Symmetric Matrix Factorization, Condition Number Reciprocal, and Determinant)

Purpose

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 $U^TU$). 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>
<thead>
<tr>
<th>Table 147. Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A,\ aux,\ r\ cond,\ det$</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
</tbody>
</table>

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

```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++

```c
spffcd | dpffcd (ap, n, iopt, rcond, det, aux, naux);
spofcd | dopofcd (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 147.
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 147 on page 620](#).

\( lda \)

is the leading dimension of the array specified for \( a \).

Specified as: an integer; \( lda > 0 \) and \( lda \geq n \).

\( n \)

is the order \( n \) of matrix \( A \).

Specified as: an 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: an 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 147 on page 620](#).

\( naux \)

is the size of the work area specified by \( aux \)—that is, the number of elements in \( aux \).

Specified as: an 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 622](#). Returned as: a one-dimensional array of (at least) length \( n(n+1)/2+n \), containing numbers of the data type indicated in [Table 147 on page 620](#).

\( a \)

is the transformed matrix \( A \) of order \( n \), containing the results of the
factorization. See “Function.” Returned as: a two-dimensional array, containing numbers of the data type indicated in Table 147 on page 620

\[ rcond \]

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 147 on page 620; \( rcond \geq 0 \).

\[ \text{det} \]

is the vector \( \text{det} \), containing the two components \( \text{det}_1 \) and \( \text{det}_2 \) of the determinant of matrix \( A \). The determinant is:

\[
\text{det}_1 \left(10^{\text{det}_2}\right)
\]

where \( 1 \leq \text{det}_1 < 10 \). Returned as: an array of length 2, containing numbers of the data type indicated in Table 147 on page 620

**Notes**

1. All subroutines accept lowercase letters for the \( \text{uplo} \) 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. SPPFCD and DPPFCD in many cases utilize new algorithms based on recursive packed storage format. As a result, on output, the array specified for AP may be stored in this new format rather than the conventional lower packed format. (See references [61 on page 1367], [77 on page 1368], and [79 on page 1368]). The array specified for AP should not be altered between calls to the factorization and solve subroutines; otherwise unpredictable results may occur.
5. See “Notes” on page 609 for information on specifying a value for \( iopt \) in the SPPS and DPPS subroutines after calling SPPFCD and DPPFCD, respectively.
6. 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.
7. 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 85. 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 89.
8. 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 51.

**Function**

The functions for these subroutines are described.

**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:

\[
A = LDL^T
\]
where:

$L$ is a unit lower triangular matrix.
$L^T$ 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 [81 on page 1368] and [82 on page 1368].

If $n$ is 0, no computation is performed. See references [44 on page 1366] and [46 on page 1366].

These subroutines call SPPF and DPPF, respectively, to perform the factorization using Gaussian elimination ($LDL^T$). 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:

$$A = LL^T$$ or $$A = U^TU$$

where:

$L$ is a lower triangular matrix.
$L^T$ is the transpose of matrix $L$.
$U$ is an upper triangular matrix.
$U^T$ 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 [81 on page 1368] and [82 on page 1368].

If $n$ is 0, no computation is performed. See references [8 on page 1363] and [44 on page 1366].

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 68.

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 68.

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.

Examples
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 9 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:

\[
\begin{align*}
\text{ AP } & \quad N \quad IOPT \quad RCOND \quad DET \quad AUX \quad NAUX \\
\text{ CALL DPPFCD( AP, 9, 3, RCOND, DET, AUX, 9 )}
\end{align*}
\]

AP = (same as input AP in Example 9)

Output:
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 1 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 $\| A \|_1 = 4.0$

On output, the value in $det, |A|$, is equal to 1.

Call Statement and Input:

```
CALL SPOFCD('L', A, 9, 9, 3, RCOND, DET, AUX, 9 )
```

Output:

\[
A = (\text{same as input } A \text{ in Example 1})
\]

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 2 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 $\| A \|_1 = 4.0$

On output, the value in $det, |A|$, is equal to 1.

Call Statement and Input:

```
CALL SPOFCD('U', A, 9, 9, 3, RCOND, DET, AUX, 9 )
```

Output:

\[
A = (\text{same as output } A \text{ in Example 2})
\]

RCOND = 0.0055555

\[
DET = (1.0, 0.0)
\]
SPOTRI, DPOTRI, CPOTRI, ZPOTRI, SPOICD, DPOICD, SPPTRI, DPPTRI, CPPTRI, ZPPTRI, SPPICD, and DPPICD (Positive Definite Real Symmetric or Complex Hermitian Matrix Inverse, Condition Number Reciprocal, and Determinant)

Purpose

These subroutines find the inverse of a positive definite real symmetric or complex Hermitian matrix $A$ using Cholesky factorization, where:

- For SPOTRI, DPOTRI, CPOTRI, ZPOTRI, SPOICD, and DPOICD, $A$ is stored in upper or lower storage mode.
- For SPPTRI, DPPTRI, CPPTRI, and ZPPTRI, $A$ is stored in upper- or lower-packed storage mode.
- For SPPICD and DPPICD, $A$ is stored in lower-packed storage mode.

Subroutines SPOICD, DPOICD, SPPICD, and DPPICD also find the reciprocal of the condition number and the determinant of matrix $A$.

Table 148. Data Types

<table>
<thead>
<tr>
<th>$A$, aux, rcond, det</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SPOTRI, SPOICD, SPPTRI, and SPPICD</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DPOTRI, DPOICD, DPPTRI, and DPPICD</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CPOTRI and CPPTRI</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZPOTRI and ZPPTRI</td>
</tr>
</tbody>
</table>

Note: For each of the _POTRI and _PPTRI subroutines, the input must be the output from the corresponding _POTRF or _PPTRF Cholesky factorization subroutine.

If you call the subroutines SPOICD, DPOICD, SPPICD, and DPPICD with $iopt = 4$, the input must be the output from SPPEF, DPPEF, SPOF/SPOFCD, or DPOF/DPOFCD, respectively, where Cholesky factorization was performed.

Syntax

**Fortran**

CALL SPOTRI | DPOTRI | CPOTRI | ZPOTRI (uplo, n, a, lda, info)  
CALL SPOICD | DPOICD (uplo, a, lda, n, iopt, rcond, det, aux, naux)  
CALL SPPTRI | DPPTRI | CPPTRI | ZPPTRI (uplo, n, ap, info)  
CALL SPPICD | DPPICD (ap, n, iopt, rcond, det, aux, naux)

**C and C++**

spotri | dpotri | cpotri | zpotri (uplo, n, a, lda, info);  
spoicd | dpoicd (uplo, a, lda, n, iopt, rcond, det, aux, naux);  
spptri | dpptri | cpptri | zpptri (uplo, n, ap, info);  
sppicd | dppicd (ap, n, iopt, rcond, det, aux, naux);

**LAPACK**

info = LAPACKE_spotri | LAPACKE_dpotri | LAPACKE_cpotri | LAPACKE_zpotri (matrix_layout, uplo, n, a, lda);  
info = LAPACKE_spptri | LAPACKE_dpptri | LAPACKE_cpptri | LAPACKE_zpptri (matrix_layout, uplo, n, ap);

On Entry

626  ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
**matrix_layout**
indicates whether the input and output matrices are stored in row major order or column major order, where:

- If `matrix_layout` = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If `matrix_layout` = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. It must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR

**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:

For SPPTRI, DPPTRI, CPPTRI, and ZPPTRI:

AP contains the transformed matrix A of order n, resulting from the Cholesky factorization performed in a previous call to SPPTRF, DPPTRF, CPPTRF, or ZPPTRF, respectively, whose inverse is computed.

For SPPICD and DPPICD:

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 DPPE, 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 148 on page 626**

**a** has the following meaning, where:

For SPOTRI, DPOTRI, CPOTRI, and ZPOTRI:

It is the transformed matrix A of order n, containing results of the factorization from a previous call to SPOTRF, DPOTRF, CPOTRF, or ZPOTRF, respectively, whose inverse is computed.

For SPOICD and DPOICD:

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 `lda` by (at least) n array, containing numbers of the data type indicated in **Table 148 on page 626**

**lda** is the leading dimension of the array specified for `a`. 
Specified as: an integer; \( \text{lda} > 0 \) and \( \text{lda} \geq n \).

\( n \) is the order \( n \) of matrix \( A \).
Specified as: an integer; \( n \geq 0 \).

\( \text{iOpt} \)
indicates the type of computation to be performed, where:
If \( \text{iOpt} = 0 \), the inverse is computed for matrix \( A \).
If \( \text{iOpt} = 1 \), the inverse and the reciprocal of the condition number are computed for matrix \( A \).
If \( \text{iOpt} = 2 \), the inverse and the determinant are computed for matrix \( A \).
If \( \text{iOpt} = 3 \), the inverse, the reciprocal of the condition number, and the determinant are computed for matrix \( A \).
If \( \text{iOpt} = 4 \), the inverse is computed for the Cholesky factored matrix \( A \).
Specified as: an integer; \( \text{iOpt} = 0, 1, 2, 3, \) or \( 4 \).

\( \text{rcond} \)
See [On Return](#).

\( \text{det} \)
See [On Return](#).

\( \text{aux} \)
has the following meaning:
If \( \text{naux} = 0 \) and error 2015 is unrecoverable, \( \text{aux} \) is ignored.
Otherwise, it is the storage work area used by this subroutine. Its size is specified by \( \text{naux} \). Specified as: an area of storage, containing numbers of the data type indicated in [Table 148 on page 626](#).

\( \text{naux} \)
is the size of the work area specified by \( \text{aux} \)—that is, the number of elements in \( \text{aux} \).
Specified as: an integer, where:
If \( \text{naux} = 0 \) and error 2015 is unrecoverable, SPOICD, DPOICD, SPPICD, and DPPICD dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.
Otherwise, \( \text{naux} \geq n \).

\( \text{info} \)
See [On Return](#).

**On Return**

\( \text{ap} \)
is an array, referred to as \( \text{AP} \), in which the transformed matrix \( A \) of order \( n \), containing the inverse of the matrix, is stored.

**For SPPTRI, DPPTRI, CPPTRI, and ZPPTRI:**
The transformed matrix is stored in upper- or lower-packed storage mode.

**For SPPICD and DPPICD:**
The transformed matrix is stored 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 148 on page 626](#).

\( \text{a} \)
is the transformed matrix \( A \) 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 148 on page 626.

\( \text{rcond} \)

is the reciprocal of the condition number, \( \text{rcond} \), of matrix \( A \). Returned as: a real number of the data type indicated in Table 148 on page 626; \( \text{rcond} \geq 0 \).

\( \text{det} \)

is the vector \( \text{det} \), containing the two components \( \text{det}_1 \) and \( \text{det}_2 \) of the determinant of matrix \( A \). The determinant is:

\[
\text{det}_1 \left(10^{\text{det}_2}\right)
\]

where \( 1 \leq \text{det}_1 < 10 \). Returned as: an array of length 2, containing numbers of the data type indicated in Table 148 on page 626.

\( \text{info} \)

has the following meaning:

If \( \text{info} = 0 \), the inverse completed successfully.

If \( \text{info} > 0 \), \( \text{info} \) is set equal to the first \( i \) where \( A_{ii} \) is zero; the matrix is not positive definite, and its inverse could not be completed.

Returned as:

- For SPOTRI, DPOTRI, CPOTRI, ZPOTRI, SPOICD, DPOICD, SPPTRI, DPPTRI, CPPTRI, ZPPTRI, SPPICD, and DPPICD, returned as: an integer; \( \text{info} \geq 0 \).
- For LAPACKE_LAPACKE_spotri, LAPACKE_dpotri, LAPACKE_cpotri, LAPACKE_zpotri, LAPACKE_spptri, LAPACKE_dpptri, LAPACKE_cpptri, and LAPACKE_zppti, returned as an integer function value; \( \text{info} \geq 0 \).

Specified as: an integer; \( \text{info} \geq 0 \).

Notes

1. In your C program, the arguments \( \text{info} \) and \( \text{rcond} \) must be passed by reference.
2. For SPOICD, DPOICD, SPPICD, and DPPICD, when you specify \( iopt = 4 \), you must do the following:
   - For SPOICD and DPOICD, specify the same storage mode for matrix \( A \) that was specified in the previous call to SPOF/SPOFCD or DPOF/DPOFCD, respectively.
   - For SPPICD and DPPICD, use Cholesky factorization in the previous call to SPPF or DPPF, respectively.
3. The scalar data specified for input arguments \( \text{uplo} \), \( \text{lda} \), and \( n \) for these subroutines must be the same as the input arguments specified for the corresponding factorization subroutines.
4. All subroutines accept lowercase letters for the \( \text{uplo} \) argument.
5. SPPICD and DPPICD in some cases utilize algorithms based on recursive packed storage format. (See references [61 on page 1367], [77 on page 1368], and [79 on page 1368]).
6. The way _POTRI and _PPTRI subroutines handle computational errors differs from LAPACK. Like LAPACK, these subroutines use the \( \text{info} \) argument to provide information about the computational error, but they also provide an error message.
7. On both input and output, matrix \( A \) conforms to LAPACK format.
8. For a description of how a positive definite symmetric matrix is stored in upper- or lower-packed storage mode in an array or in upper or lower storage mode, see “Symmetric Matrix” on page 85.

9. 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 51.

Function

These subroutines find the inverse of positive definite matrix A, where:

- \( A^{-1} \) is the inverse of matrix A, where \( A A^{-1} = A^{-1} A = I \).
- For positive definite real symmetric matrix A:
  \[
  A = LL^T \quad \text{or} \quad U^T U
  \]
  \[
  A^{-1} = L^{-T} L^{-1} \quad \text{or} \quad U^{-1} U^T
  \]
- For positive definite complex Hermitian matrix A:
  \[
  A = LL^H \quad \text{or} \quad U^H U
  \]
  \[
  A^{-1} = L^{-H} L^{-1} \quad \text{or} \quad U^{-1} U^H
  \]

Note: SPPICD and DPPICD only support a matrix in lower-packed storage mode.

Additionally, the subroutines SPOICD, DPOICD, SPPICD, and DPPICD find the reciprocal of the condition number and the determinant of positive definite symmetric matrix A using Cholesky factorization, where:

- \( 1/\| A \|_1 (\| A^{-1} \|_1) \) is the reciprocal of the condition number, where \( \| A \|_1 \) is the one-norm of matrix A.
- \( | A | \) is the determinant of matrix A, where |A| is expressed as:
  \[
  \text{det}(10^{\text{det}_2})
  \]

- The iopt argument is used to determine the combination of output items produced: the inverse, the reciprocal of the condition number, and the determinant.

If \( n \) is 0, no computation is performed. See references \[44 on page 1366\], \[46 on page 1366\], and \[52 on page 1366\].

Error conditions

Resource Errors
- Unable to allocate internal work area.
- Error 2015 is unrecoverable, naux = 0, and unable to allocate work area.

Computational Errors

Note: If the Cholesky factorization performed by one of the _POTRF and _PPTRF subroutines failed because matrix A was not positive definite, the results returned by the corresponding _POTRI or _PPTRI subroutine are unpredictable.

If the Cholesky factorization performed by SPPF, DPPF, SPOF/SPOFCD, or DPOF/DPOFCD failed because matrix A was not positive definite, the results returned by SPOICD, DPOICD, SPPICD, or DPPICD, respectively, with iopt = 4, are unpredictable.
For SPOTRI, DPOTRI, CPOTRI, ZPOTRI, SPPTRI, DPPTRI, CPPTRI, and ZPPTRI:

The inverse of matrix $A$ could not be computed.
• One or more of the diagonal elements of the factored matrix $A$ are zero. The first diagonal element that is found to be exactly zero is identified in the computational error message and returned in info. If one or more of the diagonal elements of the factored matrix $A$ are negative, the results are unpredictable.
• 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 2151 is set to be unlimited in the ESSL error option table.

For SPOICD, DPOICD, SPPICD, and DPPICD:

Matrix $A$ 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.
• $i$ 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 2150 in the error option table; otherwise, the default value causes your program to terminate. If your program is not terminated, the return code is set to 3. For details, see “What Can You Do about ESSL Computational Errors?” on page 68.

The inverse of matrix $A$ could not be computed.
• For $iopt = 4$, for _POICD and _PPICD, one or more of the diagonal elements of the factored matrix $A$ are zero. $i$ is the first diagonal element that is found to be exactly zero and is identified in the computational error message. If one or more of the diagonal elements of the factored matrix $A$ are negative, the results are unpredictable.
• $i$ 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 2150 in the error option table; otherwise, the default value causes your program to terminate. If your program is not terminated, the return code is set to 3. For details, see “What Can You Do about ESSL Computational Errors?” on page 68.

Input-Argument Errors
1. $uplo \neq 'U' \text{ or } 'L'$
2. $n < 0$
3. $lda \leq 0$
4. $lda < n$
5. $iopt \neq 0, 1, 2, 3, \text{ or } 4$
6. 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.

Examples

Example 1

This example uses SPOTRI to compute the inverse of matrix $A$, where $iopt = 4$, and matrix $A$ is the transformed matrix factored by SPOTRF in [Example 9].

Call Statement and Input:
Example 2

This example uses CPOTRI to compute the inverse of the matrix $A$, stored in lower storage mode. Matrix $A$ is the transformed matrix factored by CPOTRF in Example 3.

Call Statement and Input:

```
UPLO N A LDA INFO |
| | | | |
CALL CPOTRI( 'L', 3, A, 3, INFO )
```

$A = \text{(same as output $A$ in Example 3 for CPOTRF)}$

Output:

```
A = \begin{bmatrix}
0.05 & 0.00 \\
0.00 & -0.01 & 0.02 & 0.00 \\
-0.01 & 0.00 & 0.00 & 0.00 \\
\end{bmatrix}
```

INFO = 0

Example 3

This example uses CPOTRI to compute the inverse of the matrix $A$, stored in upper storage mode. Matrix $A$ is the transformed matrix factored by CPOTRF in Example 4.

Call Statement and Input:

```
UPLO N A LDA INFO |
| | | | |
CALL CPOTRI( 'U', 3, A, 3, INFO )
```

$A = \text{(same as output $A$ in Example 4 for CPOTRF)}$

Output:

```
A = \begin{bmatrix}
0.13 & 0.00 & -0.02 & -0.03 & 0.00 & 0.01 \\
0.07 & 0.00 & -0.01 & 0.01 \\
\end{bmatrix}
```

INFO = 0

Example 4
This example uses SPOICD to compute the inverse, reciprocal of the condition number, and determinant of matrix $A$, stored in upper storage mode. Matrix $A$ is:

$$
A = \begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 \\
1.0 & 2.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 4.0 & 4.0 & 4.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 5.0 & 5.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 6.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 8.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 9.0 \\
\end{bmatrix}
$$

The values used to compute the reciprocal of the condition number in this example are obtained with the following values:

$$
\| A \|_1 = \max(9.0, 17.0, 24.0, 30.0, 35.0, 39.0, 42.0, 44.0, 45.0) = 45.0 \\
\| A^{-1} \|_1 = 4.0
$$

On output, the value in $\det, |A|$, is equal to 1, and $RCOND = 1/180$.

Call Statement and Input:

```fortran
CALL SPOICD('U', A, 9, 9, 3, RCOND, DET, AUX, NAUX)
```

Output:

```fortran
A = \begin{bmatrix}
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 \\
. .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 & 6.0 \\
. . . .7.0 & 7.0 & 7.0 & 7.0 & 7.0 \\
. . . .8.0 & 8.0 & 8.0 & 8.0 & 8.0 \\
. . . .9.0 & 9.0 & 9.0 & 9.0 & 9.0 \\
\end{bmatrix}
```

```fortran
RCOND = 0.005555556 \\
DET = (1.0, 0.0)
```

**Example 5**

This example uses SPPTRI to compute the inverse of matrix $A$, where matrix $A$ is the transformed matrix factored in Example 5 by SPPTRF.
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:

\[
\begin{array}{c|c|c|c}
UPLO & N & AP & INFO \\
\hline
\end{array}
\]

CALL SPPTRI( 'L', 9, AP, INFO )

\[
AP = \text{(same as output AP in Example 5 for SPPTRF)}
\]

Output:

\[
AP = (2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
1.0)
\]

INFO = 0

Example 6

This example uses SPPTRI to compute the inverse of matrix A, where matrix A is the transformed matrix factored in Example 6 by SPPTRF.

Note: The AP arrays are formatted in a triangular arrangement for readability; however, they are stored in upper-packed storage mode.

Call Statement and Input:

\[
\begin{array}{c|c|c|c}
UPLO & N & AP & INFO \\
\hline
\end{array}
\]

CALL SPPTRI( 'U', 9, AP, INFO )

\[
AP = \text{(same as output AP in Example 6 for SPPTRF)}
\]

Output:

\[
AP = (2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
-1.0, 2.0, 0.0, -1.0, 2.0, 0.0, -1.0, 2.0, 0.0, -1.0, 2.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
1.0, 0.0)
\]

INFO = 0

Example 7

This example uses ZPPTRI to compute the inverse of matrix A, where matrix A, stored in lower-packed storage mode, is the transformed matrix factored in Example 7 by ZPPTRF.

Call Statement and Input:

\[
\begin{array}{c|c|c|c}
UPLO & N & AP & INFO \\
\hline
\end{array}
\]

CALL ZPPTRI( 'L', 3, AP, INFO )
Example 8

This example uses ZPPTRI to compute the inverse of matrix \( A \), where matrix \( A \), stored in upper-packed storage mode, is the transformed matrix factored in Example 8 by ZPPTRF.

Call Statement and Input:

```
UPLO  N  AP  INFO
```

```
CALL ZPPTRI('U', 3, AP, INFO)
```

Output:

\[
AP = ( (0.05, 0.00) (0.00, -0.01) (-0.01, 0.00) (0.02, 0.00) (0.00, 0.00) (0.02, 0.00) )
\]

INFO = 0

Example 9

This example uses SPPICD to compute the inverse, reciprocal of the condition number, and determinant of the same matrix \( A \) used in Example 4 however, matrix \( A \) is stored in lower-packed 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:

\[
\| A \|_1 = \max(9.0, 17.0, 24.0, 30.0, 35.0, 39.0, 42.0, 44.0, 45.0) = 45.0 \\
\| A^{-1} \|_1 = 4.0
\]

On output, the value in \( det, |A| \), 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 )
```

\[
AP = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 4.0, 4.0, 4.0, 4.0, 4.0, 5.0, 5.0, 5.0, 5.0, 6.0, 6.0, 6.0, 7.0, 7.0, 7.0, 8.0, 8.0, 9.0)
\]

Output:
Example 10

This example uses SPPICD to compute the inverse of matrix $A$, where $iopt = 4$, and matrix $A$ is the transformed matrix factored in Example 10 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:

```call
CALL SPPICD(AP, 9, 4, RCOND, DET, AUX, NAUX)
```

Output:

```
AP = (2.0, -1.0, 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, 0.0,
     2.0, -1.0, 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, 0.0,
     2.0, -1.0, 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, 0.0,
     2.0, -1.0, 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, 0.0,
     2.0, -1.0, 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, 0.0,

RCOND = 0.005556
DET = (1.0, 0.0)
```
Purpose

These subprograms compute the norm of matrix $A$ as explained below:

**SLANSY, DLANSY, CLANHE, and ZLANHE**

These subprograms compute the norm of matrix $A$, stored in upper or lower storage mode, where:
- For SLANSY and DLANSY, $A$ is a positive definite real symmetric matrix.
- For CLANHE and ZLANHE, $A$ is a positive definite complex Hermitian matrix.

**SLANSP, DLANSP, CLANHP, and ZLANHP**

These subroutines compute the norm of matrix $A$, stored in upper-packed or lower-packed storage mode, where:
- For SLANSP and DLANSP, $A$ is a positive definite real symmetric matrix.
- For CLANHP and ZLANHP, $A$ is a positive definite complex Hermitian matrix.

Table 149. Data Types

<table>
<thead>
<tr>
<th>$A$</th>
<th>work, Result</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SLANSY, SLANSP</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DLANSY, DLANSP</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>CLANHE, CLANHP</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>ZLANHE, ZLANHP</td>
</tr>
</tbody>
</table>

Syntax

Fortran

SLANSY | DLANSY | CLANHE | ZLANHE(norm, uplo, n, a, lda, work)
SLANSP | DLANSP | CLANHP | ZLANHP (norm, uplo, n, ap, work)

C and C++

slansy | dlansy | clanhe | zlanhe (norm, uplo, n, a, lda, work);
slanp | dlanp | clanhp | zlanhp (norm, uplo, n, ap, work);

LAPACK

LAPACKE_slansy | LAPACKE_dlansy | LAPACKE_clanhe | LAPACKE_zlanhe (matrix_layout, norm, uplo, n, a, lda);
LAPACKE_slanp | LAPACKE_dlanp | LAPACKE_clanhp | LAPACKE_zlanhp (matrix_layout, norm, uplo, n, ap);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:
- If matrix_layout = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If matrix_layout = LAPACK_COL_MAJOR, the matrices are stored in column major order.
Specified as: an integer. It must be LAPACK_ROW_MAJOR or
LAPACK_COL_MAJOR

**norm**
specifies the type of computation, where:
- If `norm = 'O'` or `'1'`, the one norm of `A` is computed.
- If `norm = 'T'`, the infinity norm of `A` is computed.
- If `norm = 'F'` or `'E'`, the Frobenius or Euclidean norm of `A` is computed.
- If `norm = 'M'`, the absolute value of the matrix element having the largest absolute value, i.e., `max (|A|)`, is returned.

Specified as: a single character; `norm = 'O', '1', 'T', 'F', 'E', or 'M'.`

**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'.

**n**
is the order of matrix `A`.

Specified as: an integer; `n ≥ 0`.

**ap** is the matrix `A`, stored in upper-packed or lower-packed storage mode.

Specified as: a one-dimensional array, containing numbers of the data type indicated in [Table 149 on page 637](#).

**a** is the matrix `A`, 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 149 on page 637](#).

**lda**
is the leading dimension of matrix `A`.

Specified as: an integer; `lda > 0` and `lda ≥ n`.

**work**
is the work area used by this subroutine, where:
- When `norm = 'O', '1', or 'T'`, the size of `work` is (at least) of length `n`.
- Otherwise, `work` is not referenced.

Specified as: an area of storage containing numbers of data type indicated in [Table 149 on page 637](#).

**On Return**

**Function value**
is the result of the `norm` computation, returned as a number of the data type indicated in [Table 149 on page 637](#).

- If `norm = 'O'` or `'1'`, the one norm of `A` is returned.
- If `norm = 'T'`, the infinity norm of `A` is returned.
- If `norm = 'F'` or `'E'`, the Frobenius or Euclidean norm of `A` is returned.
- If `norm = 'M'`, the absolute value of the matrix element having the largest absolute value, i.e., `max (|A|)`, is returned.
- If `n = 0`, the function returns zero.
Notes
1. Declare this function in your program as returning a value of the data type indicated in Table 149 on page 637.
2. This function accepts lowercase letters for the norm and uplo arguments.
3. For real symmetric and complex Hermitian matrices, the one norm and the infinity norm are identical.

Function

One of the following computations is performed on matrix $A$, depending on the value specified for norm:

<table>
<thead>
<tr>
<th>Value specified for norm</th>
<th>Type of computation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>'O' or '1'</td>
<td>one norm</td>
</tr>
<tr>
<td>'T'</td>
<td>infinity norm</td>
</tr>
<tr>
<td>'F' or 'E'</td>
<td>Frobenius or Euclidean norm</td>
</tr>
<tr>
<td>'M'</td>
<td>absolute value of the matrix element having the largest absolute value, i.e., $\max</td>
</tr>
</tbody>
</table>

If $n = 0$, the function returns zero.

Error conditions
Resource Errors
None.
Computational Errors
None.
Input-Argument Errors
1. $\text{norm} \neq \text{'O', '1', 'T', 'F', 'E', or 'M'}$
2. $\text{uplo} \neq \text{'U' or 'L'}$
3. $n < 0$
4. $n > \text{lda}$
5. $\text{lda} \leq 0$

Examples
Example 1
This example computes the one norm of positive definite real symmetric matrix $A$ stored in lower storage mode.

Call Statements and Input:

Call Statements and Input:

```
NORM UPLO N A LDA WORK
```

```
ANORM = DLANSY( '1', 'L', 9, A, 9, WORK )
```

```
A = (same as input matrix A in Example 1)
```

Output:

```
ANORM = 45.0
```

Example 2
This example computes the one norm of positive definite real symmetric matrix $A$ stored in upper storage mode.
Call Statements and Input:

\[
\begin{align*}
&\text{NORM UPLO N A LDA WORK} \\
&\text{ANORM = DLANSY( '1', 'U', 9, A, 9, WORK )} \\
&A = \text{(same as input matrix A in Example 2)}
\end{align*}
\]

Output:

\[
\text{ANORM = 45.0}
\]

**Example 3**

This example computes the one norm of positive definite complex Hermitian matrix \(A\) stored in lower storage mode.

Call Statements and Input:

\[
\begin{align*}
&\text{NORM UPLO N A LDA WORK} \\
&\text{ANORM = ZLANHE( '1', 'L', 3, A, 3, WORK )} \\
&A = \text{(same as input matrix A in Example 3)}
\end{align*}
\]

Output:

\[
\text{ANORM = 89.39}
\]

**Example 4**

This example computes the one norm of positive definite complex Hermitian matrix \(A\) stored in upper storage mode.

Call Statements and Input:

\[
\begin{align*}
&\text{NORM UPLO N A LDA WORK} \\
&\text{ANORM = ZLANHE( '1', 'U', 3, A, 3, WORK )} \\
&A = \text{(same as input matrix A in Example 4)}
\end{align*}
\]

Output:

\[
\text{ANORM = 57.24}
\]

**Example 5**

This example computes the one norm of positive definite real symmetric matrix \(A\) stored in lower-packed storage mode.

Call Statements and Input:

\[
\begin{align*}
&\text{NORM UPLO N AP WORK} \\
&\text{ANORM = DLANS( '1', 'L', 9, AP, WORK )} \\
&\text{AP = (same as input matrix AP in Example 5)}
\end{align*}
\]

Output:

\[
\text{ANORM = 45.0}
\]

**Example 6**

This example computes the one norm of positive definite real symmetric matrix \(A\) stored in upper-packed storage mode.

Call Statements and Input:
Example 7

This example computes the one norm of positive definite complex Hermitian matrix $A$ stored in lower-packed storage mode.

Call Statements and Input:

```
NORM    UPLO    N       AP       WORK
ANORM = DLANSP( '1', 'U', 9, AP, WORK )
```

$AP$ = (same as input matrix $AP$ in Example 6)

Output:

$ANORM = 45.0$

Example 8

This example computes the one norm of positive definite complex Hermitian matrix $A$ stored in upper-packed storage mode.

Call Statements and Input:

```
NORM    UPLO    N       AP       WORK
ANORM = ZLANHP( '1', 'L', 3, AP, WORK )
```

$AP$ = (same as input matrix $AP$ in Example 7)

Output:

$ANORM = 89.39$

Example 7

This example computes the one norm of positive definite complex Hermitian matrix $A$ stored in upper-packed storage mode.

Call Statements and Input:

```
NORM    UPLO    N       AP       WORK
ANORM = ZLANHP( '1', 'U', 3, AP, WORK )
```

$AP$ = (same as input matrix $AP$ in Example 8)

Output:

$ANORM = 57.24$
SSYSV, DSYSV, CSYSV, ZSYSV, CHESV, ZHESV, SSPSV, DSPSV, CSPSV, ZSPSV, CHPSV, and ZHPSV (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)

Purpose

These subroutines solve the system \( AX = B \) for \( X \), where \( A \) is an indefinite real or complex symmetric or complex Hermitian matrix and \( X \) and \( B \) are general matrices.

Table 150. Data Types

<table>
<thead>
<tr>
<th>( A, B, \text{work} )</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SSYSV(^A), SSPSV(^A)</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DSYSV(^A), DSPSV(^A)</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CSYSV(^A), CHESV(^A), CSPSV(^A), CHPSV(^A)</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZSYSV(^A), ZHESV(^A), ZSPSV(^A), ZHPSV(^A)</td>
</tr>
</tbody>
</table>

\(^A\) LAPACK

Syntax

Fortran

```fortran
CALL SSYSV | DSYSV | CSYSV | ZSYSV | CHESV | ZHESV (uplo, n, nrhs, a, lda, ipiv, b, ldb, work, hwork, info)
CALL SSPSV | DSPSV | CSPSV | ZSPSV | CHPSV | ZHPSV (uplo, n, nrhs, a, lda, ipiv, b, ldb, info)
```

C and C++

```c
ssysv | dsysv | csysv | zsysv | chesv | zhess (uplo, n, nrhs, a, lda, ipiv, b, ldb, work, hwork, info);
sspsv | dspsv | cspsv | zspsv | chpsv | zhpsv (uplo, n, nrhs, a, lda, ipiv, b, ldb, info);
```

LAPACK

```c
info = LAPACK_ssysv | LAPACK_dsysv | LAPACK_csysv | LAPACK_zsysv | LAPACK_chesv | LAPACK_zhesv (matrix_layout, uplo, n, nrhs, a, lda, ipiv, b, ldb);
info = LAPACK_sspsv | LAPACK_dspsv | LAPACK_cspsv | LAPACK_zspsv | LAPACK_chpsv | LAPACK_zhpsv (matrix_layout, uplo, n, nrhs, a, lda, ipiv, b, ldb);
```

On Entry

- **matrix_layout** indicates whether the input and output matrices are stored in row major order or column major order, where:
  - If `matrix_layout = LAPACK_ROW_MAJOR`, the matrices are stored in row major order.
  - If `matrix_layout = LAPACK_COL_MAJOR`, the matrices are stored in column major order.

  Specified as: an integer. It must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR

- **uplo** indicates whether the upper or lower triangular part of the matrix \( A \) is referenced, where:
  - If `uplo = 'U'`, the upper triangular part is referenced.
  - If `uplo = 'L'`, the lower triangular part is referenced.

  Specified as: a single character; `uplo = 'U'` or 'L'.

- **n** is the order of matrix \( A \) and the number of rows in matrix \( B \).

  Specified as: an integer; \( n \geq 0 \).

- **nrhs** is the number of right-hand sides; that is, the number of columns in matrix \( B \).
Specified as: an integer; \( nrhs \geq 0 \).

\( a \) is the indefinite real symmetric, complex symmetric, or complex Hermitian matrix \( A \) of order \( n \).

If \( uplo = 'U' \), it is stored in upper storage mode.

If \( uplo = 'L' \), 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 150 on page 642

\( ap \) is the indefinite real symmetric, complex symmetric, or complex Hermitian matrix \( A \) of order \( n \). It is stored in an array, referred to as \( AP \), where:

If \( uplo = 'U' \), it is stored in upper-packed storage mode.

If \( uplo = 'L' \), it is stored in lower-packed storage mode.

Specified as: one-dimensional array of (at least) length \( n(n + 1)/2 \), containing numbers of the data type indicated in Table 150 on page 642

\( lda \)

is the leading dimension of the array specified for \( A \).

Specified as: an integer; \( lda > 0 \) and \( lda \geq n \).

\( ipiv \)

See On Return

\( b \) is the general 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 \( nrhs \) array, containing numbers of the data type indicated in Table 150 on page 642

\( ldb \)

is the leading dimension of the array specified for \( B \).

Specified as: an integer; \( ldb > 0 \) and \( ldb \geq n \).

\( work \)

is a work area used by these subroutines, where:

If \( lwork = 0 \), \( work \) is ignored.

If \( lwork \neq 0 \), the size of \( work \) is determined as follows:

- If \( lwork \neq -1 \), \( work \) is (at least) of length \( lwork \).
- If \( lwork = -1 \), \( work \) is (at least) of length 1.

Specified as: an area of storage containing numbers of the data type indicated in Table 150 on page 642

\( lwork \)

is the number of elements in array \( WORK \).

Specified as: an integer; where:

- If \( lwork = 0 \), the subroutine dynamically allocates the workspace needed for use during this computation. The work area is deallocated before control is returned to the calling program.
- If \( lwork = -1 \), subroutine performs a workspace query and returns the optimal required size of \( work \) in \( work \). No computation is performed and the subroutine returns after error checking is complete.
- Otherwise, \( lwork \geq 1 \). It is suggested that the user specify \( lwork \geq 8n \).
**info**

See **On Return**.

**On Return**

**a** is the transformed matrix $A$ containing the results of the factorization. See “Function” on page 654.

Returned as: an $lda$ by (at least) $n$ array, containing numbers of the data type indicated in Table 150 on page 642.

**ap** is the transformed matrix $A$ containing the results of the factorization. See “Function” on page 654.

Returned as: a one-dimensional array of (at least) length $n(n + 1)/2$, containing numbers of the data type indicated in Table 150 on page 642.

**ipiv**

If $info = 0$, $ipiv$ contains the pivot indices.

If $ipiv_k > 0$, then rows and columns $k$ and $ipiv_k$ were interchanged and $D_{k,k}$ is a $1 \times 1$ diagonal block.

If $uplo = 'U'$ and $ipiv_k = ipiv_{k+1} < 0$, then rows and columns $k-1$ and -$ipiv_k$ were interchanged and $D_{k+1,k-1}$ is a $2 \times 2$ diagonal block.

If $uplo = 'L'$ and $ipiv_k = ipiv_{k+1} < 0$, then rows and columns $k+1$ and -$ipiv_k$ were interchanged and $D_{k+1,k}$ is a $2 \times 2$ diagonal block.

Returned as: a one-dimensional integer array of (at least) length $n$, containing integers.

**b** If $info = 0$, $b$ is the matrix $X$, containing the $nrhs$ solutions to the system. The solutions, each of length $n$, reside in the columns of $X$.

Returned as: an $ldb$ by (at least) $nrhs$ array, containing numbers of the data type indicated in Table 150 on page 642.

**work**

is a work area used by this subroutine if $lwork \neq 0$, where:

If $lwork \neq 0$ and $lwork \neq -1$, its size is (at least) of length $lwork$.

If $lwork = -1$, its size is (at least) of length 1.

Returned as: an area of storage, where:

If $lwork \geq 1$ or $lwork = -1$, then $work_1$ is set to the optimal $lwork$ value and all other elements of $work$ are overwritten.

**info**

has the following meaning:

- If $info = 0$, the factorization completed successfully.
- If $info > 0$, the factorization was unsuccessful and $info$ is set to $i$ where $d_{ii}$ is exactly zero.

Returned as:

- For SSYSV, DSYSV, CSYSV, ZSYSV, CHESV, ZHESV, SSPSV, DSPSV, CPSV, ZSPSV, CHPSV, and ZHPSV, returned as: an integer; $info \geq 0$.
- For LAPACKE_sspsv, LAPACKE_dspsv, LAPACKE_csysv, LAPACKE_zsysv, LAPACKE_chesv, LAPACKE_zhesv, LAPACKE_sspsv, LAPACKE_dspsv, LAPACKE_cpsv, LAPACKE_zpsv, LAPACKE_chpsv, and LAPACKE_zhpsv, returned as an integer function value; $info \geq 0$. 644 ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
Notes
1. These subroutines accept lowercase letters for the *uplo* argument.
2. In your C program, argument *info* must be passed by reference.
3. *a, ap, b, ipiv,* and *work* must have no common elements; otherwise, results are unpredictable.
4. For a description of how real and complex symmetric matrices are stored in lower or upper storage mode, see "Lower Storage Mode" on page 88 or "Upper Storage Mode" on page 89, respectively.
5. For a description of how complex Hermitian matrices are stored in lower or upper storage mode, see "Complex Hermitian Matrix" on page 90.
6. For a description of how real and complex symmetric matrices are stored in lower- or upper-packed storage mode, see "Lower-Packed Storage Mode" on page 85 or "Upper-Packed Storage Mode" on page 87, respectively.
7. For a description of how complex Hermitian matrices are stored in lower- or upper-packed storage mode, see "Complex Hermitian Matrix" on page 90.
6. For best performance, specify *lwork* = 0.

Function
These subroutines solve the system $AX = B$ for $X$, where $A$ is an indefinite real or complex symmetric or complex Hermitian indefinite matrix and $X$ and $B$ are general matrices.

For SSYSV, DSYSV, CSYSV, ZSYSV, SSPSV, DSPSV, CSPSV, and ZSPSV:

The indefinite real or complex symmetric indefinite matrix $A$ is factored using the Bunch-Kaufman diagonal pivoting method, where $A$ is expressed as one of the following:

$$ A = UDU^T $$

$$ A = LDL^T $$

where:

$U$ is a product of permutation and unit upper triangular matrices.

$L$ is a product of permutation and unit lower triangular matrices.

$D$ is a symmetric block diagonal matrix, consisting of $1 \times 1$ and $2 \times 2$ diagonal blocks.

Matrix $A$ is stored as follows:
- For SSYSV, DSYSV, CSYSV, and ZSYSV, matrix $A$ is stored in upper or lower storage mode.
- For SSPSV, DSPSV, CSPSV, and ZSPSV, matrix $A$ is stored in upper- or lower-packed storage mode.

For CHESV, ZHESV, CHPSV, and ZHPSV:
The indefinite complex Hermitian indefinite matrix $A$ is factored using the Bunch-Kaufman diagonal pivoting method, where $A$ is expressed as one of the following:

\[ A = UDU^H \]
\[ A = LDL^H \]

where:

$U$ is a product of permutation and unit upper triangular matrices.

$L$ is a product of permutation and unit lower triangular matrices.

$D$ is a complex Hermitian block diagonal matrix, consisting of $1 \times 1$ and $2 \times 2$ diagonal blocks.

Matrix $A$ is stored as follows:

- For CHESV and ZHESV, matrix $A$ is stored in upper or lower storage mode.
- For CHPSV and ZHPSV, matrix $A$ is stored in upper- or lower-packed storage mode.

If $n$ is 0, no computation is performed and the subroutine returns after doing some parameter checking. If $n > 0$ and $nrhs$ is 0, no solutions are computed and the subroutine returns after factoring the matrix. See references [8 on page 1363] and [18 on page 1364].

**Error conditions**

**Resource Errors**

\[ \text{lwork} = 0 \text{ and unable to allocate work area} \]

**Computational Errors**

Matrix $A$ is singular.

- The factorization completed but the block diagonal matrix $D$ is exactly singular. $info$ is set to $i$, where $d_{ii}$ is exactly zero. This 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 2147 is set to be unlimited in the ESSL error option table. For details, see **“What Can You Do about ESSL Computational Errors?” on page 68**.

**Input-Argument Errors**

1. $\text{uplo} \neq \text{'U'}$ or $\text{'L'}$
2. $n < 0$
3. $nrhs < 0$
4. $\text{lda} \leq 0$
5. $n > \text{lda}$
6. $\text{ldb} \leq 0$
7. $n > \text{ldb}$
8. $\text{lwork} \neq 0$ and $\text{lwork} \neq -1$ and $\text{lwork} <$ the minimum required value

**Examples**

**Example 1**
This example shows how to solve the system $AX = B$, for three right-hand sides, where indefinite real symmetric matrix $A$ is the same matrix factored in the Example 1 for DSYTRF.

Note: Because $lwork = 0$, the subroutine dynamically allocates $WORK$.

Call Statement and Input:

```
UPLO N NRHS A LDA IPIV B LDB WORK LWORK INFO
```

```
CALL DSYSV( 'L', 8, 3, A, 8, IPIV, B, 8, WORK, 0, INFO )
```

```
A = (same as input A in Example 1)
B = (same as input B in Example 1)
```

Output:

```
A =

\[
\begin{bmatrix}
3.0 & . & . & . & . & . & . & . \\
5.0 & 3.0 & . & . & . & . & . & . \\
1.0 & -1.0 & 4.0 & . & . & . & . & . \\
-1.0 & 1.0 & -1.0 & 8.0 & . & . & . & . \\
1.0 & 0.0 & 0.0 & -1.0 & 1.0 & . & . & . \\
0.0 & -1.0 & 1.0 & -1.0 & 3.0 & 1.0 & . & . \\
1.0 & -1.0 & 1.0 & -1.0 & -1.0 & 1.0 & 2.0 & . \\
-1.0 & 0.0 & 1.0 & -1.0 & 1.0 & 0.0 & 1.0 & 16.0
\end{bmatrix}
\]
```

```
IPIV = (-2 -2 3 4 -6 -6 7 8 )
```

```
B =

\[
\begin{bmatrix}
1.0 & 1.0 & 8.0 \\
1.0 & 2.0 & 7.0 \\
1.0 & 3.0 & 6.0 \\
1.0 & 4.0 & 5.0 \\
1.0 & 5.0 & 4.0 \\
1.0 & 6.0 & 3.0 \\
1.0 & 7.0 & 2.0 \\
1.0 & 8.0 & 1.0
\end{bmatrix}
\]
```

```
INFO = 0
```

Example 2

This example shows how to solve the system $AX = B$ for three right-hand sides, where indefinite complex symmetric matrix $A$ is the same matrix factored in the Example 2 for ZSYTRF.

Note: Because $lwork = 0$, the subroutine dynamically allocates $WORK$.

Call Statement and Input:

```
UPLO N NRHS A LDA IPIV B LDB WORK LWORK INFO
```

```
CALL ZSYSV( 'L', 4, 3, A, 4, IPIV, B, 4, WORK, 0, INFO )
```

```
A = (same as input A in Example 2)
B = (same as input B in Example 2)
```

Output:

```
A =

\[
\begin{bmatrix}
(0.368,-0.319) & . & . & . \\
(-0.062, 0.006) & (0.258,-0.147) & . & . \\
(0.625, 0.257) & (1.085,-0.335) & (0.333, 0.315) & . \\
(-0.462, 0.314) & (-0.444, 1.248) & (-0.437,-1.386) & (0.841, 0.431)
\end{bmatrix}
\]
```
Example 3

This example shows how to solve the system $AX = B$ for three right-hand sides, where indefinite complex Hermitian matrix $A$ is the same matrix factored in the [Example 3] for ZHETRF.

Notes:
1. Because $lwork = 0$, the subroutine dynamically allocates $WORK$.
2. 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:

```plaintext
UPLO N NRHS A LDA IPIV B LDB WORK LWORK INFO
CALL ZHESV('L', 4, 3, A, 4, IPIV, B, 4, WORK, 0, INFO )
```

A = (same as input A in Example 3)
B = (same as input B in Example 3)

Output:

```
A =
  [(-0.550, 0.000), (-0.027, 0.476), (0.062, 0.244), (-0.249, 0.022)]
  [(-0.483, 0.000), (-0.002, -0.269), (-0.002, -0.091), (-0.002, -0.269)]
  [(0.152, -0.244), (-0.490, 0.000), (-0.244, -0.002), (-0.490, 0.000)]
  [(0.244, -0.002), (-0.479, 0.000)]
```

```
B =
  [(0.060, 0.663), (-0.582, -1.410), (2.484, 2.216)]
  [(0.238, 0.410), (1.260, -0.430), (-1.273, 0.177)]
  [(1.562, 0.164), (6.213, 1.471), (-0.980, -2.551)]
```

INFO = 0

Example 4

This example shows how to solve the system $AX = B$ for three right-hand sides, where indefinite real symmetric matrix $A$ is the same matrix factored in the [Example 4] for DSYTRF.

Call Statement and Input:

```plaintext
UPLO N NRHS AP IPIV B LDB INFO
CALL DSPSV('L', 8, 3, AP, IPIV, B, 8, INFO )
```

AP = (same as input AP in Example 4)
B = (same as input B in Example 4)

Output:
Example 5

This example shows how to solve the system \( AX = B \) for three right-hand sides, where indefinite complex symmetric matrix \( A \) is the same matrix factored in the [Example 2](#) for ZSYTRF.

Call Statement and Input:

```fortran
CALL ZSPSV( 'L', 4, 3, AP, IPIV, B, 4, INFO )
```

AP = (same as input AP in [Example 5](#))

B = (same as input B in [Example 5](#))

Output:

\[
AP = \begin{pmatrix}
0.368 & -0.319 \\
-0.062 & 0.006 & 0.625 & 0.257 & -0.462 & 0.314 \\
0.258 & -0.147 & 1.085 & -0.335 & -0.444 & 1.248 \\
0.333 & 0.315 & -0.437 & 1.386 & 0.841 & 0.431 \\
\end{pmatrix}
\]

IPIV = (1 2 4 4)

\[
B = \begin{pmatrix}
0.409 & -0.663 \\
-0.582 & -1.410 & 2.484 & 2.216 \\
-1.664 & -0.552 & -1.503 & -4.837 & -3.577 & 2.575 \\
2.388 & 4.010 & 1.260 & -0.430 & -1.273 & 0.177 \\
1.562 & 0.164 & 6.213 & 1.471 & -0.980 & -2.551 \\
\end{pmatrix}
\]

INFO = 0

Example 6

This example shows how to solve the system \( AX = B \) for three right-hand sides, where indefinite complex Hermitian matrix \( A \) is the same matrix factored in the [Example 3](#) for ZHETRF.

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:
### CALL ZHPSV

```plaintext
Call ZHPSV('L', 4, 3, AP, IPIV, B, 4, INFO)
```

**AP** = (same as input AP in Example 6)

**B** = (same as input B in Example 6)

#### Output:

**AP** = 
\[
\begin{pmatrix}
(-0.550, 0.000),
(-0.027, 0.476),
(0.062, 0.244),
(-0.249, 0.022),
(-0.484, 0.000),
(-0.002, -0.269),
(0.152, -0.091),
(-0.490, 0.000),
(-0.245, -0.002),
(-0.479, 0.000)
\end{pmatrix}
\]

**IPIV** = (1 2 3 4)

\[
\begin{pmatrix}
(-1.623, -4.385)
& (2.635, -1.111)
& (-2.436, -0.306)

(-3.533, -0.212)
& (1.865, -2.830)
& (-0.866, -0.308)

(-1.742, -1.724)
& (-1.576, 1.575)
& (-0.478, -1.172)

(-1.537, -2.115)
& (-1.047, 1.608)
& (3.093, 0.365)
\end{pmatrix}
\]

**INFO** = 0
SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, ZHETRF, SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF, and ZHPTRF (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Factorization)

Purpose

These subroutines factor an indefinite real or complex symmetric or complex Hermitian matrix $A$. The matrix $A$ is factored using the Bunch-Kaufman diagonal pivoting method.

To solve the system of equations with one or more right-hand sides, follow the call to SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, or ZHETRF with a call to SSYTRS, DSYTRS, CSYTRS, ZSYTRS, CHETRS, or ZHETRS respectively.

To solve the system of equations with one or more right-hand sides, follow the call to SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF, or ZHPTRF with a call to SSPTRS, DSPTRS, CSPTRS, ZSPTRS, CHPTRS, or ZHPTRS respectively.

Table 151. Data Types

<table>
<thead>
<tr>
<th>$A$, $B$, work</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SSYTF, SSPTRF</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DSYTF, DSPTF</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CSYTF, CHETRF, CSPTRF, CHPTRF</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZSYTF, ZHETRF, ZSPTRF, ZHPTRF</td>
</tr>
</tbody>
</table>

Note:

- The output from the SSYTF, DSYTF, CSYTF, ZSYTF, CHETRF, or ZHETRF factorization routines should only be used as input to SSYTRS, DSYTRS, CSYTRS, ZSYTRS, CHETRS, or ZHETRS respectively.
- The output from the SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF, or ZHPTRF factorization routines should only be used as input to SSPTRS, DSPTRS, CSPTRS, ZSPTRS, CHPTRS, or ZHPTRS respectively.

On Entry

- **matrix_layout**
  
  Indicates whether the input and output matrices are stored in row major order or column major order, where:
  - If **matrix_layout** = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
  - If **matrix_layout** = LAPACK_COL_MAJOR, the matrices are stored in column major order.
Specified as: an integer. It must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

**uplo** indicates whether the upper or lower triangular part of the matrix \( A \) is referenced, where:

- If \( \text{uplo} = 'U' \), the upper triangular part is referenced.
- If \( \text{uplo} = 'L' \), the lower triangular part is referenced.

Specified as: a single character; \( \text{uplo} = 'U' \) or 'L'.

\( n \) is the order of matrix \( A \) used in the computation.
Specified as: an integer; \( n \geq 0 \).

\( a \) is the indefinite real symmetric, complex symmetric, or complex Hermitian matrix \( A \) of order \( n \).
- If \( \text{uplo} = 'U' \), it is stored in upper storage mode.
- If \( \text{uplo} = 'L' \), it is stored in lower storage mode.

Specified as: an \( \text{lda} \) by (at least) \( n \) array, containing numbers of the data type indicated in [Table 151 on page 651](#).

\( ap \) is the indefinite real symmetric, complex symmetric, or complex Hermitian matrix \( A \) of order \( n \). It is stored in an array, referred to as \( AP \), where:
- If \( \text{uplo} = 'U' \), it is stored in upper-packed storage mode.
- If \( \text{uplo} = 'L' \), it is stored in lower-packed storage mode.

Specified as: one-dimensional array of (at least) length \( n(n+1)/2 \), containing numbers of the data type indicated in [Table 151 on page 651](#).

\( \text{lda} \) is the leading dimension of the array specified for \( A \).
Specified as: an integer; \( \text{lda} > 0 \) and \( \text{lda} \geq n \).

\( \text{ipiv} \)

See [On Return](#).

**work** is a work area used by these subroutines, where:

- If \( \text{lwork} = 0 \), \( \text{work} \) is ignored.
- If \( \text{lwork} \neq 0 \), the size of \( \text{work} \) is determined as follows:
  - If \( \text{lwork} \neq -1 \), \( \text{work} \) is (at least) of length \( \text{lwork} \).
  - If \( \text{lwork} = -1 \), \( \text{work} \) is (at least) of length 1.

Specified as: an area of storage containing numbers of the data type indicated in [Table 151 on page 651](#).

**lwork** is the number of elements in array \( \text{WORK} \).
Specified as: an integer, where:
- If \( \text{lwork} = 0 \), the subroutine dynamically allocates the workspace needed for use during this computation. The work area is deallocated before control is returned to the calling program.
• If \( lwork = -1 \), subroutine performs a workspace query and returns the optimal required size of \( work \) in \( work_1 \). No computation is performed and the subroutine returns after error checking is complete.

• Otherwise, \( lwork \geq 1 \). It is suggested that the user specify \( lwork \geq 8n \).

\( info \)

See [On Return](#).

[On Return](#)

\( a \) is the transformed matrix \( A \) containing the results of the factorization. See "Function" on page 654.

Returned as: an \( lda \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 151 on page 651.

\( ap \) is the transformed matrix \( A \) containing the results of the factorization. See "Function" on page 654.

Returned as: a one-dimensional array of (at least) length \( n(n + 1)/2 \), containing numbers of the data type indicated in Table 151 on page 651.

\( ipiv \)

If \( info = 0 \), \( ipiv \) contains the pivot indices.

If \( ipiv_k > 0 \), then rows and columns \( k \) and \( ipiv_k \) were interchanged and \( D_{kk} \) is a \( 1 \times 1 \) diagonal block.

If \( uplo = 'U' \) and \( ipiv_k = ipiv_{k+1} \) \( \leq 0 \), then rows and columns \( k+1 \) and \( -ipiv_k \) were interchanged and \( D_{kk+1,k+1} \) is a \( 2 \times 2 \) diagonal block.

If \( uplo = 'L' \) and \( ipiv_k = ipiv_{k-1} \) \( \leq 0 \), then rows and columns \( k-1 \) and \( -ipiv_k \) were interchanged and \( D_{kk-1,kk-1} \) is a \( 2 \times 2 \) diagonal block.

Returned as: a one-dimensional integer array of (at least) length \( n \), containing integers.

\( work \)

is a work area used by this subroutine if \( lwork \neq 0 \), where:

If \( lwork \neq 0 \) and \( lwork \neq -1 \), its size is (at least) of length \( lwork \).

If \( lwork = -1 \), its size is (at least) of length 1.

Returned as: an area of storage, where:

If \( lwork \geq 1 \) or \( lwork = -1 \), then \( work_1 \) is set to the optimal \( lwork \) value and all other elements of \( work \) are overwritten.

\( info \)

has the following meaning:

• If \( info = 0 \), the factorization completed successfully.

• If \( info > 0 \), the factorization was unsuccessful and \( info \) is set to \( i \) where \( d_{ii} \) is exactly zero.

Returned as:

• For SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, ZHETRF, SSPTRF, DSPTRF, CPSYTRF, ZSPTRF, CHPTRF, and ZHPTRF, returned as: an integer; \( info \geq 0 \).

• For LAPACKE_ssytrf, LAPACKE_dsytrf, LAPACKE_syrtrf, LAPACKE_zsytrf, LAPACKE_chetr, LAPACKE_zhetr, LAPACKE_ssptrf, LAPACKE_dsptrf, LAPACKE_csptrf, LAPACKE_zsptrf, LAPACKE_chptrf, and LAPACKE_zhptrf, returned as an integer function value; \( info \geq 0 \).
Specified as: an integer; \( info \geq 0 \).

**Notes**

1. These subroutines accept lowercase letters for the `uplo` argument.
2. In your C program, argument `info` must be passed by reference.
3. \( a, ap, ipiv, \) and `work` must have no common elements; otherwise, results are unpredictable.
4. For a description of how real and complex symmetric matrices are stored in lower or upper storage mode, see “Lower Storage Mode” on page 88 or “Upper Storage Mode” on page 89, respectively.
5. For a description of how complex Hermitian matrices are stored in lower or upper storage mode, see “Complex Hermitian Matrix” on page 90.
6. For a description of how real and complex symmetric matrices are stored in lower- or upper-packed storage mode, see “Lower-Packed Storage Mode” on page 85 or “Upper-Packed Storage Mode” on page 87, respectively.
7. For a description of how complex Hermitian matrices are stored in lower- or upper-packed storage mode, see “Complex Hermitian Matrix” on page 90.

**Function**

For SSYTRF, DSYTRF, CSYTRF, ZSYTRF, SSPTRF, DSPTRF, CSPTRF, and ZSPTRF:

The indefinite real or complex symmetric matrix \( A \) is factored using the Bunch-Kaufman diagonal pivoting method, where \( A \) is expressed as one of the following:

\[
A = UDU^T
\]

\[
A = LDL^T
\]

where:

\( U \) is a product of permutation and unit upper triangular matrices.

\( L \) is a product of permutation and unit lower triangular matrices.

\( D \) is a symmetric block diagonal matrix, consisting of \( 1 \times 1 \) and \( 2 \times 2 \) diagonal blocks.

Matrix \( A \) is stored as follows:

- For SSYTRF, DSYTRF, CSYTRF, and ZSYTRF, matrix \( A \) is stored in upper or lower storage mode.
- For SSPTRF, DSPTRF, CSPTRF, and ZSPTRF, matrix \( A \) is stored in upper- or lower-packed storage mode.

For CHETRF, ZHETRF, CHPTRF, and ZHPTRF:
The indefinite complex Hermitian matrix $A$ is factored using the Bunch-Kaufman diagonal pivoting method, where $A$ is expressed as one of the following:

$$A = UDU^H$$

$$A = LDL^H$$

where:

$U$ is a product of permutation and unit upper triangular matrices.

$L$ is a product of permutation and unit lower triangular matrices.

$D$ is a complex Hermitian block diagonal matrix, consisting of $1 \times 1$ and $2 \times 2$ diagonal blocks.

Matrix $A$ is stored as follows:

- For CHETRF and ZHETRF, matrix $A$ is stored in upper or lower storage mode.
- For CHPTRF and ZHPTRF, matrix $A$ is stored in upper- or lower-packed storage mode.

If $n$ is 0, no computation is performed. See references [8 on page 1363] and [18 on page 1364].

**Error conditions**

**Resource Errors**

$lwork = 0$ and unable to allocate work area

**Computational Errors**

- Matrix $A$ is singular.
- The factorization completed but the block diagonal matrix $D$ is exactly singular. $info$ is set to $i$, where $d_{ii}$ is exactly zero. This 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 2147 is set to be unlimited in the ESSL error option table. For details, see "What Can You Do about ESSL Computational Errors?" on page 68.

**Input-Argument Errors**

1. $uplo \neq 'U'$ or 'L'
2. $n < 0$
3. $lda \leq 0$
4. $n > lda$
5. $lwork \neq 0$ and $lwork \neq -1$ and $lwork <$ the minimum required value

**Examples**

**Example 1**

This example shows a factorization of the indefinite real symmetric matrix $A$ of order 8.

Matrix $A$ is the same matrix factored in the Example 1 for DBSTRF.

**Note:** Because $lwork = 0$, the subroutine dynamically allocates WORK.

Call Statement and Input:
CALL DSYTRF('L', 8, A, 8, IPIV, WORK, 0, INFO)

\[
A = \begin{bmatrix}
3.0 & . & . & . & . & . & . \\
5.0 & 3.0 & . & . & . & . & . \\
-2.0 & 2.0 & 0.0 & . & . & . & . \\
2.0 & -2.0 & 0.0 & 8.0 & . & . & . \\
3.0 & 5.0 & -2.0 & -6.0 & 12.0 & . & . \\
-5.0 & -3.0 & 2.0 & -10.0 & 6.0 & 16.0 & . \\
-2.0 & 2.0 & 0.0 & -8.0 & 8.0 & 8.0 & 6.0 \\
-3.0 & -5.0 & 6.0 & -14.0 & 6.0 & 20.0 & 18.0 & 34.0
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
3.0 & . & . & . & . & . & . \\
5.0 & 3.0 & . & . & . & . & . \\
-1.0 & 1.0 & -1.0 & 8.0 & . & . & . \\
1.0 & 0.0 & 0.0 & -1.0 & 1.0 & . & . \\
0.0 & -1.0 & 1.0 & -1.0 & 3.0 & 1.0 & . \\
1.0 & -1.0 & 1.0 & -1.0 & 1.0 & 2.0 & . \\
-1.0 & 0.0 & 1.0 & -1.0 & 1.0 & 0.0 & 1.0 & 16.0
\end{bmatrix}
\]

IPIV = (-2 -2 3 4 -6 -6 7 8)
INFO = 0

**Example 2**

This example shows a factorization of the indefinite complex symmetric matrix \( A \) of order 4.

Matrix \( A \) is:

\[
\begin{bmatrix}
(0.368, -0.319) & (-0.021, 0.022) & (0.312, -0.105) & (-0.070, 0.263) \\
(-0.021, 0.022) & (0.259, -0.148) & (0.212, -0.237) & (0.071, 0.370) \\
(0.312, -0.105) & (0.212, -0.237) & (0.273, -0.041) & (0.384, -0.056) \\
(-0.070, 0.263) & (0.071, 0.370) & (0.384, -0.056) & (-0.230, 0.085)
\end{bmatrix}
\]

**Note:** Because \( \text{lwork} = 0 \), the subroutine dynamically allocates \( \text{WORK} \).

Call Statement and Input:

```
CALL DSYTRF('L', 4, A, 4, IPIV, WORK, 0, INFO)
```

\[
\begin{bmatrix}
(0.368, -0.319) & . & . & . \\
(-0.021, 0.022) & (0.259, -0.148) & . & . \\
(0.312, -0.105) & (0.212, -0.237) & (0.273, -0.041) & . \\
(-0.070, 0.263) & (0.071, 0.370) & (0.384, -0.056) & (-0.230, 0.085)
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
(0.368, -0.319) & . & . & . \\
(-0.062, 0.006) & (0.258, -0.147) & . & . \\
(0.625, 0.257) & (1.085, -0.335) & (0.333, 0.315) & . \\
(-0.462, 0.314) & (-0.444, 1.248) & (-0.437, -1.386) & (0.841, 0.431)
\end{bmatrix}
\]
Example 3

This example shows a factorization of the indefinite complex Hermitian matrix $A$ of order 4.

Matrix $A$ is:

$$
\begin{bmatrix}
(-0.550, 0.000) & (0.015, 0.262) & (-0.034, 0.134) & (0.137, 0.012) \\
(0.015,-0.262) & (-0.609, 0.000) & (-0.062,-0.150) & (-0.083, 0.021) \\
(-0.034,-0.134) & (-0.062, 0.150) & (-0.560, 0.000) & (-0.126, 0.053) \\
(0.137,-0.012) & (-0.083,-0.021) & (-0.126,-0.053) & (-0.558, 0.000)
\end{bmatrix}
$$

Notes:
1. Because $lwork = 0$, the subroutine dynamically allocates $WORK$.
2. 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:

```fortran
CALL ZHETRF( 'L' , 4 , A , 4 , IPIV , WORK , 0 , INFO )
```

Output:

$$
\begin{bmatrix}
(-0.550, 0.000) & . & . & . \\
(0.015,-0.262) & (-0.609, .) & . & . \\
(-0.034,-0.134) & (-0.062, 0.150) & (-0.560, .) & . \\
(0.137,-0.012) & (-0.083,-0.021) & (-0.126,-0.053) & (-0.558, .)
\end{bmatrix}
$$

Example 4

This example shows a factorization of the indefinite real symmetric matrix $A$ of order 8.

Matrix $A$ is the same matrix factored in the Example 1 for DBSTRF.

Call Statement and Input:

```fortran
CALL DSPTRF( 'L' , 8 , AP , IPIV , INFO )
```

Output:

$$
\begin{bmatrix}
(-0.550, 0.000) & . & . & . \\
(-0.027, 0.476) & (-0.483, 0.000) & . & . \\
(0.062, 0.244) & (-0.002,-0.269) & (-0.490, 0.000) & . \\
(-0.249, 0.022) & (0.152,-0.091) & (0.244,-0.002) & (-0.479, 0.000)
\end{bmatrix}
$$

IPIV = ( 1 2 4 4 )
INFO = 0
Example 5

This example shows a factorization of the indefinite complex symmetric matrix $A$ of order 4.

Matrix $A$ is the same matrix factored in the Example 2 for ZSYTRF.

Call Statement and Input:

```fortran
CALL ZSPTRF( 'L', 4, AP, IPIV, INFO )
```

Output:

```
AP = ( ( 0.368, -0.319 ), (-0.021, 0.022), (0.312, -0.105), (-0.070, 0.263),
      ( 0.259, -0.148), (0.212, -0.041), (0.384, -0.056),
      (-0.230, 0.085) )
```

IPIV = ( -2 -2 3 4 -6 7 8 )
INFO = 0

Example 6

This example shows a factorization of the indefinite complex Hermitian matrix $A$ of order 4.

Matrix $A$ is the same matrix factored in the Example 3 for ZHETRF.

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:

```fortran
CALL ZHPTRF( 'L', 4, AP, IPIV, INFO )
```

Output:

```
AP = ( ( 0.368, -0.319 ), (-0.062, 0.006), (0.625, 0.257), (-0.462, 0.314),
      ( 0.258, -0.147), (1.085, -0.335), (-0.444, 1.248),
      ( 0.333, 0.315), (-0.437, -1.386),
      ( 0.841, 0.431) )
```

IPIV = ( 1 2 4 4 )
INFO = 0
Output:

\[
AP = ( (-0.550, 0.000), (-0.027, 0.476), ( 0.062, 0.244), (-0.249, 0.022), (-0.484, 0.000), (-0.002, -0.269), ( 0.152, -0.091), (-0.490, 0.000), ( 0.245, -0.002), (-0.479, 0.000) )
\]

IPIV = ( 1 2 3 4 )
INFO = 0
SSYTRS, DSYTRS, CSYTRS, ZSYTRS, CHETRS, ZHETRS, SSPTRS, DSPTRS, CSPTRS, ZSPTRS, CHPTRS, and ZHPTRS (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)

Purpose

These subroutines solve the system $AX = B$ for $X$, where $A$ is an indefinite real or complex symmetric or complex Hermitian matrix and $X$ and $B$ are general matrices.

SSYTRS, DSYTRS, CSYTRS, ZSYTRS, CHETRS, or ZHETRS use the results of the factorization of matrix $A$, produced by a preceding call to SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, or ZHETRF, respectively.

SSPTRS, DSPTRS, CSPTRS, ZSPTRS, CHPTRS, or ZHPTRS use the results of the factorization of matrix $A$, produced by a preceding call to SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF, or ZHPTRF, respectively.

<table>
<thead>
<tr>
<th>Table 152. Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A, B$</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
<tr>
<td>Short-precision complex</td>
</tr>
<tr>
<td>Long-precision complex</td>
</tr>
</tbody>
</table>

$^a$LAPACK

Syntax

Fortran

| CALL SSYTRS | DSYTRS | CSYTRS | ZSYTRS | CHETRS | ZHETRS (uplo, n, nrhs, a, lda, ipiv, b, ldb, info) |
| CALL SSPTRS | DSPTRS | CSPTRS | ZSPTRS | CHPTRS | ZHPTRS (uplo, n, nrhs, ap, ipiv, b, ldb, info); |

C and C++

| ssytrs | dsytrs | csysytrs | zsysytrs | chetsrs | zhetsrs (uplo, a, b); |
| ssytrs | dsysytrs | csytrs | zsytrs | chetsrs | zhetsrs (uplo, a, b); |

LAPACK

| info = LAPACK_Ssytrs | LAPACK_Dsytrs | LAPACK_Csysytrs | LAPACK_Zsysytrs | LAPACK_Chetsrs | LAPACK_Zhetsrs (matrix_layout, uplo, n, nrhs, a, lda, ipiv, b, ldb); |
| info = LAPACK_Ssptrs | LAPACK_Dsptrs | LAPACK_Csptrs | LAPACK_Zsptrs | LAPACK_Cchptrs | LAPACK_Zchptrs (matrix_layout, uplo, n, nrhs, ap, ipiv, b, ldb); |

On Entry

**matrix_layout**

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If `matrix_layout` = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If `matrix_layout` = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. It must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR

**uplo**

indicates whether the upper or lower triangular part of the matrix $A$ is referenced, where:
If \( uplo = 'U' \), the upper triangular part is referenced.
If \( uplo = 'L' \), the lower triangular part is referenced.

Specified as: a single character; \( uplo = 'U' \) or 'L'.

\( n \) is the order of matrix \( A \) used in the computation.
Specified as: an integer; \( n \geq 0 \).

\( nrhs \)
is the number of right-hand sides; that is, the number of columns of matrix \( B \).
Specified as: an integer; \( nrhs \geq 0 \).

\( a \) is the factorization of indefinite real symmetric, complex symmetric, or complex Hermitian matrix \( A \) of order \( n \) produced by a preceding call to SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, or ZHETRF, respectively.
If \( uplo = 'U' \), it is stored in upper storage mode.
If \( uplo = 'L' \), 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 152 on page 660.

\( ap \) is the factorization of indefinite real symmetric, complex symmetric, or complex Hermitian matrix \( A \) of order \( n \), produced by a preceding call to SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF, or ZHPTRF, respectively.
If \( uplo = 'U' \), it is stored in upper-packed storage mode.
If \( uplo = 'L' \), it is 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 152 on page 660.

\( lda \)
is the leading dimension of the array specified for \( A \).
Specified as: an integer; \( lda > 0 \) and \( lda \geq n \).

\( ipiv \)
is the array containing the pivot indices produced by a preceding call to SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, ZHETRF, SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF, or ZHPTRF, respectively.
Specified as: a one-dimensional integer array of (at least) length \( n \), containing integers.

\( b \) is the general 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 \( nrhs \) array, containing numbers of the data type indicated in Table 152 on page 660.

\( ldb \)
is the leading dimension of the array specified for \( B \).
Specified as: an integer; \( ldb > 0 \) and \( ldb \geq n \).

\( info \)
See On Return

On Return

\( b \) is the matrix \( X \), containing the \( nrhs \) solutions to the system. The solutions, each of length \( n \), reside in the columns of \( X \).
Returned as: an ldb by (at least) nrhs array, containing numbers of the data type indicated in Table 152 on page 660.

info

has the following meaning:

If info=0, the factorization completed successfully.

Returned as:

- For SSYTRS, DSYTRS, CSYTRS, ZSYTRS, CHETRS, ZHETRS, SSPTRS, DSPTRS, CSPTRS, ZSPTRS, CHPTRS, and ZHPTRS, returned as: an integer; info ≥ 0.
- For LAPACKE_ssytrs, LAPACKE_dsytrs, LAPACKE_csytrs, LAPACKE_zsytrs, LAPACKE_chetrs, LAPACKE_zhetrs, LAPACKE_ssptrs, LAPACKE_dsptrs, LAPACKE_csptrs, LAPACKE_zsptrs, LAPACKE_chptrs, and LAPACKE_zhptrs, returned as an integer function value; info ≥ 0.

Notes

1. These subroutines accept lowercase letters for the uplo argument.
2. In your C program, argument info must be passed by reference.
3. a, ap, b and ipiv must have no common elements; otherwise, results are unpredictable.
4. For a description of how real and complex symmetric matrices are stored in lower or upper storage mode, see “Lower Storage Mode” on page 88 or “Upper Storage Mode” on page 89, respectively.
   For a description of how complex Hermitian matrices are stored in lower or upper storage mode, see “Complex Hermitian Matrix” on page 90.
5. For a description of how real and complex symmetric matrices are stored in lower- or upper-packed storage mode, see “Lower-Packed Storage Mode” on page 85 or “Upper-Packed Storage Mode” on page 87, respectively.
   For a description of how complex Hermitian matrices are stored in lower- or upper-packed storage mode, see “Complex Hermitian Matrix” on page 90.

Function

These subroutines solve the system $AX = B$ for $X$, where $A$ is a real or complex symmetric or complex Hermitian matrix and $X$ and $B$ are general matrices.

SSYTRS, DSYTRS, CSYTRS, ZSYTRS, CHETRS, or ZHETRS use the results of the factorization of matrix $A$, produced by a preceding call to SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, or ZHETRF, respectively.

SSPTRS, DSPTRS, CSPTRS, ZSPTRS, CHPTRS, or ZHPTRS use the results of the factorization of matrix $A$, produced by a preceding call to SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF, or ZHPTRF, respectively.

If $n$ is 0 or nrhs is 0, no computation is performed and the subroutine returns after doing some parameter checking. See references [8 on page 1363] and [18 on page 1364].

Error conditions

Computational Errors

None
Note: If the factorization performed by SSYTRF, DSYTRF, CSYTRF, 
ZSYTRF, CHETRF, ZHETRF, SSPTRF, DSPTRF, CSPTRF, ZSPTRF, CHPTRF,
or ZHPTRF failed because 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. uplo ≠ 'U' or 'L'
2. n < 0
3. nrhs < 0
4. lda ≤ 0
5. n > lda
6. ldb ≤ 0
7. n > ldb

Examples

Example 1

This example shows how to solve the system $AX=B$, for three right-hand sides, 
where indefinite real symmetric matrix $A$ is the same matrix factored in the 
Example 1 for DSYTRF.

Call Statement and Input:
```
CALL DSYTRS( 'L', 8, 3, A, 8, IPIV, B, 8, INFO )
```

$A = \begin{pmatrix}
1.0 & -38.0 & 47.0 \\
7.0 & -10.0 & 73.0 \\
6.0 & 52.0 & 2.0 \\
\end{pmatrix}$

$IPIV = \begin{pmatrix}
\end{pmatrix}$

$B = \begin{pmatrix}
-30.0 & -228.0 & -42.0 \\
32.0 & 183.0 & 105.0 \\
34.0 & 297.0 & 9.0 \\
32.0 & 244.0 & 44.0 \\
62.0 & 497.0 & 61.0 \\
\end{pmatrix}$

Output:
```
\begin{pmatrix}
1.0 & 1.0 & 8.0 \\
1.0 & 2.0 & 7.0 \\
1.0 & 3.0 & 6.0 \\
\end{pmatrix}
```

INFO = 0

Example 2

This example shows how to solve the system $AX=B$ for three right-hand sides, 
where indefinite complex symmetric matrix $A$ is the same matrix factored in the 
Example 2 for ZSYTRF.

Call Statement and Input:
Example 3

This example shows how to solve the system $AX=B$ for three right-hand sides, where indefinite complex Hermitian matrix $A$ is the same matrix factored in the Example 3 for ZHETRF.

Call Statement and Input:

```fortran
CALL ZSYTRS('L', 4, 3, A, 4, IPIV, B, 4, INFO )
```

```fortran
A = (same as output A in Example 2)
IPIV = (same as output IPIV in Example 2)
```

```fortran
B =
( 1.000, 1.000) (-1.000, 1.000) ( 2.000, 0.000 )
( 1.000, 1.000) (-1.000, 1.000) ( 0.000, 1.000 )
( 1.000, 1.000) ( 1.000,-1.000) ( 0.000, 1.000 )
( 1.000, 1.000) ( 1.000,-1.000) (-2.000, 0.000 )
```

Output:

```fortran
B =
( 0.409, -0.663) (-0.582,-1.410) ( 2.484, 2.216 )
(-1.664, -0.552) (-1.503,-4.837) (-3.577, 2.575 )
( 2.388, 4.010) ( 1.260,-0.430) (-1.273, 0.177 )
( 1.562, 0.164) ( 6.213, 1.471) (-0.980,-2.551 )
```

INFO = 0

Example 4

This example shows how to solve the system $AX=B$ for three right-hand sides, where indefinite real symmetric matrix $A$ is the same matrix factored in the Example 1 for DSYTRF.

Call Statement and Input:

```fortran
CALL DSPTRS('L', 8, 3, AP, IPIV, B, 8, INFO )
```

```
664  ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
```
AP = (same as output AP in Example 4)
IPIV = (same as output IPIV in Example 4)

\[
\begin{bmatrix}
1.0 & -38.0 & 47.0 \\
7.0 & -10.0 & 73.0 \\
6.0 & 52.0 & 2.0
\end{bmatrix}
\]

B =
\[
\begin{bmatrix}
-30.0 & -228.0 & -42.0 \\
32.0 & 183.0 & 105.0 \\
34.0 & 297.0 & 9.0 \\
32.0 & 244.0 & 44.0 \\
62.0 & 497.0 & 61.0
\end{bmatrix}
\]

Output:
\[
\begin{bmatrix}
1.0 & 1.0 & 8.0 \\
1.0 & 2.0 & 7.0 \\
1.0 & 3.0 & 6.0
\end{bmatrix}
\]

B =
\[
\begin{bmatrix}
1.0 & 4.0 & 5.0 \\
1.0 & 5.0 & 4.0 \\
1.0 & 6.0 & 3.0 \\
1.0 & 7.0 & 2.0 \\
1.0 & 8.0 & 1.0
\end{bmatrix}
\]

INFO = 0

Example 5

This example shows how to solve the system $AX=B$ for three right-hand sides, where indefinite complex symmetric matrix $A$ is the same matrix factored in the Example 2 for ZSYTRF.

Call Statement and Input:

```
CALL ZSPTRS( 'L', 4, 3, AP, IPIV, B, LDB, INFO )
```

AP = (same as output AP in Example 5)
IPIV = (same as output IPIV in Example 5)

\[
B = \begin{bmatrix}
(1.000, 1.000) & (-1.000, 1.000) & (2.000, 0.000) \\
(1.000, 1.000) & (-1.000, 1.000) & (0.000, 1.000) \\
(1.000, 1.000) & (1.000,-1.000) & (0.000, 1.000) \\
(1.000, 1.000) & (1.000,-1.000) & (-2.000, 0.000)
\end{bmatrix}
\]

Output:
\[
B = \begin{bmatrix}
(0.409,-0.663) & (-0.582,-1.410) & (2.404, 2.216) \\
(-1.664,-0.552) & (-1.503,-4.837) & (-3.577, 2.575) \\
(2.388, 4.010) & (1.260,-0.430) & (-1.273, 0.177) \\
(1.562, 0.164) & (6.213, 1.471) & (-0.900,-2.551)
\end{bmatrix}
\]

INFO = 0

Example 6

This example shows how to solve the system $AX=B$ for three right-hand sides, where indefinite complex Hermitian matrix $A$ is the same matrix factored in the Example 3 for ZHETRF.

Call Statement and Input:
CALL ZHPTRS( 'L', 4, 3, AP, IPIV, B, 4, INFO )

AP = (same as output AP in Example 6)

IPIV = (same as output IPIV in Example 6)

\[
\begin{bmatrix}
(1.000, 1.000) & (-1.000, 1.000) & (2.000, 0.000) \\
(1.000, 1.000) & (-1.000, 1.000) & (0.000, 1.000) \\
(1.000, 1.000) & (1.000, -1.000) & (0.000, 1.000) \\
(1.000, 1.000) & (1.000, -1.000) & (-2.000, 0.000)
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
(-1.623, -4.385) & (2.635, -1.111) & (-2.436, -0.306) \\
(-3.533, -0.212) & (1.865, -2.830) & (-0.866, -0.308) \\
(-1.742, -1.724) & (-1.576, 1.575) & (-0.478, -1.172) \\
(-1.537, -2.115) & (-1.047, 1.608) & (3.093, 0.365)
\end{bmatrix}
\]

INFO = 0
DBSSV (Symmetric Indefinite Matrix Factorization and Multiple Right-Hand Side Solve)

Purpose

The DBSSV subroutine solves a system of linear equations $AX = B$ for $X$, where $A$ is a real symmetric indefinite matrix, and $X$ and $B$ are real general matrices.

The matrix $A$, stored in upper- or lower-packed storage mode, is factored using the Bunch-Kaufman diagonal pivoting method, where $A$ is expressed as:

$$A = UDU^T \quad \text{or} \quad A = LDL^T$$

where:

$U$ is a product of permutation and unit upper triangular matrices.

$L$ is a product of permutation and unit lower triangular matrices.

$D$ is a symmetric block diagonal matrix, consisting of $1 \times 1$ and $2 \times 2$ diagonal blocks.

Table 153. Data Types

<table>
<thead>
<tr>
<th>$A$, $B$</th>
<th>$ipvt$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long-precision real</td>
<td>Integer</td>
<td>DBSSV</td>
</tr>
</tbody>
</table>

Syntax

Fortran

CALL DBSSV (uplo, n, nrhs, ap, ipvt, bx, ldb, nsinfo)

C and C++

dbssv (uplo, n, nrhs, ap, ipvt, bx, ldb, nsinfo);

On Entry

$uplo$

indicates whether matrix $A$ is stored in upper- or lower-packed storage mode, where:

If $uplo = 'U'$, $A$ is stored in upper-packed storage mode.

If $uplo = 'L'$, $A$ is stored in lower-packed storage mode.

Specified as: a single character. It must be 'U' or 'L'.

$n$

is the order $n$ of matrix $A$ and the number of rows of matrix $B$.

Specified as: an integer; $n \geq 0$.

$nrhs$

is the number of right-hand sides; i.e., the number of columns of matrix $B$.

Specified as: an integer; $nrhs \geq 0$.

$ap$

is array, referred to as $Ap$, in which matrix $A$, to be factored, is stored in upper- or lower-packed storage mode.

Specified as: a one-dimensional array of length $nsinfo$, containing numbers of the data type indicated in Table 153. See “Notes on page 669.”
**ipvt**
See [On Return](#).

**bx** 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 \( ld_b \) by (at least) \( nrhs \) array, containing numbers of the data type indicated in Table 153 on page 667.

**ldb**
is the leading dimension of the array specified for \( B \).
Specified as: an integer; \( ldb > 0 \) and \( ldb \geq n \).

**nsinfo**
is the number of elements in array, \( AP \).
If \( n \leq nco \), \( nsinfo = n(n + 1) / 2 \)
Where:
\( ics \) is the size in doublewords of the data cache. The data cache size can be obtained by utilizing the following C language code fragment:
```c
#include <sys/systemcfg.h>
int ics;
    ...
ics=_system_configuration.dcache_size/8;
```
Otherwise, to determine a sufficient amount of storage, use the following processor-independent formula:
\[
nco = \sqrt{2.0(ics)}
\]

To determine the minimal amount of storage see "Notes" on page 669.
Specified as: an integer; \( nsinfo > 0 \).

**On Return**

**ap** is the transformed matrix \( A \) of order \( n \), containing the results of the factorization.
If \( nsinfo \geq 0 \) and \( n > nco \), additional information that can be used to obtain a minimum \( nsinfo \) is stored in \( AP(1) \). See "Notes" on page 669 and "Function" on page 669.
Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 153 on page 667.
**ipvt**

is an integer vector of length \( n \), containing the pivot information necessary to construct the factored form of \( A \).

Returned as: a one-dimensional array of (at least) length \( n \), containing numbers of the data type indicated in Table 153 on page 667.

**bx**

is the matrix \( X \), containing the \( \text{nrhs} \) solutions to the system. The solutions, each of length \( n \), reside in the columns of \( X \).

Returned as: an \( ldb \) by (at least) \( \text{nrhs} \) array, containing numbers of the data type indicated in Table 153 on page 667.

**nsinfo**

indicates the result of the computation.

- If \( nsinfo = 0 \), the subroutine completed successfully.
- If \( nsinfo > 0 \), factorization was unsuccessful and array \( B \) was not updated. \( nsinfo \) is set to \( i \) where \( d_{ii} \) is exactly zero.
- If \( nsinfo < 0 \), factorization did not take place and the arrays, \( \text{AP} \) and \( B \), remain unchanged. \( |nsinfo| \) is the minimal storage required for factorization to take place. Error message 2200 is issued and execution terminates, unless you have used ERRSET to make error code 2200 recoverable. See “What Can You Do about ESSL Input-Argument Errors?” on page 67.

Specified as: an integer.

**Notes**

1. This subroutine accepts lowercase letters for the \( \text{uplo} \) argument.
2. In your C program, argument \( nsinfo \) must be passed by reference.
3. In the input array specified for \( \text{ap} \), the first \( n(n+1)/2 \) elements are matrix elements. The additional locations, required in the array, are used for working storage.
4. The vectors and matrices used in this computation must have no common elements; otherwise, results are unpredictable. See “Concepts” on page 75.
5. On return, if \( nsinfo \geq 0 \) and \( n > \text{nco} \), \( \text{ap} \) contains additional information in \( \text{AP}(1) \) that can be used to obtain the minimal required \( nsinfo \). This information can be accessed using the following code fragment:

   ```
   REAL*8 \text{AP} (NSINFO)
   INTEGER API(2)
   EQUIVALENCE (API, \text{AP})
   ...
   NSINFOMIN = API(2)
   ```

6. For a description of how a symmetric matrix is stored in upper- or lower-packed storage mode in an array, see “Symmetric Matrix” on page 85.

**Function**

The system \( AX = B \) is solved for \( X \), where \( A \) is a real symmetric indefinite matrix, and \( X \) and \( B \) are real general matrices.

The matrix \( A \), stored in upper- or lower-packed storage mode, is factored using the Bunch-Kaufman diagonal pivoting method, where \( A \) is expressed as:

\[
A = UDU^T \quad \text{or} \quad A = LDL^T
\]
where:

**U** is a product of permutation and unit upper triangular matrices.

**L** is a product of permutation and unit lower triangular matrices.

**D** is a symmetric block diagonal matrix, consisting of $1 \times 1$ and $2 \times 2$ diagonal blocks.

If $n$ is 0, no computation is performed and the subroutine returns after doing some parameter checking. If $n > 0$ and *nrhs* is 0, no solutions are computed and the subroutine returns after factoring the matrix.

See references [8 on page 1363] and [76 on page 1368].

**Error conditions**

**Resource Errors**

None.

**Computational Errors**

Matrix **A** is singular.

- The factorization completed but the block diagonal matrix **D** is exactly singular. *nsinfo* is set to *i*, where $d_{ii}$ is exactly zero. This 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 2147 is set to be unlimited in the ESSL error option table. For details, see “What Can You Do about ESSL Computational Errors?” on page 68.

**Input-Argument Errors**

1. *uplo* ≠ 'U' or 'L'
2. $n < 0$
3. $n > ldb$
4. $ldb \leq 0$
5. *nrhs* < 0
   - For the minimum value, see the *nsinfo* argument description.
   - Return code 1 is returned if error 2200 is recoverable.

**Examples**

**Example 1**

This example shows how to solve the system $AX = B$, for three right-hand sides, where matrix **A** is a real symmetric indefinite matrix of order 8, stored in lower-packed storage mode, and **X** and **B** are real general matrices.

On input, matrix **A** is:

\[
\begin{bmatrix}
3.0 & 5.0 & -2.0 & 2.0 & 3.0 & -5.0 & -2.0 & -3.0 \\
5.0 & 3.0 & 2.0 & -2.0 & 5.0 & -3.0 & 2.0 & -5.0 \\
-2.0 & 2.0 & 0.0 & 0.0 & -2.0 & 2.0 & 0.0 & 6.0 \\
2.0 & -2.0 & 0.0 & 8.0 & -6.0 & -10.0 & -8.0 & -14.0 \\
3.0 & 5.0 & -2.0 & -6.0 & 12.0 & 6.0 & 8.0 & 6.0 \\
-5.0 & -3.0 & 2.0 & -10.0 & 6.0 & 16.0 & 8.0 & 20.0
\end{bmatrix}
\]
Note: The AP array is formatted in a triangular arrangement for readability; however, it is stored in lower-packed storage mode.

Call Statement and Input:

Call DBSSV ( 'L', 8, 3, AP, IPVT, BX, 8, 36 )

AP = ( 3.0, 5.0, -2.0, 2.0, 3.0, -5.0, -2.0, -3.0, 
     3.0, 2.0, -2.0, 5.0, -3.0, 2.0, -5.0, 
     0.0, 0.0, -2.0, 2.0, 0.0, 6.0, 
     8.0, -6.0, -10.0, -8.0, -14.0, 
     12.0, 6.0, 8.0, 6.0, 
     16.0, 8.0, 20.0, 
     6.0, 18.0, 
     34.0 )

Output:

BX =

| 1.0 1.0 8.0 |
| 1.0 2.0 7.0 |
| 1.0 3.0 6.0 |
| 1.0 4.0 5.0 |
| 1.0 5.0 4.0 |
| 1.0 6.0 3.0 |
| 1.0 7.0 2.0 |
| 1.0 8.0 1.0 |

NSINFO = 0

Note: AP and IPVT are stored in an internal format.

Example 2

This example shows how to solve the system $AX = B$, for three right-hand sides, where matrix $A$ is a real symmetric indefinite matrix of order 8, stored in upper-packed storage mode, and $X$ and $B$ are real general matrices.

On input, matrix $A$ is:

\[
\begin{bmatrix}
34.0 & 18.0 & 17.0 & 6.0 & -14.0 & 6.0 & -5.0 & -3.0 \\
18.0 & 6.0 & 6.0 & 8.0 & -8.0 & 0.0 & 2.0 & -2.0 \\
17.0 & 6.0 & 9.0 & 9.0 & -8.0 & 0.0 & 2.0 & -2.0 \\
6.0 & 8.0 & 9.0 & 12.0 & -6.0 & -2.0 & 5.0 & 3.0 \\
-14.0 & -8.0 & -8.0 & -6.0 & 8.0 & 0.0 & -2.0 & 2.0 \\
-6.0 & 0.0 & 0.0 & -2.0 & 0.0 & 0.0 & 2.0 & -2.0 \\
-5.0 & 2.0 & 2.0 & 5.0 & -2.0 & 2.0 & 3.0 & 5.0 \\
-3.0 & -2.0 & -2.0 & 3.0 & 2.0 & -2.0 & 5.0 & 3.0
\end{bmatrix}
\]
**Note:** The AP array is formatted in a triangular arrangement for readability; however, it is stored in upper-packed storage mode.

**Call Statement and Input:**

```
UPLO N NRHS AP IPVT BX LDB NSINFO
| | | | | | |
CALL DBSSV ( 'U', 8, 3, AP, IPVT, BX, 8, 36 )
```

```
AP = ( 34.0,
18.0, 6.0,
17.0, 6.0, 9.0,
6.0, 8.0, 9.0, 12.0,
-14.0, -8.0, -8.0, -6.0, 8.0,
6.0, 0.0, 0.0, -2.0, 0.0, 0.0,
-5.0, 2.0, 2.0, 5.0, -2.0, 2.0, 3.0,
-3.0, -2.0, -2.0, 3.0, 2.0, -2.0, 5.0, 3.0 )
```

```
BX =
35.0, 114.0, 201.0
-28.0, -36.0, -216.0
 4.0,  -4.0,  40.0
12.0,  88.0,  20.0
 4.0,  56.0, -20.0
```

**Output:**

```
1.0, 1.0, 8.0
1.0, 2.0, 7.0
1.0, 3.0, 6.0
```

```
BX =
1.0, 4.0, 5.0
1.0, 5.0, 4.0
1.0, 6.0, 3.0
1.0, 7.0, 2.0
1.0, 8.0, 1.0
```

**NSINFO = 0**

**Note:** AP and IPVT are stored in an internal format.
DBSTRF (Symmetric Indefinite Matrix Factorization)

Purpose

DBSTRF factors a real symmetric indefinite matrix $A$. The matrix $A$, stored in upper- or lower-packed storage mode, is factored using the Bunch-Kaufman diagonal pivoting method, where $A$ is expressed as:

$A = UDU^T$ or $A = LDL^T$

where:

$U$ is a product of permutation and unit upper triangular matrices.

$L$ is a product of permutation and unit lower triangular matrices.

$D$ is a symmetric block diagonal matrix, consisting of $1 \times 1$ and $2 \times 2$ diagonal blocks.

To solve a system of equations with one or more right-hand sides, follow the call to this subroutine with one or more calls to DBSTRS.

Table 154. Data Types

<table>
<thead>
<tr>
<th>$A$</th>
<th>$ipvt$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long-precision real</td>
<td>Integer</td>
<td>DBSTRF</td>
</tr>
</tbody>
</table>

Note: The output from DBSTRF should be used only as input to DBSTRS, for performing a solve.

Syntax

Fortran

CALL DBSTRF (uplo, n, ap, ipvt, nsinfo)

C and C++

dbstrf (uplo, n, ap, ipvt, nsinfo);

On Entry

$uplo$

indicates whether matrix $A$ is stored in upper- or lower-packed storage mode, where:

If $uplo = 'U'$, $A$ is stored in upper-packed storage mode.

If $uplo = 'L'$, $A$ is stored in lower-packed storage mode.

Specified as: a single character. It must be 'U' or 'L'.

$n$

is the order $n$ of matrix $A$.

Specified as: an integer; $n \geq 0$.

$ap$

is array, referred to as $AP$, in which matrix $A$, to be factored, is stored in upper- or lower-packed storage mode.

Specified as: a one-dimensional array of length $n$ $nsinfo$, containing numbers of the data type indicated in Table 154. See "Notes " on page 675.

$ipvt$

See On Return
\( nsinfo \)

is the number of elements in array, \( AP \).

If \( n \leq nco \), \( nsinfo = n(n + 1) / 2 \)

Where:

\( nco = \sqrt{2.0(ics)} \)

\( ics \) is the size in doublewords of the data cache. The data cache size can be obtained by utilizing the following C language code fragment:

```c
#include <sys/systemcfg.h>
int ics;
    .
    .
ics = system_configuration.dcache_size/8;
```

\( ics \) is the size in doublewords of the data cache. The data cache size can be obtained by utilizing the following C language code fragment:

```c
#include <sys/systemcfg.h>
int ics;
    .
    .
ics = system_configuration.dcache_size/8;
```

Otherwise, to determine a sufficient amount of storage, use the following processor-independent formula:

\[
n0 = 100 \\
ns = (n + n0) (n + n0 + 1) / 2 + n(n0)
\]

For \( uplo = 'L' \),

\( nsinfo \geq ns \)

For \( uplo = 'U' \),

\[
  n1 = (n + 1) / 2 \\
nf = n((n + 1) / 2) \\
nf1 = n1(n1 + 1) \\
ns1 = nf + nf1 \\
nsinfo \geq \max(ns, ns1)
\]

To determine the minimal amount of storage see "Notes" on page 675.

Specified as: an integer; \( nsinfo > 0 \).

On Return

\( ap \) is the transformed matrix \( A \) of order \( n \), containing the results of the factorization.

If \( nsinfo \geq 0 \) and \( n > nco \), additional information that can be used to obtain a minimum \( nsinfo \) is stored in \( AP(1) \). See "Notes" on page 675 and "Function" on page 675.

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 154 on page 673.
ipvt

is an integer vector of length \( n \), containing the pivot information necessary to construct the factored form of \( A \).

Returned as: a one-dimensional array of (at least) length \( n \), containing numbers of the data type indicated in Table 154 on page 673.

nsinfo

indicates the result of the computation.

- If \( nsinfo = 0 \), the factorization completed successfully.
- If \( nsinfo > 0 \), factorization was unsuccessful and \( nsinfo \) is set to \( i \) where \( d_{ii} \) is exactly zero.
- If \( nsinfo < 0 \), factorization did not take place and the array \( AP \) remains unchanged. \(|nsinfo|\) is the minimal storage required for factorization to take place. Error message 2200 is issued and execution terminates, unless you have used ERRSET to make error code 2200 recoverable. See “What Can You Do about ESSL Input-Argument Errors?” on page 67.

Specified as: an integer.

Notes

1. This subroutine accepts lowercase letters for the uplo argument.
2. In your C program, argument \( nsinfo \) must be passed by reference.
3. In the input array specified for \( ap \), the first \( n(n+1)/2 \) elements are matrix elements. The additional locations, required in the array, are used for working storage.
4. The array specified for \( ap \) should not be altered between calls to the factorization and solve subroutines; otherwise, unpredictable results may occur.
5. On return, if \( nsinfo \geq 0 \) and \( n > nco \), \( ap \) contains additional information in \( AP(1) \) that can be used to obtain the minimal required \( nsinfo \). This information can be accessed using the following code fragment:

   ```c
   REAL*8 AP(NSINFO)
   INTEGER API(2)
   EQUIVALENCE(API, AP)
   ...
   NSINFOMIN = API(2)
   ```

6. For a description of how a symmetric matrix is stored in upper- or lower-packed storage mode in an array, see “Symmetric Matrix” on page 85.

Function

where:

- \( U \) is a product of permutation and unit upper triangular matrices.
- \( L \) is a product of permutation and unit lower triangular matrices.
- \( D \) is a symmetric block diagonal matrix, consisting of \( 1 \times 1 \) and \( 2 \times 2 \) diagonal blocks.

If \( n \) is 0, no computation is performed and the subroutine returns after doing some parameter checking.

See references [8 on page 1363] and [76 on page 1368].
Error conditions

Resource Errors
None.

Computational Errors
Matrix A is singular.
- The factorization completed but the block diagonal matrix D is exactly singular. nsinfo is set to i, where d_{ii} is exactly zero. This 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 2147 is set to be unlimited in the ESSL error option table. For details, see “What Can You Do about ESSL Computational Errors?” on page 68.

Input-Argument Errors
1. uplo ≠ ‘U’ or ‘L’
2. n < 0
3. nsinfo < (minimum value).
   - For the minimum value, see the nsinfo argument description.
   - Return code 1 is returned if error 2200 is recoverable.

Examples

Example 1
This example shows a factorization of a symmetric indefinite matrix A of order 8, stored in lower-packed storage mode, where on input matrix A is:

\[
\begin{bmatrix}
3.0 & 5.0 & -2.0 & 2.0 & 3.0 & -5.0 & -2.0 & -3.0 \\
5.0 & 3.0 & 2.0 & -2.0 & 5.0 & -3.0 & 2.0 & -5.0 \\
-2.0 & 2.0 & 0.0 & 0.0 & -2.0 & 2.0 & 0.0 & 6.0 \\
2.0 & -2.0 & 0.0 & 8.0 & -6.0 & -10.0 & -8.0 & -14.0 \\
3.0 & 5.0 & -2.0 & -6.0 & 12.0 & 6.0 & 8.0 & 6.0 \\
-5.0 & -3.0 & 2.0 & -10.0 & 6.0 & 16.0 & 8.0 & 20.0 \\
-2.0 & 2.0 & 0.0 & -8.0 & 8.0 & 8.0 & 6.0 & 18.0 \\
-3.0 & -5.0 & 6.0 & -14.0 & 6.0 & 20.0 & 18.0 & 34.0 \\
\end{bmatrix}
\]

Note: The AP array is formatted in a triangular arrangement for readability; however, it is stored in lower-packed storage mode.

Call Statement and Input:

```
CALL DBSTRF ('L', 8, AP, IPVT, 36 )
```

AP = ( 3.0, 5.0, -2.0, 2.0, 3.0, -5.0, -2.0, -3.0, 3.0, 2.0, -2.0, 5.0, -3.0, 2.0, -5.0, 0.0, 0.0, -2.0, 2.0, 0.0, 0.0, 6.0, 8.0, -6.0, -10.0, -8.0, -14.0, 12.0, 6.0, 8.0, 6.0, 16.0, 8.0, 20.0, 6.0, 18.0, 34.0 )

Output:

```
NSINFO = 0
```

Note: AP and IPVT are stored in an internal format and must be passed unchanged to the solve subroutine.
Example 2

This example shows a factorization of a symmetric indefinite matrix $A$ of order 8, stored in upper-packed storage mode, where on input matrix $A$ is:

$$
\begin{bmatrix}
34.0 & 18.0 & 17.0 & 6.0 & -14.0 & 6.0 & -5.0 & -3.0 \\
18.0 & 6.0 & 6.0 & 8.0 & -8.0 & 0.0 & 2.0 & -2.0 \\
17.0 & 6.0 & 9.0 & 9.0 & -8.0 & 0.0 & 2.0 & -2.0 \\
6.0 & 8.0 & 9.0 & 12.0 & -6.0 & -2.0 & 5.0 & 3.0 \\
-14.0 & -8.0 & -8.0 & -6.0 & 8.0 & 0.0 & -2.0 & 2.0 \\
6.0 & 0.0 & 0.0 & -2.0 & 0.0 & 0.0 & 2.0 & -2.0 \\
-5.0 & 2.0 & 2.0 & 5.0 & -2.0 & 2.0 & 3.0 & 5.0 \\
-3.0 & -2.0 & -2.0 & 3.0 & 2.0 & -2.0 & 5.0 & 3.0
\end{bmatrix}
$$

Note: The $AP$ array is formatted in a triangular arrangement for readability; however, it is stored in upper-packed storage mode.

Call Statement and Input:

```
CALL DBSTRF ('U', 8, AP, IPVT, 36 )
AP = ( 34.0, 18.0, 6.0, 
       17.0, 6.0, 9.0, 
       6.0, 8.0, 9.0, 12.0, 
       -14.0, -8.0, -8.0, -6.0, 8.0, 
       6.0, 0.0, 0.0, -2.0, 0.0, 0.0, 
       -5.0, 2.0, 2.0, 5.0, -2.0, 2.0, 3.0, 
       -3.0, -2.0, -2.0, 3.0, 2.0, -2.0, 5.0, 3.0 )
```

Output:

NSINFO = 0

Note: $AP$ and $IPVT$ are stored in an internal format and must be passed unchanged to the solve subroutine.
DBSTRS (Symmetric Indefinite Matrix Multiple Right-Hand Side Solve)

Purpose

The DBSTRS subroutine solves a system of linear equations $AX = B$ for $X$, where $A$ is a real symmetric indefinite matrix, and $X$ and $B$ are real general matrices. This subroutine uses the results of the factorization of matrix $A$, produced by a preceding call to DBSTRF.

Table 155. Data Types

<table>
<thead>
<tr>
<th>A, B</th>
<th>ipvt</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long-precision real</td>
<td>Integer</td>
<td>DBSTRS</td>
</tr>
</tbody>
</table>

Note: The input to this solve subroutine must be the output from the factorization subroutine DBSTRF.

Syntax

Fortran

CALL DBSTRS (uplo, n, nrhs, ap, ipvt, bx, ldb, info)

C and C++

dbstrs (uplo, n, nrhs, ap, ipvt, bx, ldb, info);

On Entry

$uplo$

indicates whether original matrix $A$ is stored in upper- or lower-packed storage mode, where:

If $uplo = 'U'$, $A$ is stored in upper-packed storage mode.

If $uplo = 'L$', $A$ is stored in lower-packed storage mode.

Specified as: a single character. It must be 'U' or 'L'.

$n$

is the order $n$ of factored matrix $A$ and the number of rows of matrix $B$.

Specified as: an integer; $n \geq 0$.

$nrhs$

is the number of right-hand sides; i.e., the number of columns of matrix $B$.

Specified as: an integer; $nrhs \geq 0$.

$ap$

is the factored matrix $A$ produced by a preceding call to DBSTRF.

Specified as: a one-dimensional array of length $nsinfo$, containing numbers of the data type indicated in Table 155. See “Notes” on page 679 and “DBSTRF (Symmetric Indefinite Matrix Factorization)” on page 673.

$ipvt$

is an integer vector of length $n$, containing the pivot information necessary to construct the factored form of $A$, produced by a preceding call to DBSTRF.

Specified as: a one-dimensional array of (at least) length $n$, containing numbers of the data type indicated in Table 155. See “Notes” on page 679.

$bx$

is the general 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 155.
**ldb**

is the leading dimension of the array specified for *B*.

Specified as: an integer; *ldb* > 0 and *ldb* ≥ *n*.

**info**

See [On Return](#).

**On Return**

*bx* is the matrix *X*, containing the *nrhs* solutions to the system. The solutions, each of length *n*, reside in the columns of *X*.

Returned as: an *ldb* by (at least) *nrhs* array, containing numbers of the data type indicated in [Table 155 on page 678](#).

**info** indicates the result of the computation.

- If *info* = 0, the subroutine completed successfully.

Returned as: an integer.

**Notes**

1. This subroutine accepts lowercase letters for the *uplo* argument.
2. In your C program, argument *info* must be passed by reference.
3. The array data specified for input arguments *ap* and *ipvt* for this subroutine must be the same as the corresponding output arguments for DBSTRF.
4. The scalar data specified for input arguments *uplo* and *n* must be the same as that specified for DBSTRF.
5. The vectors and matrices used in this computation must have no common elements; otherwise, results are unpredictable. See “Concepts” on page 75.
6. For a description of how a symmetric matrix is stored in upper- or lower-packed storage mode in an array, see “Symmetric Matrix” on page 85.
7. To solve *AX* = *B* for *X*, where *B* and *X* are *n* by *nrhs* matrices, precede the call to DBSTRS with a call to DBSTRF.

**Function**

The system *AX* = *B* is solved for *X*, where *A* is a real symmetric indefinite matrix, and *X* and *B* are real general matrices. This subroutine uses the results of the factorization of matrix *A*, produced by a preceding call to DBSTRF.

If *n* or *nrhs* is 0, no computation is performed and the subroutine returns after doing some parameter checking.

See references [8 on page 1363](#) and [76 on page 1368](#).

**Error conditions**

**Resource Errors**

None.

**Computational Errors**

None.

- **Note:** If the factorization performed by DBSTRF failed because 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. uplo ≠ 'U' or 'L'
2. n < 0
3. nrhs < 0
4. n > ldb
5. ldb ≤ 0

Examples

Example 1

This example shows how to solve the system \(AX = B\), for three right-hand sides, where matrix \(A\) is the same matrix factored in the [Example 1](#) for DBSTRF.

Call Statement and Input:

```
UPLO  N  NRHS  AP  IPVT  BX  LDB  INFO
|   |   |   |   |   |   |   |
CALL DBSTRS ( 'L', 8, 3, AP, IPVT, BX, 8, INFO )
```

\(AP\) = (for this subroutine must be the same as the corresponding output argument for DBSTRF. See [Example 1](#) for DBSTRF.)

\(IPVT\) = (for this subroutine must be the same as the corresponding output argument for DBSTRF. See [Example 1](#) for DBSTRF.)

\[
\begin{bmatrix}
1.0 & -38.0 & 47.0 \\
7.0 & -10.0 & 73.0 \\
6.0 & 52.0 & 2.0 \\
\end{bmatrix}
\]

\(BX\) =

\[
\begin{bmatrix}
-30.0 & -228.0 & -42.0 \\
32.0 & 183.0 & 105.0 \\
34.0 & 297.0 & 9.0 \\
32.0 & 244.0 & 44.0 \\
62.0 & 497.0 & 61.0 \\
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
1.0 & 1.0 & 8.0 \\
1.0 & 2.0 & 7.0 \\
1.0 & 3.0 & 6.0 \\
\end{bmatrix}
\]

\(BX\) =

\[
\begin{bmatrix}
1.0 & 4.0 & 5.0 \\
1.0 & 5.0 & 4.0 \\
1.0 & 6.0 & 3.0 \\
1.0 & 7.0 & 2.0 \\
1.0 & 8.0 & 1.0 \\
\end{bmatrix}
\]

INFO = 0

Example 2

This example shows how to solve the system \(AX = B\), for three right-hand sides, where matrix \(A\) is the same matrix factored in the [Example 2](#) for DBSTRF.

Call Statement and Input:

```
UPLO  N  NRHS  AP  IPVT  BX  LDB  INFO
|   |   |   |   |   |   |   |
CALL DBSTRS ( 'U', 8, 3, AP, IPVT, BX, 8, INFO )
```

680 ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
AP = (for this subroutine must be the same as the corresponding output argument for DBSTRF. See [Example 2] for DBSTRF.)

IPVT = (for this subroutine must be the same as the corresponding output argument for DBSTRF. See [Example 2] for DBSTRF.)

\[
\begin{bmatrix}
59.0 & 52.0 & 479.0 \\
30.0 & 38.0 & 232.0 \\
33.0 & 50.0 & 247.0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
35.0 & 114.0 & 201.0 \\
-28.0 & -36.0 & -216.0 \\
4.0 & -4.0 & 40.0 \\
12.0 & 88.0 & 20.0 \\
4.0 & 56.0 & -20.0 \\
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
1.0 & 1.0 & 8.0 \\
1.0 & 2.0 & 7.0 \\
1.0 & 3.0 & 6.0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
1.0 & 4.0 & 5.0 \\
1.0 & 5.0 & 4.0 \\
1.0 & 6.0 & 3.0 \\
1.0 & 7.0 & 2.0 \\
1.0 & 8.0 & 1.0 \\
\end{bmatrix}
\]

INFO = 0
STRTRI, DTRTRI, CTRTRI, ZTRTRI, STPTRI, DTPTRI, CTPTRI, and ZTPTRI (Triangular Matrix Inverse)

Purpose

These subroutines find the inverse of triangular matrix $A$:

$$A + A^{-1}$$

Matrix $A$ can be either upper or lower triangular, where:

- For STRTRI, DTRTRI, CTRTRI, and ZTRTRI, it is stored in upper- or lower-triangular storage mode.
- For STPTRI, DTPTRI, CTPTRI, and ZTPTRI, it is stored in upper- or lower-triangular-packed storage mode.

<table>
<thead>
<tr>
<th>Table 156. Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
<tr>
<td>Short-precision complex</td>
</tr>
<tr>
<td>Long-precision complex</td>
</tr>
<tr>
<td>$^\dagger$LAPACK</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL STRTRI</th>
<th>DTRTRI</th>
<th>CTRTRI</th>
<th>ZTRTRI (uplo, diag, n, a, lda, info)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>strtri</td>
<td>dttri</td>
<td>ctrtri</td>
<td>ztrtri (uplo, diag, n, a, lda, info);</td>
</tr>
<tr>
<td>LAPACK</td>
<td>info = LAPACK_strtri</td>
<td>LAPACK_dttri</td>
<td>LAPACK_ctrtri</td>
<td>LAPACK_ztrtri (matrix_layout, uplo, diag, n, a, lda);</td>
</tr>
<tr>
<td>LAPACK</td>
<td>info = LAPACK_stptri</td>
<td>LAPACK_dtptri</td>
<td>LAPACK_ctptri</td>
<td>LAPACK_ztptri (matrix_layout, uplo, diag, n, ap);</td>
</tr>
</tbody>
</table>

On Entry

- $\text{matrix\_layout}$ indicates whether the input and output matrices are stored in row major order or column major order, where:
  - If $\text{matrix\_layout} = \text{LAPACK\_ROW\_MAJOR}$, the matrices are stored in row major order.
  - If $\text{matrix\_layout} = \text{LAPACK\_COL\_MAJOR}$, the matrices are stored in column major order.
  Specified as: an integer. It must be LAPACK\_ROW\_MAJOR or LAPACK\_COL\_MAJOR

- $\text{uplo}$ indicates whether matrix $A$ is an upper or lower triangular matrix, where:
  - If $\text{uplo} = 'U'$, $A$ 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'.

\textit{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 156 on page 682.

\( lda \) is the leading dimension of the arrays specified for \( a \).

Specified as: an 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 156 on page 682.

\( n \) is the order of matrix \( A \).

Specified as: an integer; \( n \geq 0 \).

\textit{info} has the following meaning:

- If \( info = 0 \), the inverse completed successfully.
- If \( info > 0 \), \( info \) is set equal to the first \( i \) where \( A_{ii} \) is zero. Matrix \( A \) is singular and its inverse could not be computed.

Returned as:

- For STRTRI, DTRTRI, CTRTRI, ZTRTRI, STPTRI, DTPTRI, CTPTRI, and ZTPTRI, returned as: an integer; \( info \geq 0 \).
- For LAPACKE_strtri, LAPACKE_dtrtri, LAPACKE_ctrtri, LAPACKE_ztrtri, LAPACKE_stptri, LAPACKE_dtptri, LAPACKE_ctptri, and LAPACKE_ztptri, returned as an integer function value; \( info \geq 0 \).

\textbf{Notes}

1. In C programs, the argument \( info \) must be passed by reference.
2. These subroutines accept lowercase letters for the \( uplo \) and \( diag \) arguments.
3. 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.

4. The elements of the diagonal of a unit triangular matrix are always one, so you do not need to set these values.

5. The way _TRTRI and _TPTRI subroutines handle computational errors differs from LAPACK. Like LAPACK, these subroutines use the $info$ argument to provide information about the computational error, but they also provide an error message.

6. On both input and output, matrix $A$ conforms to LAPACK format.

7. 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 93.

**Function**

These subroutines find the inverse of triangular matrix $A$, where $A$ is either upper or lower triangular:

$$A \leftarrow A^{-1}$$

where:

- $A$ is the triangular matrix of order $n$.
- $A^{-1}$ is the inverse of the triangular matrix of order $n$.

If $n$ is 0, no computation is performed. See references on page 1363 and on page 1366.

**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 and returned in the argument $info$.
- 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. $uplo \neq 'U$' or 'L'
2. $diag \neq 'U$' or 'N'
3. $n < 0$
4. $lda \leq 0$
5. $lda < n$

**Examples**

**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:

\[
\begin{bmatrix}
1.00 & 3.00 & 4.00 & 5.00 & 6.00 \\
0.00 & 2.00 & 8.00 & 9.00 & 1.00 \\
0.00 & 0.00 & 4.00 & 8.00 & 4.00 \\
0.00 & 0.00 & 0.00 & -2.00 & 6.00 \\
0.00 & 0.00 & 0.00 & 0.00 & -1.00 \\
\end{bmatrix}
\]

Matrix $A^{-1}$ is:

\[
\begin{bmatrix}
1.00 & -1.50 & 2.00 & 3.75 & 35.00 \\
0.00 & 0.50 & -1.00 & -1.75 & -14.00 \\
0.00 & 0.00 & 0.25 & 1.00 & 7.00 \\
0.00 & 0.00 & 0.00 & -0.50 & -3.00 \\
0.00 & 0.00 & 0.00 & 0.00 & -1.00 \\
\end{bmatrix}
\]

Call Statement and Input:

```
UPLO DIAG N A LDA INFO
|    |    |    |    |    |
CALL STRTRI( 'U', 'N', 5, A, 5, INFO )
```

\[
A = \begin{bmatrix}
1.00 & 3.00 & 4.00 & 5.00 & 6.00 \\
.  & 2.00 & 8.00 & 9.00 & 1.00 \\
.  & .  & 4.00 & 8.00 & 4.00 \\
.  & .  & .  & -2.00 & 6.00 \\
.  & .  & .  & .  & -1.00 \\
\end{bmatrix}
\]

Output:

\[
A = \begin{bmatrix}
1.00 & -1.50 & 2.00 & 3.75 & 35.00 \\
.  & 0.50 & -1.00 & -1.75 & -14.00 \\
.  & .  & 0.25 & 1.00 & 7.00 \\
.  & .  & .  & -0.50 & -3.00 \\
.  & .  & .  & .  & -1.00 \\
\end{bmatrix}
\]

INFO = 0

**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:

\[
\begin{bmatrix}
1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
3.0 & 1.0 & 0.0 & 0.0 & 0.0 \\
4.0 & 8.0 & 1.0 & 0.0 & 0.0 \\
5.0 & 9.0 & 8.0 & 1.0 & 0.0 \\
6.0 & 1.0 & 4.0 & 6.0 & 1.0 \\
\end{bmatrix}
\]

Matrix $A^{-1}$ is:

\[
\begin{bmatrix}
1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
-3.0 & 1.0 & 0.0 & 0.0 & 0.0 \\
20.0 & -8.0 & 1.0 & 0.0 & 0.0 \\
\end{bmatrix}
\]
Note: Because matrix $A$ is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of one for the diagonal elements.

Call Statement and Input:

```
CALL STRTRI( 'L', 'U', 5, A, 5, INFO )
```

```
A =

\[
\begin{bmatrix}
3.0 & \cdot & \cdot & \cdot & \cdot \\
4.0 & 8.0 & \cdot & \cdot & \cdot \\
5.0 & 9.0 & 8.0 & \cdot & \cdot \\
6.0 & 1.0 & 4.0 & 6.0 & \cdot \\
\end{bmatrix}
\]
```

Output:

```
A =

\[
\begin{bmatrix}
-3.0 & \cdot & \cdot & \cdot & \cdot \\
20.0 & -8.0 & \cdot & \cdot & \cdot \\
-138.0 & 55.0 & -8.0 & \cdot & \cdot \\
745.0 & -299.0 & 44.0 & -6.0 & \cdot \\
\end{bmatrix}
\]
```

INFO = 0

Example 3

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:

```
\[
\begin{bmatrix}
(-4.00, 1.00) & (4.00, -3.00) & (-1.00, -3.00) & (0.00, 0.00) & (-1.00, 0.00) \\
(0.00, 0.00) & (-2.00, 0.00) & (-3.00, -1.00) & (-2.00, -1.00) & (4.00, 3.00) \\
(0.00, 0.00) & (0.00, 0.00) & (-5.00, 3.00) & (-3.00, -3.00) & (-5.00, -5.00) \\
(0.00, 0.00) & (0.00, 0.00) & (0.00, 0.00) & (4.00, -4.00) & (2.00, 0.00) \\
(0.00, 0.00) & (0.00, 0.00) & (0.00, 0.00) & (0.00, 0.00) & (2.00, -1.00) \\
\end{bmatrix}
\]
```

Matrix $A^{-1}$ is:

```
\[
\begin{bmatrix}
(-0.24, -0.06) & (-0.56, 0.24) & (0.41, 0.09) & (-0.22, 0.13) & (1.32, 2.12) \\
(0.00, 0.00) & (-0.50, 0.00) & (0.18, 0.21) & (-0.22, -0.06) & (0.21, 1.87) \\
(0.00, 0.00) & (0.00, 0.00) & (-0.15, -0.09) & (0.07, -0.11) & (0.02, -0.47) \\
(0.00, 0.00) & (0.00, 0.00) & (0.00, 0.00) & (0.12, 0.12) & (-0.05, -0.15) \\
(0.00, 0.00) & (0.00, 0.00) & (0.00, 0.00) & (0.00, 0.00) & (0.40, 0.20) \\
\end{bmatrix}
\]
```

Call Statement and Input:

```
CALL ZTRTRI( 'U', 'N', 5, A, 5, INFO )
```
Example 4

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:

$$
A = \begin{bmatrix}
(-4.00, 1.00) & (4.00, -3.00) & (-1.00, 3.00) & (0.00, 0.00) & (-1.00, 0.00) \\
(-2.00, 0.00) & (-3.00, -1.00) & (2.00, -1.00) & (4.00, 3.00) \\
(-5.00, 3.00) & (3.00, -3.00) & (-5.00, -5.00) & & \\
(4.00, -4.00) & 2.00, 0.00 & & & \\
\end{bmatrix}
$$

Output:

$$
A^{-1} = \begin{bmatrix}
(-0.24, -0.06) & (-0.56, 0.24) & (0.41, 0.09) & (-0.22, 0.13) & (1.32, 2.12) \\
(-0.50, 0.00) & (0.18, 0.21) & (-0.22, -0.06) & (0.21, 1.87) \\
(-0.15, -0.09) & (0.07, -0.11) & (0.02, -0.47) & & \\
(0.12, 0.12) & (-0.05, -0.15) & (0.40, 0.20) & & \\
\end{bmatrix}
$$

INFO = 0

Note: Because matrix $A$ is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of one for the diagonal elements.

Call Statement and Input:

```call
CALL ZTRTRI('L', 'U', 5, A, 5, INFO)
```

Matrix $A^{-1}$ is:

$$
A^{-1} = \begin{bmatrix}
(1.00, 0.00) & (0.00, 0.00) & (0.00, 0.00) & (0.00, 0.00) & (0.00, 0.00) \\
(4.00, -3.00) & (1.00, 0.00) & (0.00, 0.00) & (0.00, 0.00) & (0.00, 0.00) \\
(-1.00, 3.00) & (-3.00, -1.00) & (1.00, 0.00) & (0.00, 0.00) & (0.00, 0.00) \\
(0.00, 0.00) & (-2.00, -1.00) & (-3.00, -3.00) & (1.00, 0.00) & (0.00, 0.00) \\
(-1.00, 0.00) & (4.00, 3.00) & (-5.00, -5.00) & (2.00, 0.00) & (1.00, 0.00) \\
\end{bmatrix}
$$
Output:

\[
A = \begin{bmatrix}
-4.00 & 3.00 & . & . & . \\
-14.00 & 2.00 & 3.00 & 1.00 & . \\
-59.00 & -34.00 & 8.00 & 13.00 & 3.00 & 3.00 \\
64.00 & 8.00 & -10.00 & -9.00 & -1.00 & -1.00 & -2.00 & 0.00
\end{bmatrix}
\]

\[
\text{INFO} = 0
\]

Example 5

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:

```fortran
CALL STPTRI( 'U', 'N', 5, AP, INFO )
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:

\[
AP = (1.00, -1.50, 0.50, 2.00, -1.00, 0.25, 3.75, -1.75, 1.00, -0.50, 35.00, -14.00, 7.00, -3.00, -1.00)
\]

\[
\text{INFO} = 0
\]

Example 6

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 one for the diagonal elements.

Call Statement and Input:

```fortran
CALL STPTRI( 'L', 'U', N, AP, INFO )
AP = (. , 3.0, 4.0, 5.0, 6.0, . , 8.0, 9.0, 1.0, . , 8.0, 4.0, . , 6.0, . )
```

Output:

\[
AP = (. , -3.0, 20.0, -138.0, 745.0, . , -8.0, 55.0, -299.0, . , -8.0, 44.0, . , -6.0, . )
\]

\[
\text{INFO} = 0
\]

Example 7

This example shows how the inverse of matrix \( A \) is computed, where \( A \) is the same matrix shown in Example 3 and is stored in upper-triangular-packed storage mode. The inverse matrix computed here is the same as the inverse matrix shown in Example 3 and is stored in upper-triangular-packed storage mode.
Example 8

This example shows how the inverse of matrix $A$ is computed, where $A$ is the same matrix shown in Example 4 and is stored in lower-triangular-packed storage mode. The inverse matrix computed here is the same as the inverse matrix shown in Example 4 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 one for the diagonal elements.
SLANTR, DLANTR, CLANTR, ZLANTR, SLANTP, DLANTP, CLANTP, and ZLANTP (Trapezoidal or Triangular Matrix Norm)

Purpose

These subprograms compute the norm of matrix $A$ as explained below:

SLANTR, DLANTR, CLANTR, and ZLANTR

These subprograms compute the norm of trapezoidal matrix $A$ stored in upper- or lower-trapezoidal storage mode.

SLANTP, DLANTP, CLANTP, and ZLANTP

These subroutines compute the norm of triangular matrix $A$, stored in upper- or lower-triangular-packed storage mode.

Table 157. Data Types

<table>
<thead>
<tr>
<th>$A$</th>
<th>work, Result</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SLANTR, ZLANTR</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DLANTR, DLANTP</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>CLANTR, ZLANTP</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>ZLANTR</td>
</tr>
</tbody>
</table>

Syntax

Fortran

SLANTR | DLANTR | CLANTR | ZLANTR (norm, uplo, diag, m, n, a, lda, work)
SLANTP | DLANTP | CLANTP | ZLANTP (norm, uplo, diag, m, n, ap, work)

C and C++

slantr | dlantr | clantr | zlantr (norm, uplo, diag, m, n, a, lda, work);
slantp | dlantp | clantp | zlantp (norm, uplo, diag, n, ap, work);

LAPACK

LAPACK_slantr | LAPACK_dlantr | LAPACK_clantr | LAPACK_zlantr (matrix_layout, norm, uplo, diag, m, n, a, lda);
LAPACK_slantp | LAPACK_dlantp | LAPACK_clantp | LAPACK_zlantp (matrix_layout, norm, uplo, diag, n, ap);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If matrix_layout = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If matrix_layout = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. It must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR

norm

specifies the type of computation, where:

If norm = 'O' or '1', the one norm of $A$ is computed.
If norm = 'I', the infinity norm of $A$ is computed.
If norm = 'F' or 'E', the Frobenius or Euclidean norm of $A$ is computed.
If \( \text{norm} = 'M' \), the absolute value of the matrix element having the largest absolute value, i.e., \( \max (|A|) \), is returned.

Specified as: a single character; \( \text{norm} = 'O', '1', 'I', 'F', 'E', \) or 'M'.

\( \text{uplo} \) indicates the storage mode used for matrix \( A \), where:

For \( \text{SLANTR}, \text{DLANTR}, \text{CLANTR}, \) and \( \text{ZLANTR} \)
- If \( \text{uplo} = 'U' \), \( A \) is stored in upper-trapezoidal storage mode.
- If \( \text{uplo} = 'L' \), \( A \) is stored in lower-trapezoidal storage mode.

For \( \text{SLANTP}, \text{DLANTP}, \text{CLANTP}, \) and \( \text{ZLANTP} \)
- If \( \text{uplo} = 'U' \), \( A \) is stored in upper-triangular-packed storage mode.
- If \( \text{uplo} = 'L' \), \( A \) is stored in lower-triangular-packed storage mode.

Specified as: a single character. It must be 'U' or 'L'.

\( \text{diag} \) indicates the characteristics of the diagonal of matrix \( A \), where:

For \( \text{SLANTR}, \text{DLANTR}, \text{CLANTR}, \) and \( \text{ZLANTR} \)
- If \( \text{diag} = 'U' \), \( A \) is a unit trapezoidal matrix.
- If \( \text{diag} = 'N' \), \( A \) is not a unit trapezoidal matrix.

For \( \text{SLANTP}, \text{DLANTP}, \text{CLANTP}, \) and \( \text{ZLANTP} \)
- If \( \text{diag} = 'U' \), \( A \) is a unit triangular matrix.
- If \( \text{diag} = 'N' \), \( A \) is not a unit triangular matrix.

Specified as: a single character. It must be 'U' or 'N'.

\( m \) is the number of rows in trapezoidal matrix \( A \).

Specified as: an integer; \( m \geq 0 \).

\( n \)

For \( \text{SLANTR}, \text{DLANTR}, \text{CLANTR}, \) and \( \text{ZLANTR} \)
- \( n \) is the number of columns in matrix \( A \).

For \( \text{SLANTP}, \text{DLANTP}, \text{CLANTP}, \) and \( \text{ZLANTP} \)
- \( n \) is the order of matrix \( A \)

Specified as: an integer; \( n \geq 0 \).

\( \text{ap} \) is the matrix \( A \) of order \( n \), stored in upper- or lower-triangular-packed storage mode.

Specified as: a one-dimensional array of (at least) \( n(n+1)/2 \), containing numbers of the data type indicated in \( \text{Table 157 on page 690} \).

\( a \) is the trapezoidal matrix \( A \), stored in upper- or lower-trapezoidal storage mode.

Specified as: an \( lda \) by (at least) \( n \) array, containing numbers of the data type indicated in \( \text{Table 157 on page 690} \).

\( \text{lda} \) is the leading dimension of matrix \( A \).

Specified as: an integer; \( lda \geq m \).

\( \text{work} \) is the work area used by this subroutine, where:
When $\text{norm} = 'I'$, '1', or 'O', $\text{work}$ is (at least) of length:
- $m$ for SLANTR, DLANTR, CLANTR, and ZLANTR
- $n$ for SLANTP, DLANTP, CLANTP, and ZLANTP

Otherwise, $\text{work}$ is not referenced.

Specified as: an area of storage containing numbers of data type indicated in Table 157 on page 690

**On Return**

**Function value**

is the result of the norm computation, returned as a number of the data type indicated in Table 157 on page 690.

If $\text{norm} = 'O'$ or '1', the one norm of $A$ is returned.

If $\text{norm} = 'T'$, the infinity norm of $A$ is returned.

If $\text{norm} = 'F'$ or 'E', the Frobenius or Euclidean norm of $A$ is returned.

If $\text{norm} = 'M'$, the absolute value of the matrix element having the largest absolute value, i.e., $\max (|A|)$, is returned.

If $m = 0$ or $n = 0$, the function returns zero.

**Notes**

1. Declare this function in your program as returning a value of the data type indicated in Table 157 on page 690.
2. This function accepts lowercase letters for the $\text{norm}$, $\text{uplo}$, and $\text{diag}$ arguments.
3. For a description of triangular matrices and how they are stored in upper- and lower-triangular-packed storage mode, see “Triangular Matrix” on page 93.
4. For a description of trapezoidal matrices and how they are stored in upper- and lower-trapezoidal storage mode, see “Trapezoidal Matrix” on page 96.
5. For SLANTR, DLANTR, CLANTR, and ZLANTR, the following cases are extensions to the LAPACK standard:
   - $\text{uplo} = 'U'$ and $m > n$
   - $\text{uplo} = 'L'$ and $n > m$

**Function**

One of the following computations is performed on matrix $A$, depending on the value specified for $\text{norm}$:

<table>
<thead>
<tr>
<th>Value specified for $\text{norm}$</th>
<th>Type of computation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>'O' or '1'</td>
<td>one norm</td>
</tr>
<tr>
<td>'T'</td>
<td>infinity norm</td>
</tr>
<tr>
<td>'F' or 'E'</td>
<td>Frobenius or Euclidean norm</td>
</tr>
<tr>
<td>'M'</td>
<td>absolute value of the matrix element having the largest absolute value, i.e., $\max (</td>
</tr>
</tbody>
</table>

If $m = 0$ or $n = 0$, the function returns zero.

**Error conditions**

**Resource Errors**

None.
Computational Errors
None.

Input-Argument Errors
1. norm ≠ 'O', '1', 'T', 'F', 'E', or 'M'
2. uplo ≠ 'U' or 'L'
3. diag ≠ 'U' or 'N'
4. m < 0
5. n < 0
6. lda < 1
7. lda < m

Examples

Example 1
This example computes the infinity norm of real trapezoidal matrix A stored in lower-trapezoidal storage mode.

Call Statements and Input:

```
NORM UPLO DIAG M N A LDA WORK
ANORM = DLANTR( 'I', 'L', 'N', 10, 9, A, 10, WORK )
```

```
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 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:

ANORM = 45.0

Example 2
This example computes the Frobenius norm of real trapezoidal matrix A stored upper-trapezoidal storage mode.

Call Statements and Input:

```
NORM UPLO DIAG M N A LDA WORK
ANORM = DLANTR( 'F', 'U', 'U', 9, 10, A, 9, WORK )
```

```
A =

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 3.0 3.0 3.0 3.0 3.0 3.0 3.0
. . . . 1.0 4.0 4.0 4.0 4.0 4.0 4.0 4.0
. . . . . 1.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0
. . . . . . 1.0 6.0 6.0 6.0 6.0 6.0 6.0 6.0
. . . . . . . 1.0 7.0 7.0 7.0 7.0 7.0 7.0 7.0
. . . . . . . . 1.0 8.0 8.0 8.0 8.0 8.0 8.0 8.0
. . . . . . . . . . 1.0 9.0
```

Output:

ANORM = 28.88
Example 3

This example computes the infinity norm of complex trapezoidal matrix $A$ stored in lower-trapezoidal storage mode.

Call Statements and Input:

```c
NORM UPLO DIAG M N A LDA WORK
ANORM = ZLANTR( 'I', 'L', 'N', 5, 4, A, 5, WORK )
```

```
A =
\[
\begin{pmatrix}
(1.0, 1.0) & . & . & . & . \\
(2.0, 1.0)(2.0, 2.0) & . & . & . & . \\
(3.0, 1.0)(3.0, 2.0)(3.0, 3.0) & . & . & . & . \\
(4.0, 1.0)(4.0, 2.0)(4.0, 3.0)(4.0, 4.0) & . & . & . & . \\
(5.0, 1.0)(5.0, 2.0)(5.0, 3.0)(5.0, 4.0) & . & . & . & . \\
\end{pmatrix}
\]
```

Output:

ANORM = 22.7

Example 4

This example computes the absolute value of the matrix element having the largest absolute value of complex trapezoidal matrix $A$ stored in upper-trapezoidal storage mode.

Call Statements and Input:

```c
NORM UPLO DIAG M N A LDA WORK
ANORM = ZLANTR( 'M', 'U', 'U', 4, 5, A, 4, WORK )
```

```
A =
\[
\begin{pmatrix}
(1.0, 0.0) & (1.0, 2.0)(1.0, 3.0) & (1.0, 4.0) & (1.0, 5.0) & . \\
. & (1.0, 0.0)(2.0, 3.0) & (2.0, 4.0) & (2.0, 5.0) & . \\
. & . & (1.0, 0.0)(3.0, 4.0) & (3.0, 5.0) & . \\
. & . & . & (1.0, 0.0)(4.0, 5.0) & . \\
\end{pmatrix}
\]
```

Output:

ANORM = 6.40

Example 5

This example computes the infinity norm of real triangular matrix $A$, stored in lower-triangular-packed storage mode.

Call Statements and Input:

```c
NORM UPLO DIAG N AP WORK
ANORM = DLANTP( 'I', 'L', 'N', 9, AP, WORK )
```

```
AP =
\[
\begin{pmatrix}
1.0 & . & . & . & . & . & . & . & . \\
1.0 & 2.0 & . & . & . & . & . & . & . \\
1.0 & 2.0 & 3.0 & . & . & . & . & . & . \\
1.0 & 2.0 & 3.0 & 4.0 & . & . & . & . & . \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & . & . & . & . \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & . & . & . \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & . & . \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 8.0 & . \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 8.0 & 9.0 \\
\end{pmatrix}
\]
```

Output:
**Example 6**

This example computes the Frobenius norm of real triangular matrix $A$, stored in upper-triangular-packed storage mode.

Call Statements and Input:

```
ANORM = DLANTP( 'F', 'U', 'U', 9, AP, WORK )
```

```
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
. . 1.0 3.0 3.0 3.0 3.0 3.0
. . . 1.0 4.0 4.0 4.0 4.0
. . . . 1.0 5.0 5.0 5.0
. . . . . 1.0 6.0 6.0 6.0
. . . . . . 1.0 7.0 7.0
. . . . . . . 1.0
```

Output:

`ANORM = 28.72`

**Example 7**

This example computes the infinity norm of complex triangular matrix $A$, stored in lower-triangular-packed storage mode.

Call Statements and Input:

```
ANORM = ZLANTP( 'I', 'L', 'N', 5, AP, WORK )
```

```
AP =
```

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

Output:

`ANORM = 21.2`

**Example 8**

This example computes the absolute value of the matrix element having the largest absolute value of complex triangular matrix $A$, stored in upper-triangular-packed storage mode.

Call Statements and Input:

```
ANORM = ZLANTP( 'M', 'U', 'U', 5, AP, WORK )
```

```
AP =
```

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

Chapter 10. Linear Algebraic Equations 695
Output:

ANORM = 5.65
Banded Linear Algebraic Equation Subroutines

This contains the banded linear algebraic equation subroutine descriptions.
SGBSV, DGBSV, CGBSV, and ZGBSV (General Band Matrix Factorization and Multiple Right-Hand Side Solve)

**Purpose**

These subroutines solve the general band system of linear equations $AX = B$ for $X$, where $A$ is a general band matrix and $B$ and $X$ are general matrices.

The matrix $A$ is stored in BLAS-general-band storage mode and is factored using Gaussian elimination with partial pivoting.

*Table 158. Data Types*

<table>
<thead>
<tr>
<th>$A, B$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGBSV</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGBSV</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGBSV</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGBSV</td>
</tr>
</tbody>
</table>

**Syntax**

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SGBSV</th>
<th>DGBSV</th>
<th>CGBSV</th>
<th>ZGBSV $(n, kl, ku, nrhs, a, lda, ipiv, b, ldb, info)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>sgbsv</td>
<td>dgbsv</td>
<td>cgbsv</td>
<td>zgbsv $(n, kl, ku, nrhs, a, lda, ipiv, b, ldb, info)$</td>
</tr>
<tr>
<td>LAPACK</td>
<td>info = LAPACKE_sgbsv</td>
<td>LAPACKE_dgbsv</td>
<td>LAPACKE_cgbsv</td>
<td>LAPACKE_zgbsv $(matrix_layout, n, kl, ku, nrhs, a, lda, ipiv, b, ldb)$</td>
</tr>
</tbody>
</table>

**On Entry**

- *matrix_layout* indicates whether the input and output matrices are stored in row major order or column major order, where:
  - If *matrix_layout* = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
  - If *matrix_layout* = LAPACK_COL_MAJOR, the matrices are stored in column major order.

  Specified as: an integer. It must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR

- $n$ is the order of matrix $A$ and the number of rows in matrix $B$.
  Specified as: an integer; $n \geq 0$.

- $kl$ is the lower band width $kl$ of the matrix $A$.
  Specified as: an integer; $kl \geq 0$.

- $ku$ is the upper band width $ku$ of the matrix $A$.
  Specified as: an integer; $ku \geq 0$.

- $nrhs$ is the number of right-hand sides; that is, the number of columns of matrix $B$.
  Specified as: an integer; $nrhs \geq 0$.

- $a$ is the general band matrix $A$ of order $n$. 
Specified as: an \( \text{lda} \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 158 on page 698.

\( \text{lda} \)

is the leading dimension of the array specified for \( a \).

Specified as: an integer; \( \text{lda} > 0 \) and \( \text{lda} \geq 2kl+ku+1 \).

\( \text{ipiv} \)

See "On Return".

\( b \)

is the general 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 \( \text{ldb} \) by (at least) \( nrhs \) array, containing numbers of the data type indicated in Table 158 on page 698.

\( \text{ldb} \)

is the leading dimension of the array specified for \( B \).

Specified as: an integer; \( \text{ldb} > 0 \) and \( \text{ldb} \geq n \).

\( \text{info} \)

See "On Return".

On Return

\( a \)

is the transformed matrix \( A \) of order \( n \) containing the results of the factorization. See "Function" on page 700.

Returned as: an \( \text{lda} \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 158 on page 698.

\( \text{ipiv} \)

is the integer vector of length \( n \), containing the pivot indices.

Returned as: a one-dimensional integer array of (at least) length \( n \), containing integers; \( 1 \leq \text{ipiv}_j \leq n \) for all \( j \).

\( b \)

If \( \text{info} = 0 \), \( b \) is the general matrix \( X \), containing the \( nrhs \) solutions to the system. The solutions, each of length \( n \), reside in the columns of matrix \( X \).

Returned as: an \( \text{ldb} \) by (at least) \( nrhs \) array, containing numbers of the data type indicated in Table 158 on page 698.

\( \text{info} \)

has the following meaning:

If \( \text{info} = 0 \), the subroutine completed successfully.

If \( \text{info} > 0 \), \( \text{info} \) is set to the first \( i \), where \( U_i \) is zero. The solution has not been computed.

Returned as:

- For SGBSV, DGBSV, CGBSV, and ZGBSV, returned as: an integer; \( \text{info} \geq 0 \).
- For LAPACKE_sgbsv, LAPACKE_dgbsv, LAPACKE_cgbsv, and LAPACKE_zgbsv, returned as an integer function value; \( \text{info} \geq 0 \).

Notes

1. In your C program, argument \( \text{info} \) must be passed by reference.
2. \( a, \text{ipiv}, \) and \( b \) must have no common elements; otherwise, results are unpredictable.
3. 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 100.
4. The way these subroutines handle singularity differs from LAPACK. Like LAPACK, these subroutines use the info argument to provide information about the singularity of A, but they also provide an error message.

**Function**

These subroutines solve the general band system of linear equations $AX = B$, where $A$ is a general band matrix and $B$ and $X$ are general matrices.

If $n$ is 0, no computation is performed and the subroutine returns after doing some parameter checking. If $n > 0$ and $nrhs$ is 0, no solutions are computed and the subroutine returns after factoring the matrix.

See references [8 on page 1363] and [46 on page 1366].

**Error conditions**

**Resource Errors**
- None

**Computational Errors**
- Matrix $A$ is singular or nearly singular.
  - The first column, $i$, of $L$ with a corresponding $U_{ii} = 0$ 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. $n < 0$
2. $kl < 0$
3. $ku < 0$
4. $nrhs < 0$
5. $lda \leq 0$
6. $2kl+ku+1 > lda$
7. $ldb \leq 0$
8. $n > ldb$

**Examples**

**Example 1**

This example shows how to solve the real general band system $AX=B$, where:

Matrix $A$ is the same used as input in **Example 1** for DGBTRF.
Matrix $B$ is the same used as input in **Example 1** for DGBTRS.

Call Statement and Input:

```
CALL DGBSV( N, KL, KU, NRHS, A, LDA, IPIV, B, LDB, INFO)
```

```
A = (same as input A in **Example 1**)
B = (same as input B in **Example 1**)
```

Output:
Example 2

This example shows how to solve the complex general band system $AX = B$, where:

Matrix $A$ is the same used as input in Example 2 for ZGBTRF.
Matrix $B$ is the same used as input in Example 3 for ZGBTRS.

Call Statement and Input:

```
CALL ZGBSV( N, KL, KU, NRHS, A, LDA, IPIV, B, LDB, INFO)
```

```
A = (same as input A in Example 2)
B = (same as input B in Example 3)
```

Output:

```
A =
[ 4.000 4.000 4.000 4.000 4.000 4.000 4.000 4.000 4.000 ]
[ 3.000 3.000 3.000 3.000 3.000 3.000 3.000 3.000 3.000 ]
[ 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 ]
[ 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000 ]
[ 3.000 3.000 3.000 3.000 -3.617 -5.095 3.000 -1.546 3.037 ]
[ 0.666 0.444 0.518 0.567 -0.334 0.326 0.043 -0.790 1.000 ]
[ 0.055 -0.111 0.592 0.246 -0.829 -0.588 -0.617 0.790 ]
```

```
IPIV = ( 3, 4, 5, 6, 9, 8, 9 )
```

```
B =
[ 0.629 1.149 4.713 ]
[ -0.025 1.870 -0.726 ]
[ 0.523 0.615 2.946 ]
[ -0.286 -1.434 -2.774 ]
[ -0.104 -0.524 -1.067 ]
[ -0.118 -0.591 -0.970 ]
[ 0.220 -0.896 2.027 ]
[ 0.043 -0.790 1.604 ]
[ 0.496 0.480 3.599 ]
```

```
INFO = 0
```
SGBTRF, DGBTRF, CGBTRF and ZGBTRF (General Band Matrix Factorization)

Purpose

These subroutines factor general band matrix $A$, stored in BLAS-general-band storage mode, 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 SGBTRS, DGBTRS, CGBTRS, and ZGBTRS respectively.

Table 159. Data Types

<table>
<thead>
<tr>
<th>$A$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGBTRF$^A$</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGBTRF$^A$</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGBTRF$^A$</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGBTRF$^A$</td>
</tr>
<tr>
<td>$^A$LAPACK</td>
<td></td>
</tr>
</tbody>
</table>

Note: The output from these factorization subroutines should be used only as input to the solve subroutines SGBTRS, DGBTRS, CGBTRS, and ZGBTRS respectively.

Syntax

Fortran

CALL SGBTRF | DGBTRF | CGBTRF | ZGBTRF (m, n, kl, ku, a, lda, ipiv, info)

C and C++

sgbtrf | dgbtrf | cgbtrf | zgbtrf (m, n, kl, ku, a, lda, ipiv, info);

LAPACK

info = LAPACK_SGBTRF | LAPACK_DGBTRF | LAPACK_CGBTRF | LAPACK_ZGBTRF
(matrix_layout, m, n, kl, ku, a, lda, ipiv);

On Entry

matrix_layout
indicates whether the input and output matrices are stored in row major order or column major order, where:
- If matrix_layout = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If matrix_layout = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. It must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR

$m$ is the number of rows in matrix $A$.
Specified as: an integer; $m \geq 0$.

$n$ is the number of columns in matrix $A$.
Specified as: an integer; $n \geq 0$.

$kl$ is the lower band width $kl$ of the matrix $A$.
Specified as: an integer; $kl \geq 0$. 

702 ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
**ku** is the upper band width $ku$ of the matrix $A$.
Specified as: an integer; $ku \geq 0$.

**a** is the $m$ by $n$ general band matrix $A$ to be factored.
Specified as: an $lda$ by (at least) $n$ array, containing numbers of the data type indicated in [Table 159 on page 702]

**lda**
is the leading dimension of the array specified for $A$.
Specified as: an integer; $lda > 0$ and $lda \geq 2kl+ku+1$.

**ipiv**
See "On Return".

**info**
See "On Return".

**On Return**

**a** is the transformed matrix $A$ containing the results of the factorization. See "Function."
Returned as: an $lda$ by (at least) $n$ array, containing integers.

**ipiv**
is the integer vector of length $\min(m,n)$, containing the pivot indices.
Returned as: a one-dimensional integer array of (at least) length $\min(m,n)$, containing integers; $1 \leq ipiv_j \leq m$ for all $j$.

**info**
has the following meaning:
If $info = 0$, the subroutine completed successfully.
If $info > 0$, $info$ is set to the first $i$, where $U_{ii}$ is zero. The factorization has been completed.
Returned as:
- For SGBTRF, DGBTRF, CGBTRF and ZGBTRF, returned as: an integer; $info \geq 0$.
- For LAPACKE_sgbtrf, LAPACKE_dgbtrf, LAPACKE_cgbtrf and LAPACKE_zgbtrf, returned as an integer function value; $info \geq 0$.

**Notes**
1. In your C program, argument $info$ must be passed by reference.
2. **a** and **ipiv** must have no common elements; otherwise, results are unpredictable.
3. 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 100.
4. The way these subroutines handle singularity differs from LAPACK. Like LAPACK, these subroutines use the $info$ argument to provide information about the singularity of $A$, but they also provide an error message.

**Function**

These subroutines factor general band matrix $A$, stored in BLAS-general-band storage mode, using Gaussian elimination with partial pivoting to compute the $LU$ factorization of $A$: 
A = PLU

In the formula above:
- P is the permutation matrix
- L is a unit lower triangular band matrix
- U is a upper triangular band matrix

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 SGBTRS, DGBTRS, CGBTRS, and ZGBTRS respectively.

If m or n is 0, no computation is performed and the subroutine returns after doing some parameter checking.

See references [8 on page 1363] and [46 on page 1366].

Error conditions

Resource Errors
None

Computational Errors
Matrix A is singular or nearly singular.
- The first column, i, of L with a corresponding $U_{ii} = 0$ 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. $kl < 0$
4. $ku < 0$
5. $lda \leq 0$
6. $2kl + ku + 1 > lda$

Examples

Example 1

This example shows a factorization of the following real general band matrix of order 9. Matrix A is:

\[
\begin{bmatrix}
1.0 & 2.0 & 3.0 & 4.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
2.0 & 1.0 & 2.0 & 3.0 & 4.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
3.0 & 2.0 & 1.0 & 2.0 & 3.0 & 4.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 3.0 & 2.0 & 1.0 & 2.0 & 3.0 & 4.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 3.0 & 2.0 & 1.0 & 2.0 & 3.0 & 4.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 3.0 & 2.0 & 1.0 & 2.0 & 3.0 & 4.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 3.0 & 2.0 & 1.0 & 2.0 & 3.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 3.0 & 2.0 & 1.0 & 2.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 3.0 & 2.0 & 1.0 \\
\end{bmatrix}
\]

Call Statement and Input:

```
CALL DGBTRF( 9, 9, 2, 3, A, 8, IPIV, INFO)
```

704 ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
Example 2

This example shows a factorization of the following complex general band matrix of order 5. Matrix $A$ is:

$$
A = \begin{bmatrix}
0.100 & 0.100 & (1.000, 2.000) & (1.000, 3.000) & (0.000, 0.000) & (0.000, 0.000) \\
(2.000, 1.000) & (0.200, 0.200) & (2.000, 3.000) & (2.000, 4.000) & (0.000, 0.000) & (0.000, 0.000) \\
(0.000, 0.000) & (3.000, 2.000) & (0.300, 0.300) & (3.000, 4.000) & (3.000, 5.000) & (3.000, 5.000) \\
(0.000, 0.000) & (0.000, 0.000) & (4.000, 3.000) & (4.000, 4.000) & (4.000, 5.000) & (4.000, 5.000) \\
(0.000, 0.000) & (0.000, 0.000) & (0.000, 0.000) & (0.000, 0.000) & (0.000, 0.000) & (0.000, 0.000)
\end{bmatrix}
$$

Call Statement and Input:

```
CALL ZGBTRF(5, 5, 1, 2, A, 5, IPIV, INFO)
```

Output:

```
IPIV = ( 3, 4, 5, 6, 5, 6, 9, 8, 9 )
INFO = 0
```

Chapter 10. Linear Algebraic Equations 705
SGBTRS, DGBTRS, CGBTRS, and ZGBTRS (General Band Matrix Multiple Right-Hand Side Solve)

Purpose

SGBTRS and DGTBRS solve one of the following systems of equations for multiple right-hand sides:

1. \( AX = B \)
2. \( A^T X = B \)

CGBTRS and ZGBTRS solve one of the following systems of equations for multiple right-hand sides:

1. \( AX = B \)
2. \( A^T X = B \)
3. \( A^H X = B \)

In the formulas above:

- \( A \) represents the general band matrix stored in BLAS-general-band storage mode, containing the factorization.
- \( B \) represents the general matrix containing the right-hand sides in its columns.
- \( X \) represents the general matrix containing the solution vectors in its columns.

These subroutines use the results of the factorization of matrix \( A \) and vector \( ipiv \), produced by a preceding call to SGBTRF, DGBTRF, CGBTRF, and ZGBTRF, respectively.

### Table 160. Data Types

<table>
<thead>
<tr>
<th>( A, B )</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGBTRS(^A)</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGBTRS(^A)</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGBTRS(^A)</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGBTRS(^A)</td>
</tr>
</tbody>
</table>

\(^A\)LAPACK

**Note:** The input to these solve subroutines must be the output from the factorization subroutines SGBTRF, DGBTRF, CGBTRF, and ZGBTRF, respectively.

Syntax

**Fortran**

```fortran
CALL SGBTRS | DGBTRS | CGBTRS | ZGBTRS (trans, n, kl, ku, nrhs, a, lda, ipiv, b, ldb, info);
```

**C and C++**

```c
sgbtrs | dgbtrs | cgbtrs | zgbtrs (trans, n, kl, ku, nrhs, a, lda, ipiv, b, ldb, info);
```

**LAPACK**

```c
info = LAPACKE_sgbtrs | LAPACKE_dgbtrs | LAPACKE_cgbtrs | LAPACKE_zgbtrs (matrix_layout, trans, n, kl, ku, nrhs, a, lda, ipiv, b, ldb);
```

**On Entry**

- \( matrix\_layout \) indicates whether the input and output matrices are stored in row major order or column major order, where:
If `matrix_layout = LAPACK_ROW_MAJOR`, the matrices are stored in row major order.

If `matrix_layout = LAPACK_COL_MAJOR`, the matrices are stored in column major order.

Specified as: an integer. It must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR

`trans` indicates the form of matrix $A$ to use in the computation, where:

- If $trans = 'N'$, $A$ is used in the computation, resulting in solution 1.
- If $trans = 'T'$, $A^T$ is used in the computation, resulting in solution 2.
- If $trans = 'C'$, $A^H$ 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: an integer; $n \geq 0$.

`kl` is the lower band width $kl$ of the matrix $A$.

Specified as: an integer; $kl \geq 0$.

`ku` is the upper band width $ku$ of the matrix $A$.

Specified as: an integer; $ku \geq 0$.

`nrhs` is the number of right-hand sides; that is, the number of columns of matrix $B$ used in the computation.

Specified as: an integer; $nrhs \geq 0$.

`a` is the factorization of matrix $A$, produced by a preceding call to SGBTRF, DGBTRF, CGBTRF, or ZGBTRF, respectively.

Specified as: an $lda$ by (at least) $n$ array, containing numbers of the data type indicated in Table 160 on page 706

`lda` is the leading dimension of the array specified for $a$.

Specified as: an integer; $lda > 0$ and $lda \geq 2kl+ku+1$.

`ipiv` is the array containing the pivot indices produced by a preceding call to SGBTRF, DGBTRF, CGBTRF, or ZGBTRF, respectively.

Specified as: a one-dimensional array of (at least) length $n$, containing integers.

`b` is the general 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 160 on page 706

`ldb` is the leading dimension of the array specified for $B$.

Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

`info` See "On Return"
$b$ is the matrix $X$, containing the $nrhs$ solutions to the system. The solutions, each of length $n$, reside in the columns of $X$.

Returned as: an $ldb$ by (at least) $nrhs$ array, containing numbers of the data type indicated in Table 160 on page 706.

$\text{info}$ has the following meaning:

If $info = 0$, the subroutine completed successfully: SGBTRS, DGBTRS, CGBTRS, and ZGBTRS

Returned as:

- For SGBTRS, DGBTRS, CGBTRS, and ZGBTRS, returned as: an integer; info $\geq 0$.
- For LAPACKE_sgbtrs, LAPACKE_dgbtrs, LAPACKE_cgbtrs, and LAPACKE_zgbtrs, returned as an integer function value; info $\geq 0$.

Notes

1. These subroutines accept lowercase letters for the $trans$ arguments.
2. In your C program, argument $info$ must be passed by reference.
3. For SGBTRS and DGBTRS, if you specify 'C' for the $trans$ argument, it is interpreted as though you specified 'T'.
4. The scalar data specified for input argument $n$ must be the same for both _GTTRF and _GTBRS.
5. The array data specified for input arguments $a$ and $ipiv$ for these subroutines must be the same as the corresponding output arguments for SGBTRF, DGBTRF, CGBTRF, or ZGBTRF respectively.
6. $a$, $ipiv$, and $b$ must have no common elements; otherwise, results are unpredictable.
7. 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 100.

Function

One of the following systems of equations is solved for multiple right-hand sides:

1. $AX=B$
2. $A^TX=B$
3. $A^H X = B$ (only for CGBTRS and ZGBTRS)

where $A$ is a general band matrix and $B$ and $X$ are general matrices. These subroutines use the results of the factorization of matrix $A$, produced by a preceding call to SGBTRF, DGBTRF, CGBTRF, or ZGBTRF respectively. For details on the factorization, see “SGBTRF, DGBTRF, CGBTRF and ZGBTRF (General Band Matrix Factorization)” on page 702.

If $n$ or $nrhs$ is 0, no computation is performed.

See reference 46 on page 1366.

Error conditions

Resource Errors
None

Computational Errors
Note: If the factorization performed by SGBTRF, DGBTRF, CGBTRF, or ZGBTRF 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. trans ≠ 'N', 'T', or 'C'
2. \( n < 0 \)
3. \( kl < 0 \)
4. \( ku < 0 \)
5. \( nrhs < 0 \)
6. \( lda \leq 0 \)
7. \( 2kl+ku+1 > lda \)
8. \( ldb \leq 0 \)
9. \( n > ldb \)

Examples

Example 1

This example shows how to solve the system \( AX = B \), where real general band matrix \( A \) is the same matrix factored in Example 1 for DGBTRF.

Call Statement and Input:

\[
\text{TRANS} \quad N \quad KL \quad KU \quad NRHS \quad A \quad LDA \quad IPIV \quad B \quad LDB \quad INFO)
\]

\[
\text{CALL DGBTRS('N', \quad 9 \quad , \quad 2 \quad , \quad 3 \quad , \quad 3 \quad , \quad A \quad , \quad 8 \quad , \quad IPIV \quad , \quad B \quad , \quad 9 \quad , \quad INFO)}
\]

\[
A = \text{(same as output } A \text{ in Example 1)}
\]

\[
IPIV = \text{(same as output } IPIV \text{ in Example 1)}
\]

\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 \\
1.0 & -1.0 & 2.0 \\
1.0 & 1.0 & 3.0 \\
1.0 & -1.0 & 4.0 \\
1.0 & 1.0 & 5.0 \\
1.0 & -1.0 & 6.0 \\
1.0 & 1.0 & 7.0 \\
1.0 & -1.0 & 8.0 \\
1.0 & 1.0 & 9.0 \\
\end{bmatrix}
\]

B =

\[
\begin{bmatrix}
0.629 & 1.149 & 4.713 \\
-0.025 & 1.870 & -0.726 \\
0.523 & 0.615 & 2.946 \\
-0.286 & -1.434 & -2.774 \\
-0.104 & -0.524 & -1.067 \\
-0.118 & -0.591 & -0.970 \\
0.220 & -0.896 & 2.027 \\
-0.079 & 1.604 & -0.340 \\
0.496 & 0.480 & 3.599 \\
\end{bmatrix}
\]

INFO = 0

Example 2

This example shows how to solve the system \( A^TX = B \), where real general band matrix \( A \) is the same matrix factored in Example 1 for DGBTRF.
Call Statement and Input:

\[
\text{TRANS N KL KU NRHS A LDA IPIV B LDB INFO)}
\]

\[
\text{CALL DGBTRS}('T', 9, 2, 3, 3, A, 8, IPIV, B, 9, INFO)
\]

\[
A = \text{(same as output } A \text{ in Example 1)}
\]

\[
IPIV = \text{(same as output } IPIV \text{ in Example 1)}
\]

\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 \\
1.0 & -1.0 & 2.0 \\
1.0 & 1.0 & 3.0 \\
1.0 & -1.0 & 4.0
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
1.0 & 1.0 & 5.0 \\
1.0 & -1.0 & 6.0 \\
1.0 & 1.0 & 7.0 \\
1.0 & -1.0 & 8.0 \\
1.0 & 1.0 & 9.0
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
0.496 & 0.480 & 1.362 \\
-0.079 & 1.604 & -0.450 \\
0.220 & -0.896 & 0.179 \\
-0.118 & -0.591 & -0.211
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
0.104 & 0.524 & 0.018 \\
-0.286 & -1.434 & -0.095 \\
0.523 & 0.615 & 2.285 \\
-0.025 & 1.870 & 0.468 \\
0.629 & 1.149 & 1.586
\end{bmatrix}
\]

\[
INFO = 0
\]

**Example 3**

This example shows how to solve the system \(AX = B\), where complex general band matrix \(A\) is the same matrix factored in Example 2 for ZGBTRF.

Call Statement and Input:

\[
\text{TRANS N KL KU NRHS A LDA IPIV B LDB INFO)}
\]

\[
\text{CALL ZGBTRS}('N', 5, 1, 2, 3, A, 5, IPIV, B, 5, INFO)
\]

\[
A = \text{(same as output } A \text{ in Example 2)}
\]

\[
IPIV = \text{(same as output } IPIV \text{ in Example 2)}
\]

\[
\begin{bmatrix}
(0.100, 1.000)(-1.000, 1.000)(0.200, 0.400) \\
(0.100, 1.000)(-1.000, 1.000)(0.400, 0.800)
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
(0.100, 1.000)(-1.000, 1.000)(0.600, 1.200) \\
(0.100, 1.000)(-1.000, 1.000)(0.800, 1.600) \\
(0.100, 1.000)(-1.000, 1.000)(1.000, 2.000)
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
(-0.039, -0.250)(0.237, -0.265)(-0.247, -0.723) \\
(0.161, -0.014)(0.158, 0.163)(-0.092, -0.192)
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
(0.205, 0.105)(0.069, 0.317)(0.248, 0.130) \\
(0.116, 0.109)(-0.015, 0.224)(0.314, 0.146) \\
(-0.055, 0.038)(-0.090, -0.027)(0.032, -0.001)
\end{bmatrix}
\]

\[
INFO = 0
\]
Example 4

This example shows how to solve the system $A^H X = B$, where complex general
band matrix $A$ is the same matrix factored in Example 2 for ZGBTRF.

Call Statement and Input:

```
TRANS  N  KL  KU  NRHS  A  LDA  IPIV  B  LDB  INFO)
CALL ZGBTRS( 'C', 5, 1, 2, 3, A, 5, IPIV, B, 5, INFO)
```

\[ A = \begin{pmatrix}
(0.100, 1.000)(-1.000, 1.000)(0.200, 0.400) \\
(0.100, 1.000)(-1.000, 1.000)(0.400, 0.800) \\
(0.100, 1.000)(-1.000, 1.000)(0.600, 1.200) \\
(0.100, 1.000)(-1.000, 1.000)(0.800, 1.600) \\
(0.100, 1.000)(-1.000, 1.000)(1.000, 2.000)
\end{pmatrix} \]

\[ B = \begin{pmatrix}
(-0.158, 0.428)(-0.607, 0.209)(-0.002, 0.200) \\
(-0.094, 0.283)(-0.392, 0.149)(-0.054, 0.230) \\
(-0.057, -0.111)( 0.070, -0.161)(-0.114, 0.109) \\
(-0.088, -0.409)( 0.367, -0.460)(-0.111, -0.050)
\end{pmatrix} \]

\[ IPIV = (\text{same as output IPIV in Example 2}) \]

Output:

\[ B = \begin{pmatrix}
(0.015, -0.126)(0.150, -0.096)(0.029, 0.017) \\
(-0.158, 0.428)(-0.607, 0.209)(-0.002, 0.200) \\
(-0.094, 0.283)(-0.392, 0.149)(-0.054, 0.230) \\
(-0.057, -0.111)( 0.070, -0.161)(-0.114, 0.109) \\
(-0.088, -0.409)( 0.367, -0.460)(-0.111, -0.050)
\end{pmatrix} \]

\[ \text{INFO} = 0 \]
SGBS and DGBS (General Band Matrix Solve)

**Purpose**

These subroutines solve the system $Ax = b$ for $x$, where $A$ is a general band matrix, and $x$ and $b$ are vectors. They use the results of the factorization of matrix $A$, produced by a preceding call to SGBF or DGBF, respectively.

**Table 161. Data Types**

<table>
<thead>
<tr>
<th>$A$, $b$, $x$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGBS</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGBS</td>
</tr>
</tbody>
</table>

**Note:** The input to these solve subroutines must be the output from the factorization subroutines SGBF and DGBF, respectively.

**Syntax**

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SGBS</th>
<th>DGBS (agb, lda, n, ml, mu, ipvt, bx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>sgbs</td>
<td>dgbs (agb, lda, n, ml, mu, ipvt, bx);</td>
</tr>
</tbody>
</table>

**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 161](#) where $lda \geq 2ml+mu+16$.

- **lda**
  
is the leading dimension of the array specified for **agb**. Specified as: an integer; $lda > 0$ and $lda \geq 2ml+mu+16$.

- **n**
  
is the order of the matrix $A$. Specified as: an integer; $n > ml$ and $n > mu$.

- **ml**
  
is the lower band width $ml$ of the matrix $A$. Specified as: an integer; $0 \leq ml < n$.

- **mu**
  
is the upper band width $mu$ of the matrix $A$. Specified as: an 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 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 161](#).

**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 161](#).
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 75.

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 100.

Function

The real system \( Ax = b \) is solved for \( x \), where \( A \) is a real general band matrix, stored in general-band storage mode, and \( x \) and \( b \) are vectors. These subroutines use the results of the factorization of matrix \( A \), produced by a preceding call to SGBF or DGBF, respectively. The transformed matrix \( A \), used by this computation, consists of the upper triangular matrix \( U \) and the multipliers necessary to construct \( L \) using \( ipvt \), as defined in “Function” on page 762. See reference 46 on page 1366.

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 ≤ 0
2. ml < 0
3. ml ≥ n
4. mu < 0
5. mu ≥ n
6. lda < 2ml+mu+16

Examples

Example

This example shows how to solve the system \( Ax = b \), where general band matrix \( A \) is the same matrix factored in Example 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 )
```
\[ \text{IPVT} = (2, -65534, -131070, -196606, -262142, -327678, -327680, -327680) \]

\[ \text{BX} = (4.0000, 5.0000, 9.0000, 10.0000, 11.0000, 12.0000, 12.0000, 12.0000, 33.0000) \]

\[ \text{AGB} = \text{(same as output AGB in Example)} \]

Output:

\[ \text{BX} = (1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 0.9999, 1.0001) \]
SPBSV, DPBSV, CPBSV, and ZPBSV (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization and Multiple Right-Hand Side Solve)

Purpose

These subroutines solve the system $AX = B$ for $X$, where $X$ and $B$ are general matrices and:
- For SPBSV and DPBSV, $A$ is a positive definite real symmetric band matrix stored in upper- or lower-band-packed storage mode.
- For CPBSV and ZPBSV, $A$ is a positive definite complex Hermitian band matrix stored in upper- or lower-band-packed storage mode.

Table 162. Data Types

<table>
<thead>
<tr>
<th>$A, B$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SPBSV</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DPBSV</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CPBSV</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZPBSV</td>
</tr>
</tbody>
</table>

LAPACK

Syntax

Fortran

```fortran
CALL SPBSV | DPBSV | CPBSV | ZPBSV (uplo, n, k, nrhs, a, lda, b, ldb, info)
```

C and C++

```c
spbsv | dpbsv | cpbsv | zpbsv (uplo, n, k, nrhs, a, lda, b, ldb, info);
```

LAPACK

```c
info = LAPACK_spbsv | LAPACK_dpbsv | LAPACK_cpbsv | LAPACK_zpbsv
(matrix_layout, uplo, n, k, nrhs, a, lda, b, ldb);
```

On Entry

- **matrix_layout**
  - indicates whether the input and output matrices are stored in row major order or column major order, where:
    - If $matrix_layout =$ LAPACK\_ROW\_MAJOR, the matrices are stored in row major order.
    - If $matrix_layout =$ LAPACK\_COL\_MAJOR, the matrices are stored in column major order.
  - Specified as: an integer. It must be LAPACK\_ROW\_MAJOR or LAPACK\_COL\_MAJOR

- **uplo**
  - indicates whether the matrix $A$ is stored in upper- or lower-band-backed storage mode, 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: a single character. It must be 'U' or 'L'.

- **n**
  - is the order $n$ of matrix $A$ and the number of rows of matrix $B$. Specified as: an integer; $n \geq 0$. 

Chapter 10. Linear Algebraic Equations 715
$k$ is the half band width $k$ of the matrix $A$. Specified as: an integer; $0 \leq k \leq \max(0,n-1)$.

$nrhs$ is the number of right-hand sides; i.e., the number of columns of matrix $B$. Specified as: an integer; $nrhs \geq 0$.

$a$ is the positive definite real symmetric or complex Hermitian band matrix $A$ of order $n$, having a half band width of $k$, where:
- If $uplo = 'U'$, it is stored in upper-band-packed storage mode.
- If $uplo = 'L'$, it 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 162 on page 715.

$lda$ is the leading dimension of the array specified for $A$. Specified as: an integer; $lda > 0$ and $lda > k$.

$b$ is the matrix $B$ of right-hand side vectors. Specified as: the $ldb$ by (at least) $nrhs$ array, containing numbers of the data type indicated in Table 162 on page 715.

$ldb$ is the leading dimension of the array specified for $B$. Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

On Return

$a$ If $info = 0$, $a$ is the updated matrix $A$ containing the results of the Cholesky factorization. See “Function” on page 717.

Returned as: an $lda$ by (at least) $n$ array, containing numbers of the data type indicated in Table 162 on page 715.

$b$ If $info = 0$, $b$ is the general matrix $X$ containing the $nrhs$ solutions to the system. The solutions, each of length $n$, reside in the columns of $X$.

Returned as: an $ldb$ by (at least) $nrhs$ array, containing numbers of the data type indicated in Table 162 on page 715.

$info$ has the following meaning:

If $info = 0$, the subroutine completed successfully.

If $info = i$, the leading minor of order $i$ is not positive definite. The factorization could not be completed and the solution was not computed.

Returned as:

- For SPBSV, DPBSV, CPBSV, and ZPBSV, returned as: an integer; $info \geq 0$.
- For LAPACKE_spbsv, LAPACKE_dpbsv, LAPACKE_cpbsv, and LAPACKE_zpbsv, returned as an integer function value; $info \geq 0$.

Notes

1. In your C program, argument $info$ must be passed by reference.
2. All subroutines accept lowercase letters for the $uplo$ argument.
3. For a description of how real symmetric matrices are stored in upper- or lower-band-packed storage mode, see “Upper-Band-Packed Storage Mode” on page 106 or “Lower-Band-Packed Storage Mode” on page 107, respectively.
For a description of how complex Hermitian matrices are stored in upper- or lower-band-packed storage mode, see “Complex Hermitian Band Matrix Storage Representation” on page 108.

4. 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.

5. The matrices used in this computation must have no common elements; otherwise, results are unpredictable. See “Concepts” on page 75.

6. The way these subroutines handle computational errors differs from LAPACK. Like LAPACK, these subroutines use the info argument to provide information about the computational error, but they also provide an error message.

7. On both input and output, matrix $A$ conforms to LAPACK format.

**Function**

These subroutines solve the system $AX = B$ for $X$, where $X$ and $B$ are general matrices and:

- For SPBSV and DPBSV, $A$ is a positive definite real symmetric band matrix stored in upper- or lower-band-packed storage mode.
- For CPBSV and ZPBSV, $A$ is a positive definite complex Hermitian band matrix stored in upper- or lower-band-packed storage mode.

Matrix $A$ is factored using Cholesky factorization:

- For SPBTRF and DPBTRF:
  - $A = LL^T$ if uplo = 'L'.
  - $A = U^TU$ if uplo = 'U'.

- For CPBTRF and ZPBTRF:
  - $A = LL^H$ if uplo = 'L'.
  - $A = U^HU$ if uplo = 'U'.

Where:

- $L$ is a lower triangular band matrix
- $U$ is an upper triangular band matrix

If $n = 0$, no computation is performed and the subroutine returns after doing some parameter checking. If $n > 0$ and $nrhs = 0$, no solutions are computed and the subroutine returns after factoring the matrix. See references [8 on page 1363], [44 on page 1366], and [73 on page 1367].

**Error conditions**

**Resource Errors**

Unable to allocate internal work area.

**Computational Errors**

Matrix $A$ is not positive definite. For details, see the description of the info argument.

**Input-Argument Errors**

1. uplo ≠ 'U' or 'L'
2. $n < 0$
3. $k < 0$
4. $k > \max(0,n-1)$
5. $nrhs < 0$
6. $lda \leq 0$
7. $k \geq lda$
Examples

Example 1

This example shows how to solve the system $AX = B$, where $A$ is a real positive definite band matrix factored in the form $LL^T$.

Matrix $A$ is the same used as input in Example 1 for DPBTRF.
Matrix $B$ is the same used as input in Example 1 for DPBTRS.

Call Statement and Input:

```
CALL DPBSV('L', 9, 3, 3, A, 4, B, 9, INFO)
```

A = (same as input A in Example 1)
B = (same as input B in Example 1)

Output:

\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & . \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & . & . \\
1.0 & 1.0 & 1.0 & 1.0 & . & . & .
\end{bmatrix}
\]

INFO = 0

Example 2

This example shows how to solve the system $AX = B$, where $A$ is a real positive definite symmetric band matrix factored in the form $U^TU$.

Matrix $A$ is the same used as input in Example 2 for DPBTRF.
Matrix $B$ is the same used as input in Example 2 for DPBTRS.

Call Statement and Input:

```
CALL DPBSV('U', 9, 2, 3, A, 3, B, 9, INFO)
```

A = (same as input A in Example 2)
B = (same as input B in Example 2)

Output:

\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 \\
1.0 & -1.0 & 0.0 \\
1.0 & 1.0 & -1.0 \\
1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 0.0 \\
1.0 & 1.0 & -1.0 \\
1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 0.0 \\
1.0 & 1.0 & -1.0 \\
\end{bmatrix}
\]
$$
A = \begin{bmatrix}
1.0 & 1.0 & 1.0 \\
1.0 & -1.0 & 0.0 \\
1.0 & 1.0 & -1.0 \\
1.0 & -1.0 & 1.0 \\
1.0 & 1.0 & 0.0 \\
1.0 & -1.0 & -1.0 \\
1.0 & 1.0 & 1.0 \\
1.0 & -1.0 & 0.0 \\
1.0 & 1.0 & -1.0 \\
\end{bmatrix}
$$

INFO = 0

**Example 3**

This example shows how to solve the system $AX = B$, where $A$ is a positive definite complex Hermitian band matrix factored in the form $LL^H$.

Matrix $A$ is the same used as input in Example 3 for ZPBTRF.

Matrix $B$ is the same used as input in Example 3 for ZPBTRS.

Call Statement and Input:

```plaintext
CALL ZPBSV ( 'L', 6, 3, 3, A, 4, B, 6, INFO )
```

$A =$ (same as input $A$ in Example 3)

$B =$ (same as input $B$ in Example 3)

Output:

$$
A = \begin{bmatrix}
(1.0, 0.0) & (1.0, 0.0) & (1.0, 0.0) & (1.0, 0.0) & (1.0, 0.0) & (1.0, 0.0) \\
(1.0, 1.0) & (1.0, 1.0) & (1.0, 1.0) & (1.0, 1.0) & (1.0, 1.0) & (1.0, 1.0) \\
(1.0, 1.0) & (1.0, 1.0) & (1.0, 1.0) & (1.0, 1.0) & (1.0, 1.0) & (1.0, 1.0) \\
(1.0, 1.0) & (1.0, 1.0) & (1.0, 1.0) & (1.0, 1.0) & (1.0, 1.0) & (1.0, 1.0) \\
\end{bmatrix}
$$

$B =$ \begin{bmatrix}
(7.0, -33.0) & (1.0, -1.0) & (1.0, 1.0) \\
(19.0, -1.0) & (1.0, -1.0) & (1.0, 1.0) \\
(-5.0, -13.0) & (1.0, -1.0) & (2.0, 1.0) \\
(-11.0, -5.0) & (1.0, -1.0) & (2.0, -1.0) \\
(-3.0, 9.0) & (1.0, -1.0) & (1.0, 2.0) \\
(5.0, -1.0) & (1.0, -1.0) & (1.0, -2.0) \\
\end{bmatrix}

INFO = 0

**Example 4**

This example shows how to solve the system $AX = B$, where $A$ is a complex Hermitian band matrix factored in the form $U^H U$.

Matrix $A$ is the same used as input in Example 4 for ZPBTRF.

Matrix $B$ is the same used as input in Example 4 for ZPBTRS.

Call Statement and Input:

```plaintext
CALL ZPBSV ( 'U', 6, 2, 3, A, 3, B, 6, INFO )
```

$A =$ (same as input $A$ in Example 4)

$B =$ (same as input $B$ in Example 4)

Output:

$$
A =$ \begin{bmatrix}
. & . & (1.0, -1.0) & (1.0, 1.0) & (1.0, -1.0) & (1.0, 1.0) \\
. & (1.0, 1.0) & (1.0, -1.0) & (1.0, 1.0) & (1.0, -1.0) & (1.0, 1.0) \\
(1.0, 0.0) & (1.0, 0.0) & (1.0, 0.0) & (1.0, 0.0) & (1.0, 0.0) & (1.0, 0.0) \\
\end{bmatrix}
$$
\[
\begin{bmatrix}
(5.0, 13.0) & (1.0, -1.0) & (1.0, 1.0) \\
(-3.0, 7.0) & (1.0, -1.0) & (1.0, -1.0) \\
(11.0, -5.0) & (1.0, -1.0) & (2.0, 1.0) \\
(3.0, 7.0) & (1.0, -1.0) & (2.0, -1.0) \\
(1.0, -5.0) & (1.0, -1.0) & (1.0, 2.0) \\
(1.0, 1.0) & (1.0, -1.0) & (1.0, -2.0)
\end{bmatrix}
\]

\[\mathbf{B} = \begin{bmatrix}
(5.0, 13.0) & (1.0, -1.0) & (1.0, 1.0) \\
(-3.0, 7.0) & (1.0, -1.0) & (1.0, -1.0) \\
(11.0, -5.0) & (1.0, -1.0) & (2.0, 1.0) \\
(3.0, 7.0) & (1.0, -1.0) & (2.0, -1.0) \\
(1.0, -5.0) & (1.0, -1.0) & (1.0, 2.0) \\
(1.0, 1.0) & (1.0, -1.0) & (1.0, -2.0)
\end{bmatrix}\]

INFO = 0
SPBTRF, DPBTRF, CPBTRF, and ZPBTRF (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization)

**Purpose**

These subroutines use Cholesky factorization to factor a positive definite real symmetric or complex Hermitian band matrix $A$, stored in upper- or lower-band-packed storage mode.

To solve the system of equations, follow the call to these subroutines with one or more calls to SPBTRS, DPBTRS, CPBTRS, or ZPBTRS, respectively.

**Table 163. Data Types**

<table>
<thead>
<tr>
<th>$A$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SPBTRF&lt;sup&gt;A&lt;/sup&gt;</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DPBTRF&lt;sup&gt;A&lt;/sup&gt;</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CPBTRF&lt;sup&gt;A&lt;/sup&gt;</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZPBTRF&lt;sup&gt;A&lt;/sup&gt;</td>
</tr>
<tr>
<td>&lt;sup&gt;A&lt;/sup&gt;LAPACK</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** The output from these factorization subroutines should be used only as input to the solve subroutines SPBTRS, DPBTRS, CPBTRS, or ZPBTRS, respectively.

**Syntax**

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SPBTRF</th>
<th>DPBTRF</th>
<th>CPBTRF</th>
<th>ZPBTRF (uplo, n, k, a, lda, info)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>spbtrf</td>
<td>dpbtrf</td>
<td>cpbtrf</td>
<td>zpbtrf (uplo, n, k, a, lda, info);</td>
</tr>
<tr>
<td>LAPACK</td>
<td>info = LAPACKE_spbtrf</td>
<td>LAPACKE_dpbtrf</td>
<td>LAPACKE_cpbtrf</td>
<td>LAPACKE_zpbtrf (matrix_layout, uplo, n, k, a, lda);</td>
</tr>
</tbody>
</table>

**On Entry**

**matrix_layout**

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If $matrix\_layout = LAPACK\_ROW\_MAJOR$, the matrices are stored in row major order.
- If $matrix\_layout = LAPACK\_COL\_MAJOR$, the matrices are stored in column major order.

Specified as: an integer. It must be LAPACK\_ROW\_MAJOR or LAPACK\_COL\_MAJOR

**uplo**

indicates whether matrix $A$ is stored in upper- or lower-band-packed storage mode, 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: a single character. It must be 'U' or 'L'.

**n**

is the order $n$ of matrix $A$. Specified as: an integer; $n \geq 0$. 

Chapter 10. Linear Algebraic Equations 721
**k** is the half band width $k$ of the matrix $A$. Specified as: an integer; $0 \leq k \leq \max(0,n-1)$.

**a** is the positive definite real symmetric or complex Hermitian band matrix $A$ of order $n$, having a half band width of $k$, where:
- If $\text{uplo} = 'U'$, it is stored in upper-band-packed storage mode.
- If $\text{uplo} = 'L'$, it 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 163 on page 721.

**lda** is the leading dimension of the array specified for $A$. Specified as: an integer; $lda > 0$ and $lda > k$.

**On Return**

- **a** If $info = 0$, $a$ is the updated matrix $A$ containing the results of the Cholesky factorization. See “Function.”

  Returned as: an $lda$ by (at least) $n$ array, containing numbers of the data type indicated in Table 163 on page 721.

- **info** has the following meaning:
  - If $info = 0$, the subroutine completed successfully.
  - If $info = i$, the leading minor of order $i$ is not positive definite and the factorization could not be completed.

  Returned as:
  - For SPBTRF, DPBTRF, CPBTRF, and ZPBTR, returned as: an integer; $info \geq 0$.
  - For LAPACK_spbtrf, LAPACK_dpbtref, LAPACK_cpbtrf, and LAPACK_zpbtrf, returned as an integer function value; $info \geq 0$.

**Notes**

1. These subroutines accept lower case letters for the $\text{uplo}$ argument.
2. In your C program, argument $info$ must be passed by reference.
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 how real symmetric matrices are stored in upper- or lower-band-packed storage mode, see “Upper-Band-Packed Storage Mode” on page 106 or “Lower-Band-Packed Storage Mode” on page 107, respectively.

  For a description of how complex Hermitian matrices are stored in upper- or lower-band-packed storage mode, see “Complex Hermitian Band Matrix Storage Representation” on page 108.
5. The way these subroutines handle computational errors differs from LAPACK. Like LAPACK, these subroutines use the $info$ argument to provide information about the computational error, but they also provide an error message.
6. On both input and output, matrix $A$ conforms to LAPACK format.

**Function**

These subroutines use Cholesky factorization to factor a positive definite real symmetric or complex Hermitian band matrix $A$, stored in upper- or lower-band-packed storage mode:
• For SPBTRF and DPBTRF:
  – \( A = LL^T \) if uplo = 'L'.
  – \( A = U^T U \) if uplo = 'U'.

• For CPBTRF and ZPBTRF:
  – \( A = LL^H \) if uplo = 'L'.
  – \( A = U^H U \) if uplo = 'U'.

Where:
• \( L \) is a lower triangular band matrix
• \( U \) is a upper triangular band matrix

This factorization can then be used by SPBTRS, DPBTRS, CPBTRS, or ZPBTRS, respectively, to solve the system of equations.

If \( n = 0 \), no computation is performed.

**Error conditions**

**Resource Errors**
Unable to allocate internal work area.

**Computational Errors**
Matrix \( A \) is not positive definite. For details, see the description of the info argument.

**Input-Argument Errors**
1. uplo ≠ 'U' or 'L'
2. \( n < 0 \)
3. \( k < 0 \)
4. \( k > \max(0,n-1) \)
5. \( lda \leq 0 \)
6. \( k \geq lda \)

**Examples**

**Example 1**

This example shows a factorization of the following real positive definite symmetric band matrix \( A \) in the form \( A = LL^T \):

\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 2.0 & 2.0 & 2.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 2.0 & 3.0 & 3.0 & 2.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 3.0 & 2.0 & 1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 3.0 & 2.0 & 1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 3.0 & 2.0 & 1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 3.0 & 2.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 3.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & 3.0 & 4.0 \\
\end{bmatrix}
\]

Call Statement and Input:

```
CALL DPBTRF( 'L' , 9 , 3 , A , 4 , INFO )
```

\( A = \)

\[
\begin{bmatrix}
1.0 & 2.0 & 3.0 & 4.0 & 4.0 & 4.0 & 4.0 & 4.0 & 4.0 & 4.0 \\
1.0 & 2.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 \\
1.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
\end{bmatrix}
\]
### Example 2

This example shows a factorization of the following real positive definite symmetric band matrix \( A \) in the form \( A = U^T U \):

\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 2.0 & -2.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 0.0 & 3.0 & -2.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 3.0 & -2.0 & 1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 3.0 & -2.0 & 1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 3.0 & -2.0 & 1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 3.0 & -2.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 3.0 & -2.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 3.0 \\
\end{bmatrix}
\]

### Call Statement and Input:

```fortran
CALL DPBTRF( 'U' , 9 , 2 , A , 3 , INFO )
```

### Output:

\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 2.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 \\
\end{bmatrix}
\]

### Example 3

This example shows a factorization of the following positive definite complex Hermitian band matrix \( A \) in the form \( A = LL^H \):

\[
\begin{bmatrix}
(1.0 , 0.0) & (1.0 , -1.0) & (1.0 , -1.0) & (1.0 , -1.0) & (0.0 , 0.0) & (0.0 , 0.0) \\
(1.0 , 1.0) & (3.0 , 0.0) & (3.0 , -1.0) & (3.0 , -1.0) & (1.0 , -1.0) & (0.0 , 0.0) \\
(1.0 , 1.0) & (3.0 , 1.0) & (5.0 , 0.0) & (5.0 , -1.0) & (3.0 , -1.0) & (1.0 , -1.0) \\
(1.0 , 1.0) & (3.0 , 1.0) & (5.0 , 1.0) & (7.0 , 0.0) & (5.0 , -1.0) & (3.0 , -1.0) \\
(0.0 , 0.0) & (1.0 , 1.0) & (3.0 , 1.0) & (5.0 , 1.0) & (7.0 , 0.0) & (5.0 , -1.0) \\
(0.0 , 0.0) & (0.0 , 0.0) & (1.0 , 1.0) & (3.0 , 1.0) & (5.0 , 1.0) & (7.0 , 0.0) \\
\end{bmatrix}
\]

### Call Statement and Input:

```fortran
CALL ZPBTRF( 'L' , 6 , 3 , A , 4 , INFO )
```

### Output:

\[
\begin{bmatrix}
(1.0 , ) & (3.0 , ) & (5.0 , ) & (7.0 , ) & (7.0 , ) & (7.0 , ) \\
(1.0 , 1.0) & (3.0 , 1.0) & (5.0 , 1.0) & (5.0 , 1.0) & (5.0 , 1.0) & (5.0 , 1.0) \\
\end{bmatrix}
\]
Example 4

This example shows a factorization of the following positive definite complex Hermitian band matrix $A$ in the form $A = U^H U$:

\[
\begin{bmatrix}
{1.0, 1.0} & {3.0, 1.0} & {3.0, 1.0} & {3.0, 1.0} & . & . \\
{1.0, 1.0} & {1.0, 1.0} & {1.0, 1.0} & {1.0, 1.0} & . & . \\
\end{bmatrix}
\]

Output:

\[
A = \begin{bmatrix}
{1.0, 0.0} & {1.0, 0.0} & {1.0, 0.0} & {1.0, 0.0} & {1.0, 0.0} & {1.0, 0.0} \\
{1.0, 1.0} & {1.0, 1.0} & {1.0, 1.0} & {1.0, 1.0} & {1.0, 1.0} & . \\
{1.0, 1.0} & {1.0, 1.0} & {1.0, 1.0} & {1.0, 1.0} & . & . \\
{1.0, 1.0} & {1.0, 1.0} & {1.0, 1.0} & {1.0, 1.0} & . & . \\
\end{bmatrix}
\]

INFO = 0
SPBTRS, DPBTRS, CPBTRS, and ZPBTRS (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Multiple Right-Hand Side Solve)

Purpose

These subroutines solve the system $AX = B$ for $X$, where $X$ and $B$ are general matrices and:
• For SPBTRS and DPBTRS, $A$ is a positive definite real symmetric band matrix.
• For CPBTRS and ZPBTRS, $A$ is a positive definite complex Hermitian band matrix.

Table 164. Data Types

<table>
<thead>
<tr>
<th>$A, B$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SPBTRS$^A$</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DPBTRS$^A$</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CPBTRS$^A$</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZPBTRS$^A$</td>
</tr>
</tbody>
</table>

$^A$LAPACK

Note: The input to these solve subroutines must be the output from the factorization subroutines SPBTRF, DPBTRF, CPBTRF, and ZPBTRF, respectively.

Syntax

Fortran
CALL SPBTRS | DPBTRS | CPBTRS | ZPBTRS (uplo, n, k, nrhs, a, lda, b, ldb, info)

C and C++
spbtrs | dpbtrs | cpbtrs | zpbtrs (uplo, n, k, nrhs, a, lda, b, ldb, info);

LAPACK
info = LAPACKE_spbtrs | LAPACKE_dpbtrs | LAPACKE_cpbtrs | LAPACKE_zpbtrs
(matrix_layout, uplo, n, k, nrhs, a, lda, b, ldb);

On Entry

$matrix_layout$
indicates whether the input and output matrices are stored in row major order or column major order, where:
• If $matrix_layout = $LAPACK\_ROW\_MAJOR$, the matrices are stored in row major order.
• If $matrix_layout = $LAPACK\_COL\_MAJOR$, the matrices are stored in column major order.

Specified as: an integer. It must be $LAPACK\_ROW\_MAJOR$ or $LAPACK\_COL\_MAJOR$

$uplo$
indicates whether the factored matrix $A$ is stored in upper- or lower-band-backed storage mode, 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: a single character. It must be ‘U’ or ‘L’.

$n$
is the order $n$ of matrix $A$ and the number of rows of matrix $B$. Specified as: an integer; $n \geq 0$. 

726  ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
is the half band width $k$ of the matrix $A$. Specified as: an integer; $0 \leq k \leq \max(0,n-1)$.

$nrhs$

is the number of right-hand sides; i.e., the number of columns of matrix $B$.
Specified as: an integer; $nrhs \geq 0$.

$a$

is the factorization of positive definite matrix $A$, produced by a preceding call to SPBTRF, DPBTRF, CPBTRF, and ZPBTRF, respectively.
Specified as: an $lda$ by (at least) $n$ array, containing numbers of the data type indicated in Table 164 on page 726.

$lda$

is the leading dimension of the array specified for $A$. Specified as: an integer; $lda > 0$ and $lda > k$.

$b$

is the matrix $B$ of right-hand side vectors. Specified as: the $ldb$ by (at least) $nrhs$ array, containing numbers of the data type indicated in Table 164 on page 726.

$ldb$

is the leading dimension of the array specified for $B$. Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

On Return

$b$

is the general matrix $X$, containing the $nrhs$ solutions to the system. The solutions, each of length $n$, reside in the columns of $X$.

Returned as: an $ldb$ by (at least) $nrhs$ array, containing numbers of the data type indicated in Table 164 on page 726.

$info$

$info$ has the following meaning:

If $info = 0$, the solve completed successfully.

Returned as:

- For SPBTRS, DPBTRS, CPBTRS, and ZPBTR, returned as: an integer; $info \geq 0$.
- For LAPACKE_spbtrs, LAPACKE_dpbtrs, LAPACKE_cpbtrs, and LAPACKE_zpbtr, returned as an integer function value; $info \geq 0$.

Notes

1. In your C program, argument $info$ must be passed by reference.

2. All subroutines accept lowercase letters for the $uplo$ argument.

3. For a description of how real symmetric matrices are stored in upper- or lower-band-packed storage mode, see “Upper-Band-Packed Storage Mode” on page 106 or “Lower-Band-Packed Storage Mode” on page 107, respectively.

For a description of how complex Hermitian matrices are stored in upper- or lower-band-packed storage mode, see “Complex Hermitian Band Matrix Storage Representation” on page 108.

4. The scalar data specified for input arguments $uplo$, $n$, $k$, and $lda$ for these subroutines must be the same as the corresponding input arguments specified for SPBTRF, DPBTRF, CPBTRF, and ZPBTRF, respectively.

5. The array data specified for input argument $a$ for these subroutines must be the same as the corresponding output argument for SPBTRF, DPBTRF, CPBTRF, and ZPBTRF, respectively.

6. The matrices used in this computation must have no common elements; otherwise, results are unpredictable. See “Concepts” on page 75.
Function

These subroutines solve the system $AX = B$ for $X$, where $X$ and $B$ are general matrices and:

- For SPBTRS and DPBTRS, $A$ is a positive definite real symmetric band matrix.
- For CPBTRS and ZPBTRS, $A$ is a positive definite complex Hermitian band matrix.

These subroutines use the results of the factorization of matrix $A$, produced by a preceding call to SPBTRF, DPBTRF, CPBTRF, and ZPBTRF, respectively. For a description of how $A$ is factored, see “SPBTRF, DPBTRF, CPBTRF, and ZPBTRF (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization)” on page 721.

If $n$ or $nrhs$ is 0, no computation is performed. See references [8 on page 1363] and [44 on page 1366].

Error conditions

Computational Errors

None

Note: If the factorization performed by SPBTRF, DPBTRF, CPBTRF, and ZPBTRF failed because matrix $A$ 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. $n < 0$
3. $k < 0$
4. $k > \max(0, n-1)$
5. $nrhs < 0$
6. $lda \leq 0$
7. $k \geq lda$
8. $ldb \leq 0$
9. $n > ldb$

Examples

Example 1

This example shows how to solve the system $AX = B$, where matrix $A$ is the same positive definite symmetric band matrix factored in Example 1 for DPBTRF in the form $A = LL^T$.

Call Statement and Input:

```
UPLO N K NRHS A LDA B LDB INFO
| | | | | | | | |
CALL DPBTRS( 'L', 9, 3, 3, A, 4, B, 9, INFO )
```

$A$ = (same output as in Example 1)

\[
\begin{bmatrix}
4.0 & 0.0 & 1.0 \\
8.0 & 0.0 & 1.0 \\
12.0 & 0.0 & 0.0 \\
16.0 & 0.0 & 1.0 \\
\end{bmatrix}
\]

$B$ =

\[
\begin{bmatrix}
16.0 & 0.0 & 0.0 \\
16.0 & 0.0 & -1.0 \\
\end{bmatrix}
\]
$$
\begin{bmatrix}
15.0 & 1.0 & 0.0 \\
13.0 & 1.0 & -2.0 \\
10.0 & 2.0 & -3.0
\end{bmatrix}
$$

Output:

$$
\begin{bmatrix}
1.0 & 1.0 & 1.0 \\
1.0 & -1.0 & 0.0 \\
1.0 & 1.0 & -1.0 \\
1.0 & -1.0 & 1.0 \\
1.0 & 1.0 & 0.0 \\
1.0 & -1.0 & -1.0 \\
1.0 & 1.0 & 1.0 \\
1.0 & -1.0 & 0.0 \\
1.0 & 1.0 & -1.0
\end{bmatrix}
$$

INFO = 0

**Example 2**

This example shows how to solve the system $AX = B$, where matrix $A$ is the same positive definite symmetric band matrix factored in Example 2 for DPBTRF in the form $U^T U$.

Call Statement and Input:

```
CALL DPBTRS('U', 9, 2, 3, A, 3, B, 9, INFO)
```

$A =$ (same as output $A$ in Example 2)

$$
\begin{bmatrix}
1.0 & 3.0 & 0.0 \\
0.0 & -6.0 & 2.0 \\
1.0 & 9.0 & -4.0 \\
1.0 & -9.0 & 4.0 \\
1.0 & 9.0 & 0.0 \\
1.0 & -9.0 & -4.0 \\
1.0 & 9.0 & 4.0 \\
0.0 & -8.0 & -1.0 \\
2.0 & 6.0 & -2.0
\end{bmatrix}
$$

Output:

$$
\begin{bmatrix}
1.0 & 1.0 & 1.0 \\
1.0 & -1.0 & 0.0 \\
1.0 & 1.0 & -1.0 \\
1.0 & -1.0 & 1.0 \\
1.0 & 1.0 & 0.0 \\
1.0 & -1.0 & -1.0 \\
1.0 & 1.0 & 1.0 \\
1.0 & -1.0 & 0.0 \\
1.0 & 1.0 & -1.0
\end{bmatrix}
$$

INFO = 0

**Example 3**

This example shows how to solve the system $AX = B$, where matrix $A$ is the same positive definite complex Hermitian band matrix factored in Example 3 for ZPBTRF in the form $LL^H$.

Call Statement and Input:
Example 4

This example shows how to solve the system $AX = B$, where matrix $A$ is the same positive definite complex Hermitian band matrix factored in Example 4 for ZPBTRF in the form $U^H U$.

Call Statement and Input:

```plaintext
CALL ZPBTRS( 'U', 6, 2, 3, A, 3, B, 6, INFO )
```

Output:

```plaintext
A = (same as output $A$ in Example 4)

\[
\begin{pmatrix}
1.0 & 1.0 & 3.0 & -3.0 & 6.0 & 0.0 \\
1.0 & 1.0 & 3.0 & -9.0 & 13.0 & -7.0 \\
1.0 & 1.0 & 15.0 & -3.0 & 22.0 & 15.0 \\
1.0 & 1.0 & 3.0 & -15.0 & 25.0 & -14.0 \\
1.0 & 1.0 & 15.0 & -1.0 & 18.0 & 19.0 \\
1.0 & 1.0 & 3.0 & -11.0 & 13.0 & -14.0
\end{pmatrix}
\]

$B = \begin{pmatrix}
7.0 & 33.0 & 1.0 & -1.0 & 1.0 & 1.0 \\
19.0 & -1.0 & 1.0 & -1.0 & 1.0 & -1.0 \\
5.0 & -13.0 & 1.0 & -1.0 & 2.0 & 1.0 \\
-11.0 & -5.0 & 1.0 & -1.0 & 2.0 & -1.0 \\
-3.0 & 9.0 & 1.0 & -1.0 & 1.0 & 2.0 \\
5.0 & -10.0 & 1.0 & -1.0 & 1.0 & -2.0
\end{pmatrix}$

INFO = 0
```

Example 4

This example shows how to solve the system $AX = B$, where matrix $A$ is the same positive definite complex Hermitian band matrix factored in Example 4 for ZPBTRF in the form $U^H U$.

Call Statement and Input:

```plaintext
CALL ZPBTRS( 'L', 6, 3, 3, A, 4, B, 6, INFO )
```

Output:

```plaintext
A = (same as output $A$ in Example 3)

\[
\begin{pmatrix}
1.0 & 1.0 & 1.0 & -7.0 & 5.0 & -5.0 \\
1.0 & 1.0 & 9.0 & -13.0 & 18.0 & -4.0 \\
1.0 & 1.0 & 17.0 & -19.0 & 27.0 & 0.0 \\
1.0 & 1.0 & 25.0 & -23.0 & 35.0 & 2.0 \\
1.0 & 1.0 & 23.0 & -19.0 & 28.0 & 5.0 \\
1.0 & 1.0 & 19.0 & -13.0 & 18.0 & -1.0
\end{pmatrix}
\]

$B = \begin{pmatrix}
1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0
\end{pmatrix}$

INFO = 0
SGTSV, DGTSV, CGTSV, and ZGTSV (General Tridiagonal Matrix Factorization and Multiple Right-Hand Side Solve)

**Purpose**

These subroutines solve the general tridiagonal system of linear equations $AX=B$ for $X$, where $A$ is a general tridiagonal matrix and $B$ and $X$ are general matrices.

The matrix $A$ is factored using Gaussian elimination with partial pivoting.

**Table 165. Data Types**

<table>
<thead>
<tr>
<th>$dl$, $d$, $du$, $B$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGTSV$^A$</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGTSV$^A$</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGTSV$^A$</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGTSV$^A$</td>
</tr>
</tbody>
</table>

$^A$LAPACK

**Syntax**

**Fortran**

```fortran
CALL SGTSV | DGTSV | CGTSV | ZGTSV (n, nrhs, dl, d, du, b, ldb, info)
```

**C and C++**

```c
sgtsv | dgtsv | cgtsv | zgtsv (n, nrhs, dl, d, du, b, ldb);
```

**LAPACK**

```c
info = LAPACKE_sgtsv | LAPACKE_dgtsv | LAPACKE_cgtsv | LAPACKE_zgtsv
(matrix_layout, n, nrhs, dl, d, du, b, ldb);
```

**On Entry**

- $matrix_{layout}$ indicates whether the input and output matrices are stored in row major order or column major order, where:
  - If $matrix_{layout} = $LAPACK\_ROW\_MAJOR, the matrices are stored in row major order.
  - If $matrix_{layout} = $LAPACK\_COL\_MAJOR, the matrices are stored in column major order.

Specified as: an integer. It must be $LAPACK\_ROW\_MAJOR$ or $LAPACK\_COL\_MAJOR$.

- $n$ is the order of matrix $A$ and the number of rows in matrix $B$.
  Specified as: an integer; $n \geq 0$.

- $nrhs$ is the number of right-hand sides; that is, the number of columns of matrix $B$ used in the computation.
  Specified as: an integer; $nrhs \geq 0$.

- $dl$ is the array $DL$, containing the $n - 1$ subdiagonal elements of $A$.
  Specified as: a one-dimensional array of (at least) length $n - 1$, containing numbers of the data type indicated in Table 165.

- $d$ is the array $D$, containing the $n$ diagonal elements of $A$.
  Specified as: a one-dimensional array of (at least) length $n$, containing numbers of the data type indicated in Table 165.
**du** is the array **DU**, containing the \( n - 1 \) superdiagonal elements of **A**.

Specified as: a one-dimensional array of (at least) length \( n - 1 \), containing numbers of the data type indicated in Table 165 on page 731.

**b** is the general 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) \( n \) array, containing numbers of the data type indicated in Table 165 on page 731.

**ldb** is the leading dimension of the array specified for **B**.

Specified as: an integer; \( ldb > 0 \) and \( ldb \geq n \).

**info**

See “On Return”.

**On Return**

**dl** The array **DL** is overwritten.

Returned as: a one-dimensional array of (at least) length \( n - 1 \), containing numbers of the data type indicated in Table 165 on page 731.

**d** The array **D** is overwritten.

Returned as: a one-dimensional array of (at least) length \( n \), containing numbers of the data type indicated in Table 165 on page 731.

**du** The array **DU** is overwritten.

Returned as: a one-dimensional array of (at least) length \( n - 1 \), containing numbers of the data type indicated in Table 165 on page 731.

**b** If \( info = 0 \), **b** is the general 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**.

Returned as: an \( ldb \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 165 on page 731.

**info**

has the following meaning:

If \( info = 0 \), the subroutine completed successfully.

If \( info > 0 \), \( info \) is set to the first \( i \), where \( U_{ii} \) is zero. **B** is overwritten; that is, the solution has not been computed.

Returned as:

- For **SGTSV**, **DGTCSV**, **CGTSV**, and **ZGTSV**, returned as: an integer; \( info \geq 0 \).
- For **LAPACKE_sgtsv**, **LAPACKE_dgtsv**, **LAPACKE_cgtsv**, and **LAPACKE_zgtsv**, returned as an integer function value; \( info \geq 0 \).

**Notes**

1. In your C program, argument \( info \) must be passed by reference.
2. **dl**, **d**, **du**, and **B** must have no common elements; otherwise, results are unpredictable.
3. For a description of how general tridiagonal matrices are stored, see “General Tridiagonal Matrix” on page 112.
The way these subroutines handle singularity differs from LAPACK. Like LAPACK, these subroutines use the info argument to provide information about the singularity of A, but they also provide an error message.

**Function**

These subroutines solve the general tridiagonal system of linear equations $AX = B$, where $A$ is a general tridiagonal matrix and $B$ and $X$ are general matrices.

If $n$ is 0 or $nrhs$ is 0, no computation is performed and the subroutine returns after doing some parameter checking.

See reference [8 on page 1363].

**Error conditions**

**Resource Errors**
None

**Computational Errors**
- Matrix $A$ is singular or nearly singular.
- The first column, $i_1$ of $L$ with a corresponding zero 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 2168 is set to be unlimited in the ESSL error option table.

**Input-Argument Errors**
1. $n < 0$
2. $nrhs < 0$
3. $ldb \leq 0$
4. $n > ldb$

**Examples**

**Example 1**

This example shows how to solve the real general tridiagonal system $AX = B$, where:

Matrix $A$ is the same used as input in Example 1 for DGTRF.
Matrix $B$ is the same used as input in Example 1 for DGTRS.

**Note:** On output, arrays DL, D, and DU are overwritten.

Call Statement and Input:

```
N  NRHS DL  D  DU  B  LDB  INFO
|   |   |   |   |   |
CALL DGTSV( 9, 3, DL, D, DU, B, 9, INFO)
```

```
DL = (same as input DL in [Example 1])
D  = (same as input D  in [Example 1])
DU = (same as input DU in [Example 1])
B  = (same as input B  in [Example 1])
```

Output:
Example 2

This example shows how to solve the complex general tridiagonal system $AX = B$, where:

Matrix $A$ is the same used as input in Example 2 for ZGTTRF.
Matrix $B$ is the same used as input in Example 3 for ZGTTRS.

Note: On output, arrays DL, D, and DU are overwritten.

Call Statement and Input:

```
CALL ZGTSV( 4, 3, DL, D, DU, B, 4, INFO)
```

DL = (same as input DL in Example 2)
D = (same as input D in Example 2)
DU = (same as input DU in Example 2)
B = (same as input B in Example 3)

Output:

$$
B = \begin{bmatrix}
-0.247 & 0.119 & 0.0 \\
0.311 & 0.220 & 0.0 \\
0.357 & -0.394 & 0.0 \\
-0.073 & 0.183 & 0.0 \\
\end{bmatrix}
$$

INFO = 0
SGTTRF, DGTTRF, CGTTRF, and ZGTTRF (General Tridiagonal Matrix Factorization)

Purpose

These subroutines factor general tridiagonal 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 SGTTRS, DGTTRS, CGTTRS, or ZGTTRS, respectively.

Table 166. Data Types

<table>
<thead>
<tr>
<th>$dl$, $d$, $du$, $du2$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGTTRF$^A$</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGTTRF$^A$</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGTTRF$^A$</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGTTRF$^A$</td>
</tr>
</tbody>
</table>

$^A$LAPACK

Note: The output from these factorization subroutines should be used only as input to the solve subroutines SGTTRS, DGTTRS, CGTTRS, or ZGTTRS, respectively.

Syntax

Fortran

CALL SGTTRF | DGTTRF | CGTTRF | ZGTTRF ($n$, $dl$, $d$, $du$, $du2$, $ipiv$, $info$)

C and C++

sgttrf | dgttrf | cgttrf | zgttrf ($n$, $dl$, $d$, $du$, $du2$, $ipiv$, $info$);

LAPACK

info = LAPACKE_sgttrf | LAPACKE_dgttrf | cgttrf | LAPACKE_zgttrf ($matrix\_layout$, $n$, $dl$, $d$, $du$, $du2$, $ipiv$);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If $matrix\_layout = LAPACK\_ROW\_MAJOR$, the matrices are stored in row major order.
- If $matrix\_layout = LAPACK\_COL\_MAJOR$, the matrices are stored in column major order.

Specified as: an integer. It must be LAPACK\_ROW\_MAJOR or LAPACK\_COL\_MAJOR

$n$  the order of general tridiagonal matrix $A$ used in the computation.

Specified as: an integer; $n \geq 0$.

$dl$  is the array $DL$, containing the $n - 1$ subdiagonal elements of $A$.

Specified as: a one-dimensional array of (at least) length $n - 1$, containing numbers of the data type indicated in Table 166.

$d$  is the array $D$, containing the $n$ diagonal elements of $A$. 
Specified as: a one-dimensional array of (at least) length \( n \), containing numbers of the data type indicated in Table 166 on page 735.

\( du \) is the array \( DU \), containing the \( n - 1 \) superdiagonal elements of \( A \).

Specified as: a one-dimensional array of (at least) length \( n - 1 \), containing numbers of the data type indicated in Table 166 on page 735.

\( du2 \)
See "On Return".

\( ipiv \)
See "On Return".

\( info \)
See "On Return".

**On Return**

\( dl \) is the array \( DL \), containing the \( n - 1 \) multipliers that define matrix \( L \) from the factorization of \( A \).

Returned as: a one-dimensional array of (at least) length \( n - 1 \), containing numbers of the data type indicated in Table 166 on page 735.

\( d \) is the array \( D \), containing the \( n \) diagonal elements of matrix \( U \) from the factorization of \( A \).

Returned as: a one-dimensional array of (at least) length \( n \), containing numbers of the data type indicated in Table 166 on page 735.

\( du \) is the array \( DU \), containing the \( n - 1 \) elements of the first superdiagonal of matrix \( U \) from the factorization of \( A \).

Returned as: a one-dimensional array of (at least) length \( n - 1 \), containing numbers of the data type indicated in Table 166 on page 735.

\( du2 \)
is the array \( DU2 \), containing the \( n - 2 \) elements of the second superdiagonal of matrix \( U \) from the factorization of \( A \).

Returned as: a one-dimensional array of (at least) length \( n - 2 \), containing numbers of the data type indicated in Table 166 on page 735.

\( ipiv \)
Contains the pivot indices.

For \( 1 \leq i \leq n \), row \( i \) of the matrix was interchanged with row \( ipiv_i \). \( ipiv_i \) will always be either \( i \) or \( i + 1 \).

If \( ipiv_i = i \), no row interchange was required.

Returned as: a one-dimensional integer array of (at least) length \( n \), containing integers; \( 1 \leq ipiv_i \leq n \).

\( info \)
has the following meaning:

If \( info = 0 \), the subroutine completed successfully.

If \( info > 0 \), \( info \) is set to the first \( i \), where \( U_i \) is zero. The factorization has been completed.

Returned as:

- For SGTTRF, DGTTRF, CGTTRF, and ZGTTRF, returned as: an integer; \( info \geq 0 \).
• For LAPACKE_sgttrf, LAPACKE_dgttrf, LAPACKE_cgttrf, and
  LAPACKE_zgttrf, returned as an integer function value; info ≥ 0.

Notes
1. In your C program, argument info must be passed by reference.
2. $dl$, $d$, $du$, $du2$, and $ipiv$ must have no common elements; otherwise results are
   unpredictable.
3. For a description of how general tridiagonal matrices are stored, see “General
   Tridiagonal Matrix” on page 112.
4. The way these subroutines handle singularity differs from LAPACK. Like
   LAPACK, these subroutines use the info argument to provide information about
   the singularity of $A$, but they also provide an error message.

Function

These subroutines factor general tridiagonal matrix $A$ using Gaussian elimination
with partial pivoting where:

$$A = LU$$

In the formula above:

• $L$ is a product of permutation and unit lower bidiagonal matrices.
• $U$ is upper triangular with non-zeros in only the main diagonal and first two
  superdiagonals.

If $n$ is 0, no computation is performed and the subroutine returns after doing some
parameter checking.

See reference [8 on page 1363].

Error conditions

Resource Errors
None

Computational Errors
Matrix $A$ is singular or nearly singular.
• The first column, $i$, of $L$ with a corresponding zero 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 2168 is set to be unlimited in the ESSL error option
table.

Input-Argument Errors

$n < 0$

Examples

Example 1

This example shows a factorization of a real general tridiagonal matrix of order
9.

Matrix $A$ is:
Example 2

This example shows a factorization of a complex general tridiagonal matrix of order 4.

Matrix A is:

\[
\begin{bmatrix}
1.0 & 4.0 & 0.0 & 0.0 \\
3.0 & 1.0 & 4.0 & 0.0 \\
0.0 & 0.0 & 3.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 3.0
\end{bmatrix}
\]

Call Statement and Input:

\[
\begin{align*}
\text{CALL DGTRF( } & 9 \text{, DL, D, DU, DU2, IPIV, INFO )} \\
\text{DL} & = (3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0) \\
\text{D} & = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0) \\
\text{DU} & = (4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0)
\end{align*}
\]

Output:

\[
\begin{align*}
\text{DL} & = (0.333, 0.818, 0.696, 0.908, 0.849, -0.799, 0.625) \\
\text{D} & = (3.000, 3.666, 3.000, 3.303, 3.532, 3.000, 4.799) \\
\text{DU} & = (1.000, -1.333, 1.000, -2.787, 4.000, 1.000, 3.196) \\
\text{DU2} & = (4.000, 0.000, 4.000, 0.000, 4.000, 0.000)
\end{align*}
\]

\[
\begin{align*}
\text{IPIV} & = (2, 2, 4, 4, 5, 7, 7, 9) \\
\text{INFO} & = 0
\end{align*}
\]
IPIV = (2, 2, 4, 4)
INFO = 0
SGTTRS, DGTTRS, CGTTRS, and ZGTTRS (General Tridiagonal Matrix Multiple Right-Hand Side Solve)

Purpose

SGTTRS and DGTTRS solve one of the following systems of equations for multiple right-hand sides:

1. \( AX = B \)
2. \( A^TX = B \)

CGTTRS and ZGTTRS solve one of the following systems of equations for multiple right-hand sides:

1. \( AX = B \)
2. \( A^TX = B \)
3. \( A^HX = B \)

In the formulas above:
- \( A \) represents the general tridiagonal matrix \( A \) containing the 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 vectors \( dl, d, du, du2, \) and \( ipiv \), produced by a preceding call to SGTTRF, DGTTRF, CGTTRF, and ZGTTRF, respectively.

<table>
<thead>
<tr>
<th>Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>( dl, d, du, du2, B )</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
<tr>
<td>Short-precision complex</td>
</tr>
<tr>
<td>Long-precision complex</td>
</tr>
<tr>
<td>(^a)LAPACK</td>
</tr>
</tbody>
</table>

Note: The input to these solve subroutines must be the output from the factorization subroutines SGTTRF, DGTTRF, CGTTRF, and ZGTTRF, respectively.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL SGTTRS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C and C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>sgttrs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LAPACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>info = LAPACK_sgttrs</td>
</tr>
</tbody>
</table>

On Entry

- \( matrix\_layout \)
  - indicates whether the input and output matrices are stored in row major order or column major order, where:
- If \( \text{matrix\_layout} = \text{LAPACK\_ROW\_MAJOR} \), the matrices are stored in row major order.
- If \( \text{matrix\_layout} = \text{LAPACK\_COL\_MAJOR} \), the matrices are stored in column major order.

Specified as: an integer. It must be LAPACK\_ROW\_MAJOR or LAPACK\_COL\_MAJOR

\( \text{trans} \)

indicates the form of matrix \( A \) to use in the computation, where:

- If \( \text{trans} = 'N' \), \( A \) is used in the computation, resulting in solution 1.
- If \( \text{trans} = 'T' \), \( A^T \) is used in the computation, resulting in solution 2.
- If \( \text{trans} = 'C' \), \( A^H \) 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: an integer; \( n \geq 0 \).

\( \text{nrhs} \)

is the number of right-hand sides; that is, the number of columns of matrix \( B \) used in the computation.

Specified as: an integer; \( nrhs \geq 0 \).

\( dl \)

is the array \( DL \), containing the \( n - 1 \) multipliers that define matrix \( L \) from the factorization of \( A \), produced by a preceding call to SGTTRF, DGTTRE, CGTTRF, or ZGTTRF, respectively.

Specified as: a one-dimensional array of (at least) length \( n - 1 \), containing numbers of the data type indicated in Table 167 on page 740.

\( d \)

is the array \( D \), containing the \( n \) diagonal elements of matrix \( U \) from the factorization of \( A \), produced by a preceding call to SGTTRF, DGTTRE, CGTTRF, or ZGTTRF, respectively.

Specified as: a one-dimensional array of (at least) length \( n \), containing numbers of the data type indicated in Table 167 on page 740.

\( du \)

is the array \( DU \), containing the \( n - 1 \) elements of the first superdiagonal of matrix \( U \) from the factorization of \( A \), produced by a preceding call to SGTTRF, DGTTRE, CGTTRF, or ZGTTRF, respectively.

Specified as: a one-dimensional array of (at least) length \( n - 1 \), containing numbers of the data type indicated in Table 167 on page 740.

\( du2 \)

is the array \( DU2 \), containing the \( n - 2 \) elements of the second superdiagonal of matrix \( U \) from the factorization of \( A \), produced by a preceding call to SGTTRF, DGTTRE, CGTTRF, or ZGTTRF, respectively.

Specified as: a one-dimensional array of (at least) length \( n - 2 \), containing numbers of the data type indicated in Table 167 on page 740.

\( ipiv \)

is the array containing the pivot indices produced by a preceding call to SGTTRF, DGTTRE, CGTTRF, and ZGTTRF, respectively.

Specified as: a one-dimensional array of (at least) length \( n \), containing integers; \( 1 \leq ipiv_i \leq n \).
is the general 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) $n$ array, containing numbers of the data type indicated in Table 167.

$ldb$

is the leading dimension of the array specified for $B$.

Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

$info$

See "On Return".

On Return

$b$ is the matrix $X$, containing the $nrhs$ solutions to the system. The solutions, each of length $n$, reside in the columns of $X$.

Returned as: an $ldb$ by (at least) $nrhs$ array, containing numbers of the data type indicated in Table 167.

$info$

has the following meaning:

If $info = 0$, the subroutine completed successfully.

Returned as:

- For >SGTTRS, DGTTRS, CGTTRS, and ZGTTRS, returned as: an integer; info $\geq 0$.
- For LAPACKE_sgttrs, LAPACKE_dgttrs, LAPACKE_cgttrs, and LAPACKE_zgttrs, returned as an integer function value; info $\geq 0$.

Notes

1. These subroutines accept lowercase letters for the $trans$ arguments.
2. In your C program, argument $info$ must be passed by reference.
3. $dl$, $d$, $du$, $du2$, $ipiv$, and $B$ must have no common elements; otherwise results are unpredictable.
4. For SGTTRS and DGTTRS, if you specify 'C' for the $trans$ argument, it is interpreted as though you specified 'T'.
5. The scalar data specified for input argument $n$ must be the same for both _GTTRF and _GTTRS.
6. The array data specified for input arguments $d$, $dl$, $du$, $du2$, and $ipiv$ for these subroutines must be the same as the corresponding output arguments for _GTTRF, _GTTF, _GTTRF, and _GTTRF, respectively.
7. For a description of how general tridiagonal matrices are stored, see "General Tridiagonal Matrix" on page 112.

Function

One of the following systems of equations is solved for multiple right-hand sides:

1. $AX=B$
2. $A^TX=B$
3. $A^HX=B$ (only for CGTTRS and ZGTTRS)

where $A$ is a general tridiagonal matrix and $B$ and $X$ are general matrices. These subroutines use the results of the factorization of matrix $A$, produced by a preceding call to _GTTRF, _GTTRF, _GTTRF, or _GTTRF, respectively. For details
on the factorization, see "SGTTF, DGTTTF, CGTTRF, and ZGTTRF (General Tridiagonal Matrix Factorization)" on page 735.

If \( n = 0 \) or \( nrhs \) is 0, no computation is performed and the subroutine returns after doing some parameter checking.

See reference [8 on page 1363].

**Error conditions**

**Resource Errors**

None

**Computational Errors**

None

**Note:** If the factorization performed by SGTTF, DGTTTF, CGTTRF or ZGTTRF 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. \( \text{trans} \neq 'N', 'T', \text{or 'C'} \)
2. \( n < 0 \)
3. \( nrhs < 0 \)
4. \( ldb \leq 0 \)
5. \( n > ldb \)

**Examples**

**Example 1**

This example shows how to solve the real general tridiagonal system \( AX = B \), where matrix \( A \) is the same matrix factored in [Example 1] for DGTTTF.

Call Statement and Input:

```
TRANS N NRHS DL D DU DU2 IPIV B LDB INFO
<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
CALL DGTTTRS( 'N', 9, 3, DL, D, DU, DU2, IPIV, B, 9, INFO)
```

\[
\begin{align*}
DL &= \text{(same as output DL in [Example 1])} \\
D &= \text{(same as output D in [Example 1])} \\
DU &= \text{(same as output DU in [Example 1])} \\
DU2 &= \text{(same as output DU2 in [Example 1])} \\
IPIV &= \text{(same as output IPIV in [Example 1])} \\
\end{align*}
\]

\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 \\
1.0 & -1.0 & 2.0 \\
1.0 & 1.0 & 3.0 \\
1.0 & -1.0 & 4.0 \\
1.0 & 1.0 & 5.0 \\
1.0 & -1.0 & 6.0 \\
1.0 & 1.0 & 7.0 \\
1.0 & -1.0 & 8.0 \\
1.0 & 1.0 & 9.0
\end{bmatrix}
\]

Output:
Example 2

This example shows how to solve the real general tridiagonal system \( A^T X = B \), where matrix \( A \) is the same matrix factored in Example 1 for DGTTRF.

Call Statement and Input:

```plaintext
CALL DGTTRS( 'T', 9, 3, DL, D, DU, DU2, IPIV, B, 9, INFO)
```

\[
\begin{bmatrix}
0.609 & 0.478 & 4.597 \\
0.098 & 0.130 & -0.899 \\
n-0.231 & -0.641 & -2.723 \\
0.234 & 0.312 & 2.105 \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
0.364 & 0.153 & 2.516 \\
-0.017 & -0.023 & -0.958 \\
n-0.019 & -0.359 & -0.147 \\
0.267 & 0.357 & 2.505 \\
0.198 & -0.070 & 1.484 \\
\end{bmatrix}
\]

INFO = 0

Example 3

This example shows how to solve the complex general tridiagonal system \( AX = B \), where matrix \( A \) is the same matrix factored in Example 2 for ZGTTRF.

Call Statement and Input:

```
744 ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
```
This example shows how to solve the complex general tridiagonal system \( A^T X = B \), where matrix \( A \) is the same matrix factored in Example 2 for ZGTTRF.

Call Statement and Input:

\[
\begin{align*}
\text{CALL ZGTTRS}(&'T', 4, 3, \ DL, \ D, \ DU, \ DU2, \ IPIV, \ B, 4, \ INFO) \\
\end{align*}
\]

Output:

\[
\begin{align*}
\begin{pmatrix}
(1.0, 1.0) & (1.0, 1.0) & (1.0, -1.0) \\
(1.0, 1.0) & (-1.0, -1.0) & (1.0, -1.0) \\
(1.0, 1.0) & (1.0, 1.0) & (1.0, -1.0) \\
\end{pmatrix}
\end{align*}
\]

\[
\begin{align*}
\begin{pmatrix}
(-0.247, 0.000) & (0.119, 0.000) & (0.000, 0.247) \\
(0.311, 0.000) & (0.220, 0.000) & (0.000, -0.311) \\
(0.357, 0.000) & (-0.394, 0.000) & (0.000, -0.357) \\
(-0.073, 0.000) & (0.183, 0.000) & (0.000, 0.073) \\
\end{pmatrix}
\end{align*}
\]

INFO = 0

Example 4
SPTSV, DPTSV, CPTSV, and ZPTSV (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization and Multiple Right-Hand Side Solve)

Purpose

SPTSV and DPTSV solve the tridiagonal system $AX = B$ for $X$, where $X$ and $B$ are general matrices and $A$ is a positive definite real symmetric matrix stored in LAPACK-symmetric-tridiagonal storage mode.

CPTSV and ZPTSV solve one of the following tridiagonal systems for $X$, where $X$ and $B$ are general matrices and $A$ is a positive definite complex Hermitian matrix stored in LAPACK-complex Hermitian-tridiagonal storage mode:

- If you specify the subdiagonal of $A$ in $e$, then this subroutine solves $AX = B$.
- If you specify the superdiagonal of $A$ in $e$, then this subroutines solves $A^TX = B$.

Table 168. Data Types

<table>
<thead>
<tr>
<th>d</th>
<th>e, B</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SPTSV(^3)</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DPTSV(^3)</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>Short-precision complex</td>
<td>CPTSV(^3)</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision complex</td>
<td>ZPTSV(^3)</td>
</tr>
</tbody>
</table>

\(^3\)LAPACK

Syntax

Fortran

```
CALL SPTSV | DPTSV | CPTSV | ZPTSV (n, nrhs, d, e, b, ldb, info)
```

C and C++

```
sptsv | dptsv | cptsv | zptsv (n, nrhs, d, e, b, ldb, info);
```

LAPACK

```
info = LAPACKE_sptsv | LAPACKE_dptsv | LAPACKE_cptsv | LAPACKE_zptsv(matrix_layout, n, nrhs, d, e, b, ldb);
```

On Entry

- **matrix_layout** indicates whether the input and output matrices are stored in row major order or column major order, where:
  - If $matrix\_layout = \text{LAPACK\_ROW\_MAJOR}$, the matrices are stored in row major order.
  - If $matrix\_layout = \text{LAPACK\_COL\_MAJOR}$, the matrices are stored in column major order.

Specified as: an integer. It must be LAPACK\_ROW\_MAJOR or LAPACK\_COL\_MAJOR

- **n** is the order $n$ of tridiagonal matrix $A$. Specified as: an integer; $n \geq 0$.

- **nrhs** is the number of right-hand sides; i.e., the number of columns of matrix $B$. Specified as: an integer; $nrhs \geq 0$.

- **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 168.
is the vector $e$ containing the subdiagonal or superdiagonal 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 $n-1$, containing numbers of the data type indicated in Table 168 on page 746.

$b$ is the matrix $B$ of right-hand side vectors. Specified as the $ldb$ by (at least) $nrhs$ array, containing numbers of the data type indicated in Table 168 on page 746.

$ldb$ is the leading dimension of the array specified for $B$. Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

On Return

$d$ if $info=0$, is the vector $d$, containing the diagonal $D$ 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 168 on page 746.

$e$ if $info=0$, is the vector $e$, as follows:

For SPTSV and DPTSV

$e$ contains the subdiagonal or superdiagonal elements of the unit lower bidiagonal factor $L$ in positions $1$ through $n-1$ in an array, referred to as $E$.

For CPTSV and ZPTSV

$e$ contains the following:

- If, on entry, you specified the subdiagonal of matrix $A$ in $e$, $e$ contains the subdiagonal elements of the unit lower bidiagonal factor $L$ in positions $1$ through $n-1$ in an array, referred to as $E$.
- If, on entry, you specified the superdiagonal of matrix $A$ in $e$, $e$ contains the superdiagonal elements of the unit upper bidiagonal factor $U$ in positions $1$ through $n-1$ in an array, referred to as $E$.

Returned as: a one-dimensional array of (at least) length $n-1$, containing numbers of the data type indicated in Table 168 on page 746. It has the same length as $E$ on entry.

$b$ if $info = 0$, $b$ is the general matrix $X$, containing the solutions to the system.

Returned as: an $ldb$ by (at least) $nrhs$ array, containing numbers of the data type indicated in Table 168 on page 746.

$info$ has the following meaning:

If $info = 0$, the subroutine completed successfully.

If $info = i$, the leading minor of order $i$ is not positive definite. The factorization could not be completed and the solution was not computed.

Returned as:

- For SPTSV, DPTSV, CPTSV, and ZPTSV, returned as: an integer; $info \geq 0$.
- For LAPACK_sptsv, LAPACK_dptsv, LAPACK_cptsv, and LAPACK_zptsv, returned as an integer function value; $info \geq 0$.

Notes

1. In your C program, argument $info$ must be passed by reference.

2. For a description of how real symmetric tridiagonal matrices are stored in LAPACK-symmetric-tridiagonal storage mode, see “LAPACK-Symmetric-Tridiagonal Storage Mode” on page 114. For a description of how complex
Hermitian tridiagonal matrices are stored in LAPACK-complex Hermitian-tridiagonal storage mode. “Complex Hermitian Tridiagonal Storage Representation” on page 116.

3. The way these subroutines handle computational errors differs from LAPACK. Like LAPACK, these subroutines use the info argument to provide information about the computational error, but they also provide an error message.

4. On both input and output, matrix A conforms to LAPACK format.

**Function**

SPTSV and DPTSV solve the tridiagonal system $AX = B$ for $X$, where $X$ and $B$ are general matrices and $A$ is a positive definite real symmetric matrix stored in LAPACK-symmetric-tridiagonal storage mode.

The matrix $A$ is factored using $A = LDL^T$.

**Note:** Because $A$ is symmetric, this may be considered to be a $U^TDU$ factorization as well.

CPTSV and ZPTSV solve one of the following tridiagonal systems for $X$, where $X$ and $B$ are general matrices and $A$ is a positive definite complex Hermitian matrix stored in LAPACK-complex Hermitian-tridiagonal storage mode:

- If you specify the subdiagonal of $A$ in $e$, then this subroutine solves $AX = B$ and $A = LDL^H$.
- If you specify the superdiagonal of $A$ in $e$, then this subroutine solves $A^TX = B$ and $A = U^TDU$.

If $n = 0$, no computation is performed and the subroutine returns after doing some parameter checking. If $n > 0$ and $nrhs = 0$, no solutions are computed and the subroutine returns after factoring the matrix. See references [8 on page 1363], [44 on page 1366], and [73 on page 1367].

**Error conditions**

**Computational Errors**

Matrix $A$ is not positive definite. For details, see the description of the info argument.

**Input-Argument Errors**

1. $n < 0$
2. $nrhs < 0$
3. $ldb <= 0$
4. $n > ldb$

**Examples**

**Example 1**

This example shows how to solve the positive definite real symmetric tridiagonal system of linear equations $AX = B$, where:

Matrix $A$ is the same used as input in **Example 1** for DPTTRF.

Matrix $B$ is the same used as input in **Example 1** for DPTTRS.

Call Statement and Input:

```
CALL DPTSV( 4 , 2 , D , E , B , 4 , INFO )
```

748 ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
Example 2
This example shows how to solve the positive definite complex Hermitian tridiagonal system of linear equations $AX = B$, where:
- Matrix $A$ is the same used as input in Example 2 for ZPTTRF.
- Matrix $B$ is the same used as input in Example 2 for ZPTTRS.

Call Statement and Input:
```
CALL ZPTSV( 4 , 3 , D , E , B , 4 , INFO )
```

Output:
```
D = ( 1.0, 2.0, 3.0, 4.0 )
E = ( ( 1.0, 1.0 ) ( 1.0, 1.0 ) ( 1.0, 1.0 ) )
B =
    [ ( 1.0, 0.0 ) ( 1.0, 0.0 ) ( 1.0, 0.0 ) ]
    [ ( 1.0, 1.0 ) ( 1.0, 1.0 ) ( 1.0, 1.0 ) ]
```

INFO = 0

Example 3
This example shows how to solve the positive definite complex Hermitian tridiagonal system of linear equations $A^TX = B$, where:
- Matrix $A$ is the same used as input in Example 3 for ZPTTRF.
- Matrix $B$ is the same used as input in Example 3 for ZPTTRS.

Call Statement and Input:
```
CALL ZPTSV( 4 , 3 , D , E , B , 4 , INFO )
```
\[ D = \text{(same as output } D \text{ in Example 3)} \]
\[ E = \text{(same as output } E \text{ in Example 3)} \]
\[ B = \text{(same input } B \text{ in Example 3)} \]

Output:
\[ D = (1.0 \ 2.0 \ 3.0 \ 4.0) \]
\[ E = (1.0, -1.0) (1.0, -1.0) (1.0, -1.0) \]
\[ B = \begin{bmatrix} 3.00, -3.33 & 6.33, -0.33 & -0.33, -6.33 \\ 0.66, 1.66 & -1.00, 2.33 & 2.33, 1.00 \\ 0.83, -1.50 & 2.33, -0.66 & -0.66, -2.33 \\ 1.50, 1.00 & 0.50, 2.50 & 2.50, -0.50 \end{bmatrix} \]

\[ \text{INFO} = 0 \]
SPTTRF, DPTTRF, CPTTRF, and ZPTTRF (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization)

Purpose

SPTTRF and DPTTRF factor a positive definite real symmetric tridiagonal matrix stored in LAPACK-symmetric-tridiagonal storage mode:

\[ A = LDL^T \]

CPTTRF and ZPTTRF factor a positive definite complex Hermitian tridiagonal matrix stored in LAPACK-complex Hermitian-tridiagonal storage mode:

- If you specify the subdiagonal of \( A \) in vector \( e \), then \( A = LDL^H \)
- If you specify the superdiagonal of \( A \) in vector \( e \), then \( A = U^H DU \)

To solve the system of equations with one or more right-hand sides, follow the call to SPTTRF, DPTTRF, CPTTRF, or ZPTTRF with a call to SPTTRS, DPTTRS, CPTTRS, or ZPTTRS, respectively.

Table 169. Data Types

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>e</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>Short-precision complex</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision complex</td>
</tr>
</tbody>
</table>

\(^A\)LAPACK

Note: The output from these factorization subroutines should be used only as input to the solve subroutines SPTTRS, DPTTRS, CPTTRS, or ZPTTRS, respectively.

Syntax

Fortran

```fortran
CALL SPTTRF | DPTTRF | CPTTRF | ZPTTRF (n, d, e, info)
```

C and C++

```c
spttrf | dpttrf | cpttrf | zpttrf (n, d, e, info);
```

LAPACKE

```c
info = LAPACKE_spttrf | LAPACKE_dpttrf | LAPACKE_cpttrf | LAPACKE_zpttrf
(matrix_layout, n, d, e);
```

On Entry

- **matrix_layout**
  - indicates whether the input and output matrices are stored in row major order or column major order, where:
  - If `matrix_layout` = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
  - If `matrix_layout` = LAPACK_COL_MAJOR, the matrices are stored in column major order.
  - Specified as: an integer. It must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR

- **n** is the order \( n \) of tridiagonal matrix \( A \). Specified as: an integer; \( n \geq 0 \).
**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 169 on page 751.

**e** is the vector $e$ containing the subdiagonal or superdiagonal 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 $n-1$, containing numbers of the data type indicated in Table 169 on page 751.

**On Return**

**d** If $info = 0$, is the vector $d$, containing the diagonal $D$ 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 169 on page 751.

**e** If $info = 0$, is the vector $e$, as follows:

**For SPTTRF and DPTTRF**

- $e$ contains the subdiagonal elements of the unit lower bidiagonal factor $L$ in positions 1 through $n-1$ in an array referred to as E.

**For CPTTRF and ZPTTRF**

- $e$ contains the following:
  - If on entry you specified the subdiagonal of matrix $A$ in $e$, $e$ contains the subdiagonal elements of the unit bidiagonal factor $L$ in positions 1 through $n-1$ in an array, referred to as E.
  - If on entry you specified the superdiagonal of matrix $A$ in $e$, $e$ contains the subdiagonal elements of the unit bidiagonal factor $U$ in positions 1 through $n-1$ in an array, referred to as E.

Returned as: a one-dimensional array of (at least) length $n-1$, containing numbers of the data type indicated in Table 169 on page 751. It has the same length as E on entry.

**info** has the following meaning:

If $info = 0$, the subroutine completed successfully.

If $info = i$, the leading minor of order $i$ is not positive definite, and the factorization could not be completed.

Returned as:

- For SPTTRF, DPTTRF, CPTTRF, and ZPTTRF, returned as: an integer; $info \geq 0$.
- For LAPACKE_spttrf, LAPACKE_dpttrf, LAPACKE_cpttrf, and LAPACKE_zpttrf, returned as an integer function value; $info \geq 0$.

**Notes**

1. In your C program, argument $info$ must be passed by reference.

2. For a description of how real symmetric tridiagonal matrices are stored in LAPACK-symmetric-tridiagonal storage mode, see “LAPACK-Symmetric Tridiagonal Storage Mode” on page 114. For a description of how complex Hermitian tridiagonal matrices are stored in LAPACK-complex Hermitian-tridiagonal storage mode, “Complex Hermitian Tridiagonal Storage Representation” on page 116.
3. The way these subroutines handle computational errors differs from LAPACK. Like LAPACK, these subroutines use the info argument to provide information about the computational error, but they also provide an error message.

4. On both input and output, matrix A conforms to LAPACK format.

**Function**

SPTTRF and DPTTRF factor a positive definite real symmetric tridiagonal matrix stored in LAPACK-symmetric-tridiagonal storage mode:

\[ A = LDL^T \]

**Note:** Because A is symmetric, this may be considered to be a \( U^T DU \) factorization as well.

CPTTRF and ZPTTRF factor a positive definite complex Hermitian tridiagonal matrix stored in LAPACK-complex-Hermitian-tridiagonal storage mode:

- If you specify the subdiagonal of \( A \) in vector \( e \), then \( A = LDL^H \)
- If you specify the superdiagonal of \( A \) in vector \( e \), then \( A = U^H DU \)

To solve the system of equations with one or more right-hand sides, follow the call to SPTTRF, DPTTRF, CPTTRF, or ZPTTRF with a call to SPTTRS, DPTTRS, CPTTRS, or ZPTTRS, respectively.

If \( n \) is 0, no computation is performed and the subroutine returns after doing some parameter checking. See references [8 on page 1363], [44 on page 1366], and [73 on page 1367].

**Error conditions**

**Computational Errors**

Matrix \( A \) is not positive definite. For details, see the description of the info argument.

**Input-Argument Errors**

1. \( n < 0 \)

**Examples**

**Example 1**

This example shows a factorization of the positive definite real symmetric tridiagonal matrix \( A \), in the form \( A = LDL^T \):

\[
\begin{bmatrix}
  1.0 & 1.0 & 0.0 & 0.0 \\
  1.0 & 2.0 & 1.0 & 0.0 \\
  0.0 & 1.0 & 3.0 & 1.0 \\
  0.0 & 0.0 & 1.0 & 1.0 \\
\end{bmatrix}
\]

**Call Statement and Input:**

```
CALL DPTTRF( 4 , D , E , INFO )
```

```
D = ( 1.0, 2.0, 3.0, 1.0 )
E = ( 1.0, 1.0, 1.0 )
```

**Output:**
Example 2

This example shows a factorization of the positive definite complex Hermitian tridiagonal matrix \( A \), in the form \( A = LDL^H \):

\[
\begin{pmatrix}
1.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 2.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 \\
\end{pmatrix}
\]

Output:

\[
\begin{pmatrix}
1.0 & 2.0 & 3.0 & 4.0 \\
1.0 & 1.0 & 1.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 \\
\end{pmatrix}
\]

INFO = 0

Example 3

This example shows a factorization of the positive definite complex Hermitian tridiagonal matrix \( A \), in the form \( A = U^HDU \):

\[
\begin{pmatrix}
1.0 & 0.0 & 0.0 & 0.0 \\
1.0 & -1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 2.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 \\
\end{pmatrix}
\]

Output:

\[
\begin{pmatrix}
1.0 & 2.0 & 3.0 & 4.0 \\
1.0 & -1.0 & 1.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 \\
\end{pmatrix}
\]

INFO = 0
SPTTRS, DPTTRS, CPTTRS, and ZPTTRS (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Multiple Right-Hand Solve)

Purpose

SPTTRS and DPTTRS solve the tridiagonal system $AX = B$ for $X$, where $X$ and $B$ are general matrices and $A$ is a positive definite real symmetric matrix.

CPTTRS and ZPTTRS solve one of the following tridiagonal systems for $X$, where $X$ and $B$ are general matrices and $A$ is a positive definite complex Hermitian matrix.

- If, in the call to CPTTRF or ZPTTRF, you specified the subdiagonal of $A$ in $e$:
  - If $uplo = 'L'$, then this subroutine solves $AX = B$.
  - If $uplo = 'U'$, then this subroutine solves $A^TX = B$.
- If, in the call to CPTTRF or ZPTTRF, you specified the superdiagonal of $A$ in $e$:
  - If $uplo = 'L'$, then this subroutine solves $A^TX = B$.
  - If $uplo = 'U'$, then this subroutine solves $AX = B$.

These subroutines use the results of the factorization of matrix $A$, produced by a preceding call to SPTTRF, DPTTRF, CPTTRF, or ZPTTRF respectively.

Table 170. Data Types

<table>
<thead>
<tr>
<th>$d$</th>
<th>$e, B$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SPTTRS$^A$</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DPTTRS$^A$</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>Short-precision complex</td>
<td>CPTTRS$^A$</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision complex</td>
<td>ZPTTRS$^A$</td>
</tr>
</tbody>
</table>

$^A$LAPACK

Note: The input to these solve subroutines must be the output from the factorization subroutines SPTTRF, DPTTRF, CPTTRF, or ZPTTRF respectively.

Syntax

Fortran

<table>
<thead>
<tr>
<th>CALL SPTTRS</th>
<th>DPTTRS ($n$, $nrhs$, $d$, $e$, $b$, $ldb$, $info$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL CPTTRS</td>
<td>ZPTTRS ($uplo$, $n$, $nrhs$, $d$, $e$, $b$, $ldb$, $info$)</td>
</tr>
</tbody>
</table>

C and C++

<table>
<thead>
<tr>
<th>spttrs</th>
<th>dppttrs($n$, $nrhs$, $d$, $e$, $b$, $ldb$, $info$);</th>
</tr>
</thead>
<tbody>
<tr>
<td>cppttrs</td>
<td>zpttrs ($uplo$, $n$, $nrhs$, $d$, $e$, $b$, $ldb$, $info$);</td>
</tr>
</tbody>
</table>

LAPACK

| info = LAPACKE_spttrs | LAPACKE_dppttrs ($matrix_layout$, $n$, $nrhs$, $d$, $e$, $b$, $ldb$); |
| info = LAPACKE_cppttrs | LAPACKE_zpttrs ($matrix_layout$, $uplo$, $n$, $nrhs$, $d$, $e$, $b$, $ldb$); |

On Entry

$matrix_layout$

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If $matrix_layout = $LAPACK_ROW_MAJOR, the matrices are stored in row major order.
• If `matrix_layout = LAPACK_COL_MAJOR`, the matrices are stored in column major order.

Specified as: an integer. It must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR

`uplo` indicates whether \( e \) is the subdiagonal of the unit bidiagonal lower triangular factor \( L \) or superdiagonal of the unit bidiagonal upper triangular factor \( U \):

If `uplo = 'L'`, \( e \) is the subdiagonal of the unit bidiagonal lower triangular factor \( L \).

If `uplo = 'U'`, \( e \) is the superdiagonal of the unit bidiagonal upper triangular factor \( U \).

Specified as: a single character. It must be \('L' \) or \('U' \).

`n` is the order \( n \) of tridiagonal matrix \( A \). Specified as: an integer; \( n \geq 0 \).

`nrhs` is the number of right-hand sides; i.e., the number of columns of matrix \( B \).
Specified as: an integer; \( nrhs \geq 0 \).

`d` is the vector \( d \), containing part of the factorization of matrix \( A \) from SPTTRF, DPTTRF, CPTTRF, or ZPTTRF, respectively, in an array, referred to as 0.
Specified as: a one-dimensional array of (at least) length \( n \), containing numbers of the data type indicated in Table 170 on page 755.

`e` is the vector \( e \), containing the subdiagonal elements of the unit bidiagonal factor \( L \) in positions 1 through \( n-1 \) in an array, referred to as \( E \).

For SPTTRS and DPTTRS
is the vector \( e \), containing the subdiagonal elements of the unit bidiagonal factor \( L \) in positions 1 through \( n-1 \) in an array, referred to as \( E \).

For CPTTRS and ZPTTRS
is the vector \( e \), containing the subdiagonal or superdiagonal of matrix \( A \) in positions 1 through \( n-1 \) in an array, referred to as \( E \).

• If `uplo = 'L'`, \( e \) contains the subdiagonal elements of the unit bidiagonal factor \( L \) in positions 1 through \( n-1 \) in an array, referred to as \( E \).

• If `uplo = 'U'`, \( e \) contains the superdiagonal elements of the unit bidiagonal factor \( U \) in positions 1 through \( n-1 \) in an array, referred to as \( E \).

Specified as: a one-dimensional array, of (at least) length \( n-1 \), containing numbers of the data type indicated in Table 170 on page 755.

`b` is the matrix \( B \) of right-hand side vectors. Specified as the `ldb` by (at least) `nrhs` array, containing numbers of the data type indicated in Table 170 on page 755.

`ldb` is the leading dimension of the array specified for \( B \). Specified as: an integer; `ldb > 0` and `ldb \equiv n`.

On Return

`b` is the general matrix \( X \), containing the solutions to the system.

Returned as: an `ldb` by (at least) `nrhs` array, containing numbers of the data type indicated in Table 170 on page 755.
**info**

*info* has the following meaning:

If \( \text{info} = 0 \), the solve completed successfully.

Returned as:

- For SPTTRS, DPTTRS, CPTTRS, and ZPTTRS, returned as: an integer; \( \text{info} \geq 0 \).
- For LAPACK_spttrs, LAPACK_dpttrs, LAPACK_cpttrs, and LAPACK_zpttrs, returned as an integer function value; \( \text{info} \geq 0 \).

**Notes**

1. In your C program, argument *info* must be passed by reference.
2. All subroutines accept lowercase letters for the *uplo* argument.
3. For a description of how real symmetric tridiagonal matrices are stored in LAPACK-symmetric-tridiagonal storage mode, see "LAPACK-Symmetric-Triviaonal Storage Mode" on page 114. For a description of how complex Hermitian tridiagonal matrices are stored in LAPACK-complex Hermitian-tridiagonal storage mode, "Complex Hermitian Tridiagonal Storage Representation" on page 116.
4. The scalar data specified for input argument *n* for these subroutines must be the same as the corresponding input argument specified for SPTTRF, DPTTRF, CPTTRF, or ZPTTRF, respectively.
5. The array data specified for input arguments *d* and *e* for these subroutines must be the same as the corresponding output arguments for SPTTRF, DPTTRF, CPTTRF, and ZPTTRF, respectively.

**Function**

SPTTRS and DPTTRS solve the tridiagonal system \( AX = B \) for \( X \), where \( X \) and \( B \) are general matrices and \( A \) is a positive definite real symmetric matrix.

CPTTRS and ZPTTRS solve one of the following tridiagonal systems for \( X \), where \( X \) and \( B \) are general matrices and \( A \) is a positive definite complex Hermitian matrix.

- If, in the call to CPTTRF or ZPTTRF, you specified the subdiagonal of \( A \) in *e*:
  - If *uplo* = 'L', then this subroutine solves \( AX = B \).
  - If *uplo* = 'U', then this subroutine solves \( A^T X = B \).
- If, in the call to CPTTRF or ZPTTRF, you specified the superdiagonal of \( A \) in *e*:
  - If *uplo* = 'L', then this subroutine solves \( A^T X = B \).
  - If *uplo* = 'U', then this subroutine solves \( AX = B \).

These subroutines use the results of the factorization of matrix \( A \), produced by a preceding call to SPTTRF, DPTTRF, CPTTRF, or ZPTTRF respectively. For a description of how \( A \) is factored, see "SPTTRF, DPTTRF, CPTTRF, and ZPTTRF (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization)" on page 751.

If \( n \) or *nrhs* is 0, no computation is performed. See references [8 on page 1363] and [44 on page 1366].
Error conditions

Computational Errors
None

Note: If the factorization performed by SPTTRF, DPTTRF, CPTTRF, or ZPTTRF failed because matrix \( A \) 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. \( \text{uplo} \neq 'U' \) or 'L'
2. \( n < 0 \)
3. \( \text{nrhs} < 0 \)
4. \( \text{ldb} \leq 0 \)
5. \( n > \text{ldb} \)

Examples

Example 1
This example shows how to solve the system of linear equations \( AX = B \) where positive definite real symmetric tridiagonal matrix \( A \) is the same matrix factored in Example 1 for DPTTRF in the form \( LDL^T \).

Call Statement and Input:

```plaintext
CALL DPTTRS( 4, 2, D, E, B, 4, INFO )
D = (same as output D in Example 1)
E = (same as output E in Example 1)
B = [ 2.0 -2.0 ]
    [ 4.0 -3.0 ]
    [ 5.0  0.0 ]
    [ 2.0  1.0 ]
```

Output:

```plaintext
B = [ 1.0 -1.0 ]
    [ 1.0 -1.0 ]
    [ 1.0  0.0 ]
    [ 1.0  1.0 ]
INFO = 0
```

Example 2
This example shows how to solve the system of linear equations \( AX = B \) where positive definite complex Hermitian tridiagonal matrix \( A \) is the same matrix factored in Example 2 for ZPTTRF in the form \( LDL^H \).

Call Statement and Input:

```plaintext
CALL ZPTTRS( 'L', 4, 3, D, E, B, 4, INFO )
D = (same as output D in Example 2)
E = (same as output E in Example 2)
```

┌┐
| |
└┘
Example 3
This example shows how to solve the system of linear equations $A^T X = B$
where positive definite complex Hermitian tridiagonal matrix $A$ is the same
matrix factored in Example 2 for ZPTTRF in the form $LDL^T$.

Call Statement and Input:

$\begin{align*}
\text{UPLO} & \quad \text{NRHS} \quad \text{D} \quad \text{E} \quad \text{B} \quad \text{LDB} \quad \text{INFO} \\
\text{CALL ZPTTRS}('U', 4, 3, D, E, B, 4, INFO)
\end{align*}$

$D =$ (same as output $D$ in Example 2)

$E =$ (same as output $E$ in Example 2)

$B =$

\[
\begin{pmatrix}
(2.0, -1.0) & (3.0, 1.0) & (1.0, -3.0) \\
(7.0, -1.0) & (8.0, 6.0) & (6.0, -8.0) \\
(12.0, -1.0) & (13.0, 11.0) & (11.0, -13.0) \\
(13.0, 3.0) & (10.0, 16.0) & (16.0, -10.0)
\end{pmatrix}
\]

Output:

\[
\begin{pmatrix}
(1.0, 0.0) & (1.0, 1.0) & (1.0, -1.0) \\
(1.0, 0.0) & (1.0, 1.0) & (1.0, -1.0) \\
(1.0, 0.0) & (1.0, 1.0) & (1.0, -1.0) \\
(1.0, 0.0) & (1.0, 1.0) & (1.0, -1.0)
\end{pmatrix}
\]

INFO = 0

Example 4
This example shows how to solve the system of linear equations $AX = B$
where positive definite complex Hermitian tridiagonal matrix $A$ is the same
matrix factored in Example 3 for ZPTTRF in the form $U^H D U$.

Call Statement and Input:

$\begin{align*}
\text{UPLO} & \quad \text{NRHS} \quad \text{D} \quad \text{E} \quad \text{B} \quad \text{LDB} \quad \text{INFO} \\
\text{CALL ZPTTRS}('U', 4, 3, D, E, B, 4, INFO)
\end{align*}$

$D =$ (same as output $D$ in Example 3)

$E =$ (same as output $E$ in Example 3)

$B =$

\[
\begin{pmatrix}
(2.0, -1.0) & (3.0, 1.0) & (1.0, -3.0) \\
(7.0, -1.0) & (8.0, 6.0) & (6.0, -8.0) \\
(12.0, -1.0) & (13.0, 11.0) & (11.0, -13.0) \\
(13.0, 3.0) & (10.0, 16.0) & (16.0, -10.0)
\end{pmatrix}
\]

Output:

\[
\begin{pmatrix}
(3.00, -3.33) & (6.33, -0.33) & (-0.33, -6.33) \\
(0.66, 1.66) & (-1.00, 2.33) & (2.33, 1.00) \\
(0.83, -1.50) & (2.33, -0.66) & (-0.66, -2.33) \\
(1.50, 1.00) & (0.50, 2.50) & (2.50, -0.50)
\end{pmatrix}
\]

INFO = 0
Output:

\[
B = \begin{bmatrix}
(1.0, 0.0) & (1.0, 1.0) & (1.0, -1.0) \\
(1.0, 0.0) & (1.0, 1.0) & (1.0, -1.0) \\
(1.0, 0.0) & (1.0, 1.0) & (1.0, -1.0) \\
(1.0, 0.0) & (1.0, 1.0) & (1.0, -1.0)
\end{bmatrix}
\]

INFO = 0

**Example 5**

This example shows how to solve the system of linear equations \(A^T X = B\) where positive definite complex Hermitian tridiagonal matrix \(A\) is the same matrix factored in Example 3 for ZPTTRF in the form \(U^H D U\).

Call Statement and Input:

```
CALL ZPTTRS('U', 4, 3, D, E, B, 4, INFO)
```

\(D\) = (same as output \(D\) in Example 3)

\(E\) = (same as output \(E\) in Example 3)

\[
B = \begin{bmatrix}
(2.0, -1.0) & (3.0, 1.0) & (1.0, -3.0) \\
(7.0, -1.0) & (8.0, 6.0) & (6.0, -8.0) \\
(12.0, -1.0) & (13.0, 11.0) & (11.0, -13.0) \\
(13.0, 3.0) & (10.0, 16.0) & (16.0, -10.0)
\end{bmatrix}
\]

Output:

\[
B = \begin{bmatrix}
(3.00, -3.33) & (6.33, -0.33) & (-0.33, -6.33) \\
(0.66, 1.66) & (-1.00, 2.33) & (2.33, 1.00) \\
(0.83, -1.50) & (2.33, -0.66) & (-0.66, -2.33) \\
(1.50, 1.00) & (0.50, 2.50) & (2.50, -0.50)
\end{bmatrix}
\]

INFO = 0
SGBF and DGBF (General Band Matrix Factorization)

Purpose

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 171. Data Types

<table>
<thead>
<tr>
<th>$A$</th>
<th>Subroutines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGBF</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGBF</td>
</tr>
</tbody>
</table>

Note: The output from these factorization subroutines should be used only as input to the solve subroutines SGBS and DGBS, respectively.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SGBF</th>
<th>DGBF ($agb$, $lda$, $n$, $ml$, $mu$, $ipvt$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>sgbf</td>
<td>dgbf ($agb$, $lda$, $n$, $ml$, $mu$, $ipvt$);</td>
</tr>
</tbody>
</table>

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 171 where $lda \geq 2ml+mu+16$.

$lda$

is the leading dimension of the array specified for $agb$. Specified as: an integer; $lda > 0$ and $lda \geq 2ml+mu+16$.

$n$

is the order of the matrix $A$. Specified as: an integer; $n > ml$ and $n > mu$.

$ml$

is the lower band width $ml$ of the matrix $A$. Specified as: an integer; $0 \leq ml < n$.

$mu$

is the upper band width $mu$ of the matrix $A$. Specified as: an 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 762. Returned as: an $lda$ by (at least) $n$ array, containing numbers of the data type indicated in Table 171.

$ipvt$

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 integers.
Notes

1. \textit{ipvt} is not a permutation vector in the strict sense. It is used to record column interchanges in \textit{L} due to partial pivoting and to improve performance.
2. The entire \textit{lda} by \textit{n} array specified for \textit{agb} must remain unchanged between calls to the factorization and solve subroutines.
3. This subroutine can be used for tridiagonal matrices (\textit{ml} = \textit{mu} = 1); however, the tridiagonal subroutines \textit{SGTF/DGTF} and \textit{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 100.

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 \textit{SGBS} or \textit{DGBS}, respectively, to solve the system of equations. See reference [46 on page 1366].

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 \textit{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 68.

Input-Argument Errors

1. \textit{lda} \(\leq 0\)
2. \textit{ml} < 0
3. \textit{ml} \(\geq n\)
4. \textit{mu} < 0
5. \textit{mu} \(\geq n\)
6. \textit{lda} < \(2\textit{ml}+\textit{mu}+16\)

Examples

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:

\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
4.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 5.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 6.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 7.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 8.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 9.0 & 1.0 & 1.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 10.0 & 11.0 & 12.0 \\
\end{bmatrix}
\]

Matrix A is stored in general-band storage mode in the two-dimensional array AGB of size LDA by N, where LDA = 2ml+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) and SGEFCD and DGEFCD (see Example).

Call Statement and Input:

```
AGB  LDA  N  ML  MU  IPVT )
```

```
CALL SGBF( AGB , 23 , 9 , 2 , 3 , IPVT )
```

Output:
AGB

┌
|
|
|
|
|
|
|
|
|
|
|
= |
|
|
|
|
|
|
|
|
|
|
|
└

0.0000
0.0000
0.0000
0.0000
0.0000
0.2500
0.0000
0.2500
0.2500
0.0000
0.0000
0.0000
0.0000
0.0000
0.0000
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
0.2000
0.1500
0.1500
0.0000
0.0000
0.0000
0.0000
0.0000
0.0000
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
0.1600
0.0000
0.1000
0.1000
0.0000
0.0000
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

764

0.0000 0.0000
0.0000 1.0000
1.0000 1.0000
1.0000 1.0000
1.0000 1.0000
0.1400 0.1250
0.0714 0.0000
0.0714 -0.0714
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
0.0000 0.0000
0.0000 0.0000
0.0000 0.0000

1.0000
1.0000
1.0000
1.0000
1.0000
0.1100
-0.0556
-0.0694
0.0000
0.0000
0.0000
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 |
1.0000
1.0000 1.0000 |
1.0000
1.0000 12.0000 |
1.0000 11.0000
0.3111 |
0.1000
5.5380 -325.00 |
-0.0306
0.9385 0.0000 |
-0.0194
0.0000 0.0000 |
0.0000
0.0000 0.0000 |
0.0000
0.0000 0.0000 |
0.0000
0.0000 0.0000 |
0.0000
0.0000 0.0000 |
0.0000
0.0000 0.0000 |
0.0000
0.0000 0.0000 |
0.0000
0.0000 0.0000 |
0.0000
0.0000 0.0000 |
0.0000
0.0000 0.0000 |
0.0000
0.0000 0.0000 |
0.0000
0.0000 0.0000 |
0.0000
0.0000 0.0000 |
0.0000
0.0000 0.0000 |
0.0000
0.0000 0.0000 |
0.0000
0.0000 0.0000 |
┘

= (2, -65534, -131070, -196606, -262142, -327678, -327678,
-327680, -327680)

ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference


SGBS and DGBS (General Band Matrix Solve)

Purpose

These subroutines solve the system $Ax = b$ for $x$, where $A$ is a general band matrix, and $x$ and $b$ are vectors. They use the results of the factorization of matrix $A$, produced by a preceding call to SGBF or DGBF, respectively.

Table 172. Data Types

<table>
<thead>
<tr>
<th>$A$, $b$, $x$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGBS</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGBS</td>
</tr>
</tbody>
</table>

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++: sgb | 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 integer; $lda > 0$ and $lda \geq 2ml+mu+16$.

$lda$

is the leading dimension of the array specified for $agb$. Specified as: an integer; $lda > 0$ and $lda \geq 2ml+mu+16$.

$n$

is the order of the matrix $A$. Specified as: an integer; $n > ml$ and $n > mu$.

$ml$

is the lower band width $ml$ of the matrix $A$. Specified as: an integer; $0 \leq ml < n$.

$mu$

is the upper band width $mu$ of the matrix $A$. Specified as: an 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 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 161 on page 712.

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 161 on page 712.
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 75.
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 100.

Function
The real system Ax = b is solved for x, where A is a real general band matrix, stored in general-band storage mode, and x and b are vectors. These subroutines use the results of the factorization of matrix A, produced by a preceding call to SGBF or DGBF, respectively. The transformed matrix A, used by this computation, consists of the upper triangular matrix U and the multipliers necessary to construct L using ipvt, as defined in “Function” on page 762. See reference 46 on page 1366.

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 ≤ 0
2. ml < 0
3. ml ≥ n
4. mu < 0
5. mu ≥ n
6. lda < 2ml+mu+16

Examples

Example
This example shows how to solve the system Ax = b, where general band matrix A is the same matrix factored in Example 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:

<table>
<thead>
<tr>
<th>AGB</th>
<th>LDA</th>
<th>N</th>
<th>ML</th>
<th>MU</th>
<th>IPVT</th>
<th>BX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>23</td>
<td>9</td>
<td>2</td>
<td>3</td>
<td>IPVT</td>
</tr>
</tbody>
</table>

CALL SGBS( AGB, 23, 9, 2, 3, IPVT, BX )

766 ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
IPVT = (2, -65354, -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)

Output:
BX = (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)

Purpose

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 173. Data Types

<table>
<thead>
<tr>
<th>$A$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SPBF and SPBCHF</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DPBF and DPBCHF</td>
</tr>
</tbody>
</table>

Note:
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

```fortran
CALL SPBF | DPBF | SPBCHF | DPBCHF (apb, lda, n, m)
```

C and C++

```c
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 173. See "Notes" on page 769.

$lda$

is the leading dimension of the array specified for $apb$. Specified as: an integer; $lda > 0$ and $lda > m$.

$n$

is the order $n$ of matrix $A$. Specified as: an integer; $n > m$.

$m$

is the half band width of the matrix $A$. Specified as: an 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 173 on page 768. 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 107.

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, “Coding Your Program,” on page 133.
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, “Coding Your Program,” on page 133.

Input-Argument Errors
1. lda ≤ 0
2. m < 0
3. \( m \geq n \)
4. \( m \geq lda \)

Examples

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:

\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 2.0 & 2.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 2.0 & 3.0 & 2.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 1.0 & 2.0 & 3.0 & 2.0 & 1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 1.0 & 2.0 & 3.0 & 2.0 & 1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 1.0 & 2.0 & 3.0 & 2.0 & 1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & 3.0 & 2.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & 3.0 & 2.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & 3.0
\end{bmatrix}
\]

and on output, matrix \( A \) is:

\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 1.0
\end{bmatrix}
\]

where array location \( APB(2,9) \) is set to 0.0.

Call Statement and Input:

```
CALL SPBF( APB, 3, 9, 2 )
```

\[
APB = \begin{bmatrix}
1.0 & 2.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 & 0.0 \\
1.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 0.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0
\end{bmatrix}
\]

Output:

\[
APB = \begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0
\end{bmatrix}
\]

Example 2

This example shows a Cholesky factorization of the same matrix used in Example 1.

Call Statement and Input:

```
CALL SPCHF( APB, 3, 9, 2 )
```

\[
APB = \begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0
\end{bmatrix}
\]

APB = (same as input APB in Example 1)
Output:

\[
\text{APB} = \begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & . \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & . & . \\
\end{bmatrix}
\]
SPBS, DPBS, SPBCHS, and DPBCHS (Positive Definite Symmetric Band Matrix Solve)

Purpose

These subroutines solve the system $Ax = b$ for $x$, where $A$ is a positive definite symmetric band matrix, and $x$ and $b$ are vectors. They use the results of the factorization of matrix $A$, 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 174. Data Types

<table>
<thead>
<tr>
<th>$A$, $b$, $x$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SPBS and SPBCHS</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DPBS and DPBCHS</td>
</tr>
</tbody>
</table>

Note:

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 768.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C and C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL SPBS</td>
<td>DPBS</td>
</tr>
</tbody>
</table>

On Entry

- **apb** is the factorization of matrix $A$, 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 174. See “Notes” on page 773.

- **lda** is the leading dimension of the array specified for apb. Specified as: an integer; $lda > 0$ and $lda > m$.

- **n** is the order $n$ of matrix $A$. Specified as: an integer; $n > m$.

- **m** is the half band width of the matrix $A$. Specified as: an integer; $0 \leq m < n$.

- **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 174.

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 174.
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 75.
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 107.

Function
The system \( Ax = b \) is solved for \( x \), where \( A \) is a positive definite symmetric band matrix, stored in lower-band-packed storage mode, and \( x \) and \( b \) are vectors. These subroutines use the results of the factorization of matrix \( A \), 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. \( m \geq lda \)

Examples
Example 1
This example shows how to solve the system \( Ax = b \), where matrix \( A \) is the same matrix factored in the Example 1 for SPBF and DPBF, using Gaussian elimination.

Call Statement and Input:

\[
\text{APB LDA N M BX}
\]
\[
\text{CALL SPBS( APB , 3 , 9 , 2 , BX )}
\]

\[
\begin{align*}
\text{APB} & = \text{(same as output APB in Example 1)} \\
\text{BX} & = \text{(3.0, 6.0, 9.0, 9.0, 9.0, 9.0, 9.0, 8.0, 6.0)}
\end{align*}
\]

Output:

\[
\begin{align*}
\text{BX} & = \text{(1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)}
\end{align*}
\]
This example shows how to solve the system $Ax = b$, where matrix $A$ is the same matrix factored in the Example 2 for SPBCHF and DPBCHF, using Cholesky factorization.

Call Statement and Input:

```
CALL SPBCHS( APB, 3, 9, 2, BX )
```

$APB =$  (same as output APB in Example 2)

$BX =$  (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)

Purpose

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 175. Data Types

<table>
<thead>
<tr>
<th>c, d, e, f</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGTF</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGTF</td>
</tr>
</tbody>
</table>

Note: The output from these factorization subroutines should be used only as input to the solve subroutines SGTS and DGTS, respectively.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SGTF</th>
<th>DGTF (n, c, d, e, f, ipvt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>sgtf</td>
<td>dgft (n, c, d, e, f, ipvt);</td>
</tr>
</tbody>
</table>

On Entry

- \( n \) is the order \( n \) of tridiagonal matrix \( A \). Specified as: an 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 175.
- \( d \) 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 175.
- \( e \) 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 175.
- \( f \) 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 175.
- \( d \) 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 175.
- \( e \) 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 175.
\( f \) 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 175 on page 775.

\( ipvt \)

is the integer vector \( ipvt \) of length \( n \), containing the pivot information. Returned as: a one-dimensional array of (at least) length \( n \), containing integers.

**Notes**

1. For a description of how tridiagonal matrices are stored, see "General Tridiagonal Matrix" on page 112.
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 [51 on page 1366], [63 on page 1367], [64 on page 1367], and [109 on page 1369]. 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 68.

**Input-Argument Errors**

\( n < 0 \)

**Examples**

**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:
CALL DGETF( 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)

**Purpose**

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 176. Data Types**

<table>
<thead>
<tr>
<th>$c$, $d$, $e$, $f$, $b$, $x$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGTS</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGTS</td>
</tr>
</tbody>
</table>

**Note:** The input to these solve subroutines must be the output from the factorization subroutines SGTF and DGTF, respectively.

**Syntax**

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SGTS</th>
<th>DGT S ($n$, $c$, $d$, $e$, $f$, $ipvt$, $bx$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>sgt s</td>
<td>dgts ($n$, $c$, $d$, $e$, $f$, $ipvt$, $bx$)</td>
</tr>
</tbody>
</table>

**On Entry**

- $n$ is the order $n$ of tridiagonal matrix $A$. Specified as: an 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 176.
- $d$ 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 176.
- $e$ 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 176.
- $f$ 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 176.
- $ipvt$ 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 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 176. For details on specifying the length, see “Notes” on page 779.

**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 176 on page 778. For details about the length, see “Notes.”

Notes
1. For a description of how tridiagonal matrices are stored, see “General Tridiagonal Matrix” on page 112.
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 $Ax = b$, where $A$ is a general tridiagonal matrix. See references 51 on page 1366, 63 on page 1367, 64 on page 1367, and 109 on page 1369.

Error conditions

Computational Errors
None

Input-Argument Errors

$n < 0$

Examples

Example
This example solves the tridiagonal system $Ax = b$, where matrix $A$ is the same matrix factored in Example for SGTF and DGTF, and where:

\[
\begin{align*}
b &= (4.0, 6.0, 6.0, 4.0) \\
x &= (1.0, 1.0, 1.0, 1.0)
\end{align*}
\]

Call Statement and Input:

```
CALL DGTS( 4, C, D, E, F, IPVT, BX )
```

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|}
C & D & E & F & IPVT & BX \\
\hline
\end{array}
\]

\[
\begin{align*}
C &= \text{(same as output C in Example)} \\
D &= \text{(same as output D in Example)} \\
E &= \text{(same as output E in Example)} \\
F &= \text{(same as output F in Example)} \\
IPVT &= \text{(same as output IPVT in Example)} \\
BX &= (4.0, 6.0, 6.0, 4.0, . )
\end{align*}
\]

Output:

\[
\begin{align*}
BX &= (1.0, 1.0, 1.0, 1.0, . )
\end{align*}
\]
SGTNP, DGTNP, CGTNP, and ZGTNP (General Tridiagonal Matrix Combined Factorization and Solve with No Pivoting)

Purpose

These subroutines solve the tridiagonal system \( Ax = b \) using Gaussian elimination, where tridiagonal matrix \( A \) is stored in tridiagonal storage mode.

Table 177. Data Types

<table>
<thead>
<tr>
<th>c, d, e, b, x</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGTNP</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGTNP</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGTNP</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGTNP</td>
</tr>
</tbody>
</table>

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);
```

On Entry

- \( n \) is the order \( n \) of tridiagonal matrix \( A \). Specified as: an 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 177. 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 177. 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 177. 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 177.

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 177.
Notes

For a description of how tridiagonal matrices are stored, see “General Tridiagonal Matrix” on page 112.

Function

The solution of the tridiagonal system $Ax = b$ 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

$n < 0$

Examples

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 $Ax = b$, where $b$ is:

$$(11.0, 14.0, 17.0, 13.0)$$

and $x$ is:

$$(1.0, 1.0, 1.0, 1.0)$$

On output, arrays $C$, $D$, and $E$ are overwritten.

Call Statement and Input:

```fortran
CALL DGTNP( 4, C, D, E, BX )
```

$C = ( . , 1.0, 2.0, 3.0 )$

$D = ( 7.0, 8.0, 9.0, 10.0 )$

$E = ( 4.0, 5.0, 6.0, . )$

$BX = ( 11.0, 14.0, 17.0, 13.0 )$
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 $Ax = b$, where $b$ is:

$$((-11.0,19.0), (-14.0,50.0), (-17.0,93.0), (-13.0,85.0))$$

and $x$ 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:

```fortran
CALL ZSTNP( 4, C, D, E, BX )
```

Output:

$$
\begin{bmatrix}
(0.0, 1.0) & (1.0, 2.0) & (2.0, 3.0) & (3.0, 4.0)
\end{bmatrix}
$$
SGTNPF, DGTNPF, CGTNPF, and ZGTNPF (General Tridiagonal Matrix Factorization with No Pivoting)

Purpose

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 178. Data Types

<table>
<thead>
<tr>
<th>c, d, e</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGTNPF</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGTNPF</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGTNPF</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGTNPF</td>
</tr>
</tbody>
</table>

Note:

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);

On Entry

n is the order n of tridiagonal matrix A. Specified as: an integer; n ≥ 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 178.

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 178.

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 178.

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: an 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 178 on page 783.

\( d \) 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 178 on page 783.

\( e \) 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 178 on page 783. It has the same length as \( E \) on entry.

**Notes**

For a description of how tridiagonal matrices are stored, see “General Tridiagonal Matrix” on page 112.

**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 [91 on page 1368].

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 \( 1 \)

**Examples**

**Example 1**

This example shows a factorization of the tridiagonal matrix \( A \), of order 4:
Example 2

This example shows a factorization of the tridiagonal matrix $A$, of order 4:

\[
\begin{bmatrix}
7.0 & 7.0 & 4.0 & 4.0 \\
1.0 & 1.0 & 8.0 & 8.0 \\
0.0 & 0.0 & 2.0 & 2.0 \\
0.0 & 0.0 & 0.0 & 0.0
\end{bmatrix}
\]

Call Statement and Input:

\[
\begin{array}{cccccc}
N & C & D & E & IOPT \\
\hline
\end{array}
\]

CALL DGTNPF( 4, C, D, E, 0 )

C = ( 1.0, 1.0, 1.0, 1.0 )
D = ( 4.0, 2.0, 3.0, 1.0 )
E = ( 1.0, 1.0, 1.0, . )

Output:

C = ( 1.0, -1.0, -1.0, 1.0 )
D = ( -1.0, -1.0, -1.0, -1.0 )
E = ( 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.
SGTNPS, DGTNPS, CGTNPS, and ZGTNPS (General Tridiagonal Matrix Solve with No Pivoting)

Purpose

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 179. Data Types

<table>
<thead>
<tr>
<th>c, d, e, b, x</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGTNPS</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGTNPS</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGTNPS</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGTNPS</td>
</tr>
</tbody>
</table>

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);

On Entry

n is the order n of tridiagonal matrix A. Specified as: an integer; n ≥ 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 179.

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 179.

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 179.

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 179.

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 179.
Notes

For a description of how tridiagonal matrices are stored, see “General Tridiagonal Matrix” on page 112.

Function

The solution of tridiagonal system \( Ax = b \) is computed using the factorization produced by SGTNPF, DGTNPF, CGTNPF, or ZGTNPF, respectively. The factorization is based on Gaussian elimination. See reference [91 on page 1368].

Error conditions

Computational Errors
None

Input-Argument Errors
\( n < 0 \)

Examples

Example 1

This example finds the solution of tridiagonal system \( Ax = b \), where matrix \( A \) is the same matrix factored in Example 1 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:

```plaintext
CALL DGTNPS( 4, C, D, E, BX )
```

\( C \) = (same as output \( C \) in Example 1)
\( D \) = (same as output \( D \) in Example 1)
\( E \) = (same as output \( E \) in Example 1)
\( BX \) = \((2.0, 4.0, 5.0, 2.0)\)

Output:

\( BX = (1.0, 1.0, 1.0, 1.0)\)

Example 2

This example finds the solution of tridiagonal system \( Ax = b \), where matrix \( A \) is the same matrix factored in Example 2 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:

```plaintext
CALL ZGTNPS( 4, C, D, E, BX )
```

\( C \) = (same as output \( C \) in Example 2)
\( D \) = (same as output \( D \) in Example 2)
\( E \) = (same as output \( E \) in Example 2)
\( BX \) = \((-11.0, 19.0), (-14.0, 50.0), (-17.0, 93.0), (-13.0, 8)\)
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)

**Purpose**

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 180. Data Types**

<table>
<thead>
<tr>
<th>Type</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SPTF</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DPTF</td>
</tr>
</tbody>
</table>

**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);
```

**On Entry**

- \( n \) is the order of tridiagonal matrix \( A \). Specified as: an 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 180.
- \( d \) 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 180.
- \( iopt \) indicates the type of computation to be performed, where:
  - If \( iopt = 0 \) or 1, Gaussian elimination is used to factor the matrix.
  - Specified as: an 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 180.
- \( d \) 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 180. It has the same length as \( D \) on entry.

**Notes**

For a description of how positive definite symmetric tridiagonal matrices are stored, see “Positive Definite Symmetric Tridiagonal Matrix” on page 115.
**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 [91 on page 1368].

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

**Examples**

**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:**

```
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)

*Note*

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)

Purpose

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 181. Data Types

<table>
<thead>
<tr>
<th>c, d, b, x</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SPTS</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DPTS</td>
</tr>
</tbody>
</table>

Note: The input to these solve subroutines must be the output from the factorization subroutines SPTF and DPTF, respectively.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SPTS</th>
<th>DPTS (n, c, d, bx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>spts</td>
<td>dpts (n, c, d, bx);</td>
</tr>
</tbody>
</table>

On Entry

$n$ is the order $n$ of tridiagonal matrix $A$. Specified as: an 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 181.

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 181.

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 181.

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 181.

Notes

For a description of how tridiagonal matrices are stored, see “Positive Definite or Negative Definite Symmetric Matrix” on page 89.

Function

The solution of positive definite symmetric tridiagonal system $Ax = b$ is computed using the factorization produced by SPTF or DPTF, respectively. The factorization is based on Gaussian elimination. See reference [91 on page 1368].
Error conditions

Computational Errors
None

Input-Argument Errors

\( n < 0 \)

Examples

Example

This example finds the solution of tridiagonal system \( Ax = b \), where matrix \( A \) is the same matrix factored in Example for SPTF and DPTF. \( b \) is:

\[
(2.0, 4.0, 5.0, 2.0)
\]

and \( x \) is:

\[
(1.0, 1.0, 1.0, 1.0)
\]

Call Statement and Input:

\[
\begin{array}{cccc}
N & C & D & BX \\
\hline
\end{array}
\]

\[
\text{CALL DPTS( 4 , C , D , BX )}
\]

\[
\begin{align*}
C & = (. , -1.0 , -1.0 , -1.0) \\
D & = (-1.0 , -1.0 , -1.0 , -1.0) \\
BX & = (2.0 , 4.0 , 5.0 , 2.0)
\end{align*}
\]

Output:

\[
BX = (1.0 , 1.0 , 1.0 , 1.0)
\]
Sparse Linear Algebraic Equation Subroutines

This contains the sparse linear algebraic equation subroutine descriptions.
DGSF (General Sparse Matrix Factorization Using Storage by Indices, Rows, or Columns)

Purpose

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

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL DGSF (iopt, n, nz, a, ia, ja, lna, iparm, rparm, oparm, aux, naux)</td>
</tr>
<tr>
<td>C and C++</td>
<td>dgsf (iopt, n, nz, a, ia, ja, lna, iparm, rparm, oparm, aux, naux);</td>
</tr>
</tbody>
</table>

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: an integer; \( iopt = 0, 1, \) or \( 2 \).

\( n \)
is the order \( n \) of sparse matrix \( A \). Specified as: an integer; \( n \geq 0 \).

\( nz \)
is the number of elements in sparse matrix \( A \), stored in an array, referred to as \( A \).

Specified as: an 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 integers; \( IA(i) \geq 1 \). See “Sparse Matrix” on page 116 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 integers; \( JA(i) \geq 1 \). See “Sparse Matrix” on page 116 for more information on storage techniques.
**lna**

is the length of the arrays specified for \( a, ia, \) and \( ja \).

Specified as: an integer; \( \text{lna} > 2nz \). If you do not specify a sufficient amount, it results in an error. See "Error conditions" on page 797.

The size of \( \text{lna} \) depends on the structure of the input matrix. The requirement that \( \text{lna} > 2nz \) does not guarantee a successful run of the program. If the input matrix is expected to have many fill-ins, \( \text{lna} \) should be set larger. Larger \( \text{lna} \) may result in a performance improvement.

For details on how \( \text{lna} \) relates to storage compressions, see "Performance and Accuracy Considerations" on page 525.

**iparm**

is an array of parameters, \( \text{IPARM}(i) \), where:

- **IPARM(1)** determines whether the default values for \( \text{iparm} \) and \( \text{rparm} \) are used by this subroutine.
  
  If \( \text{IPARM}(1) = 0 \), the following default values are used:

  \[
  \begin{align*}
  \text{IPARM}(2) &= 10 \\
  \text{IPARM}(3) &= 1 \\
  \text{IPARM}(4) &= 0 \\
  \text{RPARM}(1) &= 10^{-12} \\
  \text{RPARM}(2) &= 0.1
  \end{align*}
  \]

  If \( \text{IPARM}(1) = 1 \), the default values are not used.

- **IPARM(2)** determines the number of minimal Markowitz counts that are examined to determine a pivot. (See reference \[121 on page 1370].)

- **IPARM(3)** has the following meaning, where:

  If \( \text{IPARM}(3) = 0 \), this subroutine checks the values in arrays \( IA \) and \( JA \).
  
  If \( \text{IPARM}(3) = 1 \), this subroutine assumes that the input values are correct in arrays \( IA \) and \( JA \).

- **IPARM(4)** has the following meaning, where:

  If \( \text{IPARM}(4) = 0 \), this computation is not performed.
  
  If \( \text{IPARM}(4) = 1 \), 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 \( \text{OPARM}(2) \) and \( \text{OPARM}(3) \), respectively.

- **IPARM(5)** is reserved.

Specified as: an array of (at least) length 5, containing integers, where the \( \text{iparm} \) values must be:

\[
\begin{align*}
\text{IPARM}(1) &= 0 \text{ or } 1 \\
\text{IPARM}(2) &\geq 1 \\
\text{IPARM}(3) &= 0 \text{ or } 1 \\
\text{IPARM}(4) &= 0 \text{ or } 1
\end{align*}
\]

**rparm**

is an array of parameters, \( \text{RPARM}(i) \), 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:

\[
\begin{align*}
RPARM(1) & \geq 0.0 \\
0.0 & \leq RPARM(2) \leq 1.0
\end{align*}
\]

For additional information about *rparm*, see "Performance and Accuracy Considerations" on page 525.

**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: an integer.

For 32-bit integer arguments

\[naux \geq 10n+100.\]

For 64-bit integer arguments

\[naux \geq 18n+100.\]

**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 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 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 116.
2. You have the option of having the minimum required value for \( n_{aux} \) dynamically returned to your program. For details, see “Using Auxiliary Storage in ESSL” on page 51.

**Function**

The matrix \( A \) is factored by Gaussian elimination, using a modified Markowitz count with threshold pivoting to compute the sparse LU factorization of \( A \):

\[
LU = PAQ
\]

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 [16 on page 1364], [56 on page 1366], and [112 on page 1370].

**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 \( ina \).

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 68:
   - 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. \( iopt \neq 0, 1, \) or 2
2. \( n < 0 \)
3. \( nz \leq 0 \)
4. \( lna \leq 2nz \)
5. \( IPARM(1) \neq 0 \) or 1
6. \( IPARM(2) \leq 0 \)
7. \( IPARM(3) \neq 0 \) or 1
8. \( IPARM(4) \neq 0 \) or 1
9. \( RPARM(1) < 0.0 \)
10. \( RPARM(2) < 0.0 \) or \( RPARM(2) > 1.0 \)
11. \( iopt = 1 \) and \( ia(i) \geq ia(i+1), \) \( i = 1, n \)
12. \( iopt = 2 \) and \( ja(i) \geq ja(i+1), \) \( i = 1, n \)
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.

Examples

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):

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
\text{IOPT} & \text{N} & \text{NZ} & \text{A} & \text{IA} & \text{JA} & \text{LNA} & \text{IPARM} & \text{RPARM} & \text{OPARM} & \text{AUX} & \text{NAUX} \\
\hline
\text{CALL DGSF(} & 0, & 5, & 13, & A, & IA, & JA, & 27, & IPARM, & RPARM, & OPARM, & AUX, & 150 \text{ )} \\
\hline
\end{array}
\]

798 ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
Call Statement and Input (Storage-By-Columns):

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|}
\hline
IOPT & N & NZ & A & IA & JA & LNA & IPARM & RPARM & OPARM & AUX & NAUX \\
\hline
\end{array}
\]

CALL DGSF( 1, 5, 13, A, IA, JA, 27, IPARM, RPARM, OPARM, AUX, 150 )

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|}
\hline
A & = & (2.0, 1.0, 1.0, 3.0, 4.0, 1.0, 5.0, 2.0, 2.0, 1.0, 1.0, \\
& & & & & & & & & \ldots, \\
& & & & & & & & & 4.0, 3.0, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots ) \\
\hline
IA & = & (1, 2, 2, 3, 4, 4, 4, 5, 5, 1, 2, \ldots, \ldots, \\
& & & & & & & & & \ldots, \ldots, \ldots, \ldots, \ldots, \ldots ) \\
\hline
JA & = & (1, 1, 2, 3, 4, 4, 5, 1, 2, 3, 4, 3, 5, \ldots, \ldots, \\
& & & & & & & & & \ldots, \ldots, \ldots, \ldots, \ldots, \ldots ) \\
\hline
IPARM & = & (1, 3, 1, 1) \\
\hline
RPARM & = & (1.0-12, 0.100) \\
\hline
\end{array}
\]

Call Statement and Input (Storage-By-Rows):

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|}
\hline
IOPT & N & NZ & A & IA & JA & LNA & IPARM & RPARM & OPARM & AUX & NAUX \\
\hline
\end{array}
\]

CALL DGSF( 2, 5, 13, A, IA, JA, 27, IPARM, RPARM, OPARM, AUX, 150 )

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|}
\hline
A & = & (2.0, 1.0, 2.0, 1.0, 2.0, 4.0, 3.0, 1.0, 4.0, 1.0, 1.0, \\
& & & & & & & & & 3.0, 5.0, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \\
& & & & & & & & & \ldots, \ldots, \ldots, \ldots, \ldots, \ldots ) \\
\hline
IA & = & (1, 3, 6, 8, 12, 14, \ldots, \ldots, \ldots, \\
& & & & & & & & & \ldots, \ldots, \ldots, \ldots, \ldots, \ldots ) \\
\hline
JA & = & (1, 3, 1, 2, 5, 3, 4, 1, 2, 4, 5, 3, 4, \ldots, \ldots, \\
& & & & & & & & & \ldots, \ldots, \ldots, \ldots, \ldots, \ldots ) \\
\hline
IPARM & = & (1, 3, 1, 1) \\
\hline
RPARM & = & (1.0-12, 0.100) \\
\hline
\end{array}
\]

Output:

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|}
\hline
A & = & (0.5, \ldots, 0.3, 1.0, \ldots, 1.0, \ldots, 3.0, \ldots, \ldots, \ldots, 1.0, \\
& & & & & & & & & 1.0, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, -1.7, -0.5, -1.0, -1.0, \\
& & & & & & & & & 4.0, -3.0, -4.0) \\
\hline
IA & = & (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \ldots, \ldots, \\
& & & & & & & & & 2, 1, 1, 3, 3, 5, 5) \\
\hline
JA & = & (1, 0, 5, 2, 0, 4, 0, 2, 0, 0, 0, 3, 4, \ldots, \ldots, \\
& & & & & & & & & \ldots, \ldots, 4, 2, 4, 1, 3, 1) \\
\hline
OPARM & = & (1.000000, 0.333333, 3.000000) \\
\hline
\end{array}
\]

Note: On input, a "." means that you do not have to store a value in that position in the array. However, the storage position is required and may be overwritten during the computation. On output, a "." means that the value in that position in the array is not significant.
DGSS (General Sparse Matrix or Its Transpose Solve Using Storage by Indices, Rows, or Columns)

Purpose

This subroutine solves either of the following systems:

\[ Ax = b \]
\[ A^T x = b \]

where \( A \) is a sparse matrix, \( A^T \) is the transpose of sparse matrix \( A \), and \( x \) and \( b \) are vectors. DGSS uses the results of the factorization of matrix \( A \), produced by a preceding call to DGSF.

Note: The input to this solve subroutine must be the output from the factorization subroutine, DGSF.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL DGSS (jopt, n, a, ia, ja, lna, bx, aux, naux)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>dgss (jopt, n, a, ia, ja, lna, bx, aux, naux);</td>
</tr>
</tbody>
</table>

On Entry

\( jopt \)

indicates the type of computation to be performed, where:

- If \( jopt = 0 \), \( Ax = b \) is solved, where the right-hand side is not sparse.
- If \( jopt = 1 \), \( A^T x = b \) is solved, where the right-hand side is not sparse.
- If \( jopt = 10 \), \( Ax = b \) is solved, where the right-hand side is sparse.
- If \( jopt = 11 \), \( A^T x = b \) is solved, where the right-hand side is sparse.

Specified as: an integer; \( jopt = 0, 1, 10, \) or \( 11 \).

\( n \)

is the order \( n \) of sparse matrix \( A \). Specified as: an integer; \( n \geq 0 \).

\( a \)

is the factorization of sparse matrix \( A \), 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 integers.

\( ja \)

is the array, referred to as \( JA \), produced by a preceding call to DGSF.

Specified as: an array of length \( lna \), containing 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: an integer; \( lna > 0 \).

\( bx \)

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 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: an integer.

For 32-bit integer arguments

\[ naux \geq 10n+100. \]

For 64-bit integer arguments

\[ naux \geq 18n+100. \]

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 integers.

bx

is the solution vector 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 51.

Function

The system \( Ax = b \) is solved for x, where A is a sparse matrix and x and b are vectors. Depending on the value specified for the jopt argument, DGSS can also solve the system \( A^T x = b \), where \( A^T \) is the transpose of sparse matrix A.

If the value specified for the jopt argument is 0 or 10, the following equation is solved:

\[ Ax = b \]

If the value specified for the jopt argument is 1 or 11, the following equation is solved:

\[ A^T x = b \]

DGSS uses the results of the factorization of matrix A, produced by a preceding call to DGSF. The transformed matrix A consists of the upper triangular matrix U and the lower triangular matrix L.

See references [16 on page 1364], [56 on page 1366], and [112 on page 1370].
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.

Examples
Example 1
This example shows how to solve the system Ax = b, where matrix 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 A is:
┌
|
|
|
|
|
└

2.0
1.0
0.0
2.0
0.0

0.0
1.0
0.0
2.0
0.0

4.0
0.0
3.0
0.0
1.0

0.0
0.0
4.0
1.0
1.0

0.0
3.0
0.0
5.0
0.0

┐
|
|
|
|
|
┘

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
IA
JA
BX

= (0.5, . , 0.3, 1.0, . , 1.0, . , 3.0, .
1.0, . , . , . , . , . , . , . , -1.7,
4.0, -3.0, -4.0)
= (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
. , . , . , 2, 1, 1, 3, 3, 5, 5)
= (1, 0, 5, 2, 0, 4, 0, 2, 0, 0, 0, 3, 4,
. , . , . , 4, 2, 4, 4, 1, 3, 1)
= (1.0, 1.0, 1.0, 1.0, 1.0)

, . , . , 1.0,
-0.5, -1.0, -1.0,
. , . , . , . ,
. , . , . , . ,

Output:
IA
BX

= (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, . , . , . , . ,
. , . , . , 2, 1, 1, 3, 3, 5, 5)
= (-5.500000, 9.500000, 3.000000, -2.000000, -1.000000)

Note: On input, a “.” means that you do not have to store a value in that
position in the array. However, the storage position is required and may be
overwritten during the computation. On output, a “.” means that the value in
that position in the array is not significant.
Example 2
This example shows how to solve the system ATx = b, using the same matrix 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:

802

ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference


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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
BX = (1.0, 0.0, 0.0, 0.0, 0.0)

Note: On input, a “.” means that you do not have to store a value in that
position in the array. However, the storage position is required and may
be overwritten during the computation. On output, a “.” means that the value in
that position in the array is not significant.

Example 3
This example shows how to solve the system \( Ax = b \), using the same matrix \( A \)
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:
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, 0, 0, 0, 0, 0, 0, 0, 0)
BX = (0.000000, 3.000000, 0.000000, 0.000000, -1.000000)

Note: On input, a “.” means that you do not have to store a value in that
position in the array. However, the storage position is required and may
be overwritten during the computation. On output, a “.” means that the value in
that position in the array is not significant.

Example 4
This example shows how to solve the system \( A^T x = b \), using the same matrix \( A \)
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:
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, 0, 0, 0, 0, 0, 0, 0, 0)
BX = (0.000000, 0.000000, -3.000000, 1.000000, 0.000000, -3.000000)

Note: On input, a “.” means that you do not have to store a value in that
position in the array. However, the storage position is required and may
be overwritten during the computation. On output, a “.” means that the value in
that position in the array is not significant.
DGKFS (General Sparse Matrix or Its Transpose Factorization, Determinant, and Solve Using Skyline Storage Mode)

Purpose
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 Ax = b or Aᵀx = 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

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL DGKFS (n, au, nu, idu, al, nl, idl, iparm, rparm, aux, naux, bx, ldbx, mbx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>dgkfs (n, au, nu, idu, al, nl, idl, iparm, rparm, aux, naux, bx, ldbx, mbx);</td>
</tr>
</tbody>
</table>

On Entry

n  is the order of general sparse matrix A. Specified as: an integer; n ≥ 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: an integer; nu ≥ 0 and nu ≥ (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 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: an integer; **nl** ≥ 0 and **nl** ≥ (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 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 811.
    ```plaintext
    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

<table>
<thead>
<tr>
<th>Type of Computation</th>
<th>$Ax = b$</th>
<th>$Ax = b$ and Determinant(A)</th>
<th>$A^Tx = b$</th>
<th>$A^Tx = b$ and Determinant(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factor and Solve</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>110</td>
</tr>
<tr>
<td>Factor Only</td>
<td>1</td>
<td>11</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Solve Only</td>
<td>2</td>
<td>N/A</td>
<td>102</td>
<td>N/A</td>
</tr>
</tbody>
</table>

- **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 811.

- **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 811.
    - 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 $u_{ak}$ 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 811. 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 813.

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 integers, where:

\[
\begin{align*}
IPARM(1) & = 0 \text{ or } 1 \\
IPARM(2) & = 0, 1, 2, 10, 11, 100, 102, \text{ or } 110 \\
\text{If IPARM}(2) & = 0, 2, 10, 100, 102, \text{ or } 110, \text{ then IPARM}(3) = 0 \\
\text{If IPARM}(2) & = 1 \text{ or } 11, \text{ then } 0 \leq \text{IPARM}(3) \leq n \\
IPARM(4), IPARM(5) & = 0 \text{ or } 1 \\
\text{If IPARM}(2) & = 0, 1, 10, 11, 100, \text{ or } 110, \text{ then:} \\
IPARM(10) & = 0 \text{ or } 1 \\
IPARM(11), IPARM(12) & = -1, 0, \text{ or } 1 \\
IPARM(13) & = -1 \text{ or } 1 \\
IPARM(14), IPARM(15) & = -1, 0, \text{ or } 1
\end{align*}
\]

rparm

is an array of parameters, RPARN(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 \(10^{-15} \leq \text{IPARM}(10) \leq 1\).

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 \(|\text{RPARM}(10+i)|\) should be a sufficiently large nonzero value to avoid overflow when calculating the reciprocal of the pivot. The suggested value is \(10^{-15} \leq |\text{RPARM}(10+i)| \leq 1\).

If IPARM(10+i) ≠ 1, RPARM(10+i) is not used.

If you are doing a solve only, these arguments are not used.
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:

\[ \text{RPARM}(10) \geq 0.0 \]
\[ \text{RPARM}(11) \text{ through RPARM}(15) \neq 0.0 \]

\textit{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: an 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 \textit{factor only}:

\textbf{For 32-bit integer arguments}

Use \( naux \geq 5n \).

\textbf{For 64-bit integer arguments}

Use \( naux \geq 7n \).

If you are doing a \textit{factor and solve} or a \textit{solve only}:

\textbf{For 32-bit integer arguments}

Use \( naux \geq 5n + 4mbx \).

\textbf{For 64-bit integer arguments}

Use \( naux \geq 7n + 4mbx \).

\( bx \)

has the following meaning, where:

If you are doing a \textit{factor and solve} or a \textit{solve only}, \( bx \) is the array, containing the \( mbx \) right-hand side vectors \( b \) of the system \( Ax = b \) or \( A^Tx = b \). Each vector \( b \) is length \( n \) and is stored in the corresponding column of the array.

If you are doing a \textit{factor only}, this argument is not used in the computation.

Specified as: an \( ld bx \) by (at least) \( mbx \) array, containing long-precision real numbers.

\( ld bx \)

has the following meaning, where:

If you are doing a \textit{factor and solve} or a \textit{solve only}, \( ld bx \) is the leading dimension of the array specified for \( bx \).

If you are doing a \textit{factor only}, this argument is not used in the computation.

Specified as: an integer; \( ld bx \geq n \) and:

If \( mbx \neq 0 \), then \( ld bx > 0 \).

If \( mbx = 0 \), then \( ld bx \geq 0 \).
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: an integer; \( mbx \geq 0 \).

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 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 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 $IPARM(20+i) \geq 0$, 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 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 $1 \leq detbas < 10$. 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 \( Ax = b \) or \( A^T x = b \). 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 13. 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 13. Five Pivot Regions](image)

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 126 and "Diagonal-Out Skyline Storage Mode" on page 124.

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 factorized.

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 \( \text{naux} \) dynamically returned to your program. For details, see “Using Auxiliary Storage in ESSL” on page 51.

**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 \( \text{IPARM}(3) \) rows and columns are already factored in the input matrix. It factors the remaining \( n-\text{IPARM}(3) \) rows and columns.
in matrix $A$. (See “Notes” on page 811 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 $Ax = b$ or $A^T x = b$, 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 [11 on page 1364], [19 on page 1364], [32 on page 1365], [56 on page
1366], and [83 on page 1368]. If $n$ is 0, no computation is performed. If $m rx$ 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 68.

**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. \( \text{IPARM}(3) < 0 \)
15. \( \text{IPARM}(3) > n \)
16. \( \text{IPARM}(3) > 0 \) and \( \text{IPARM}(2) \neq 1 \) or \( 11 \)
17. \( \text{IPARM}(4), \text{IPARM}(5) \neq 0 \) or \( 1 \)
18. \( \text{IPARM}(2) = 0, 1, 10, 100, \text{or} 100 \) and:

\[ \text{IPARM}(10) \neq 0 \] or \( 1 \)
\[ \text{IPARM}(11), \text{IPARM}(12) - 1, \text{or} 1 \]
\[ \text{IPARM}(13) - 1 \text{ or} 1 \]
\[ \text{IPARM}(14), \text{IPARM}(15) - 1, \text{or} 1 \]
\[ \text{RPARM}(10) < 0.0 \]
\[ \text{RPARM}(10+i) = 0.0 \text{ and} \text{IPARM}(10+i) = 1 \text{ for } i = 1, 5 \]
19. \( \text{IPARM}(2) = 0, 2, 10, 100, 102, \text{or} 100 \) and:

\[ \text{ldbx} \leq 0 \text{ and} \text{mbx} \neq 0 \text{ and} n \neq 0 \]
\[ \text{ldbx} < 0 \text{ and} \text{mbx} = 0 \]
\[ \text{ldbx} < n \text{ and} \text{mbx} \neq 0 \]
\[ \text{mbx} < 0 \]
20. 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.

**Examples**

**Example 1**

This example shows how to factor a 9 by 9 general sparse matrix \( A \) and solve the system \( Ax = b \) with three right-hand sides. The default values are used for \( \text{IPARM} \) and \( \text{RPARM} \). Input matrix \( A \), shown here, is stored in diagonal-out skyline storage mode. Matrix \( A \) is:

\[
\begin{bmatrix}
2.0 & 2.0 & 2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
2.0 & 4.0 & 4.0 & 2.0 & 2.0 & 0.0 & 0.0 & 0.0 & 2.0 \\
2.0 & 4.0 & 6.0 & 4.0 & 4.0 & 0.0 & 2.0 & 0.0 & 4.0 \\
2.0 & 4.0 & 6.0 & 6.0 & 6.0 & 2.0 & 4.0 & 0.0 & 6.0 \\
0.0 & 0.0 & 0.0 & 2.0 & 4.0 & 4.0 & 4.0 & 2.0 & 4.0 \\
0.0 & 2.0 & 4.0 & 6.0 & 8.0 & 6.0 & 8.0 & 4.0 & 10.0 \\
0.0 & 0.0 & 0.0 & 2.0 & 4.0 & 6.0 & 8.0 & 6.0 & 8.0 \\
0.0 & 0.0 & 0.0 & 2.0 & 4.0 & 6.0 & 8.0 & 8.0 & 10.0 \\
2.0 & 4.0 & 6.0 & 6.0 & 8.0 & 6.0 & 10.0 & 8.0 & 16.0
\end{bmatrix}
\]

Output matrix \( A \), shown here, is in \( LU \) factored form with \( U^1 \) on the diagonal, and is stored in diagonal-out skyline storage mode. Matrix \( B \) is:

\[
\begin{bmatrix}
0.5 & 2.0 & 2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 0.5 & 2.0 & 2.0 & 2.0 & 0.0 & 0.0 & 0.0 & 2.0 \\
1.0 & 1.0 & 0.5 & 2.0 & 2.0 & 0.0 & 2.0 & 0.0 & 2.0 \\
1.0 & 1.0 & 1.0 & 0.5 & 2.0 & 2.0 & 2.0 & 0.0 & 2.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.5 & 2.0 & 2.0 & 2.0 \\
0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.5 & 2.0 & 2.0 & 2.0 \\
0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 1.0 & 0.5 & 2.0 & 2.0 \\
0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.5 & 2.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.5
\end{bmatrix}
\]

Call Statement and Input:
<table>
<thead>
<tr>
<th>N</th>
<th>AU</th>
<th>NU</th>
<th>IDU</th>
<th>AL</th>
<th>NL</th>
<th>IDL</th>
<th>IPARM</th>
<th>RPARAM</th>
<th>AUX</th>
<th>NAUX</th>
<th>BX</th>
<th>LDBX</th>
<th>MBX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>33</td>
<td>35</td>
<td>10</td>
<td>1  1</td>
<td></td>
<td>57</td>
<td>12</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALL DGKFS( 9, AU, 33, IDU, AL, 35, IDL, IPARM, RPARAM, AUX, 57, BX, 12, 3 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
AU & = (2.0, 4.0, 2.0, 6.0, 4.0, 2.0, 6.0, 4.0, 2.0, 4.0, 6.0, \\
    & \quad 4.0, 2.0, 6.0, 4.0, 2.0, 8.0, 8.0, 4.0, 4.0, 2.0, 8.0, \\
    & \quad 6.0, 4.0, 2.0, 16.0, 10.0, 8.0, 10.0, 4.0, 6.0, 4.0, 2.0) \\
IDU & = (1, 2, 4, 7, 10, 14, 17, 22, 26, 34) \\
AL & = (0.0, 0.0, 2.0, 0.0, 4.0, 2.0, 0.0, 6.0, 4.0, 2.0, 0.0, \\
    & \quad 2.0, 0.0, 8.0, 6.0, 4.0, 2.0, 0.0, 6.0, 4.0, 2.0, 0.0, \\
    & \quad 8.0, 6.0, 4.0, 2.0, 0.0, 8.0, 10.0, 6.0, 8.0, 6.0, 6.0, \\
    & \quad 4.0, 2.0) \\
IDL & = (1, 2, 4, 7, 11, 13, 18, 22, 27, 36) \\
IPARM & = (0, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \\
    & \quad \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots) \\
RPARAM & = (not relevant) \\
\begin{bmatrix}
6.00 & 12.00 & 18.00 \\
16.00 & 32.00 & 48.00 \\
26.00 & 52.00 & 78.00 \\
36.00 & 72.00 & 108.00 \\
20.00 & 40.00 & 60.00 \\
\end{bmatrix}
BX & = \\
\begin{bmatrix}
48.00 & 96.00 & 144.00 \\
34.00 & 68.00 & 102.00 \\
38.00 & 76.00 & 114.00 \\
66.00 & 132.00 & 198.00 \\
\end{bmatrix}
\]

Output:

\[
\begin{align*}
AU & = (0.5, 0.5, 2.0, 0.5, 2.0, 2.0, 0.5, 2.0, 0.5, 2.0, 0.5, \\
    & \quad 2.0, 2.0, 0.5, 2.0, 0.5, 2.0, 0.5, 2.0, 0.5, 2.0, 0.5, \\
    & \quad 2.0, 2.0, 0.5, 2.0, 0.5, 2.0, 0.5, 2.0, 0.5, 2.0, 0.5, \\
    & \quad 0.0, 2.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, \\
    & \quad 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, \\
    & \quad 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, \\
    & \quad 1.0, 1.0) \\
IDU & = (same as input) \\
AL & = (0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, \\
    & \quad 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, \\
    & \quad 1.0, 1.0) \\
IDL & = (same as input) \\
IPARM & = (0, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \\
    & \quad \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \\
    & \quad \ldots, 9, \ldots, \ldots, \ldots, 0, 0, 0, 0, 0, 9) \\
RPARAM & = (\ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \\
    & \quad \ldots, 8.0, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots) \\
\begin{bmatrix}
1.00 & 2.00 & 3.00 \\
1.00 & 2.00 & 3.00 \\
1.00 & 2.00 & 3.00 \\
1.00 & 2.00 & 3.00 \\
1.00 & 2.00 & 3.00 \\
\end{bmatrix}
BX & = \\
\begin{bmatrix}
1.00 & 2.00 & 3.00 \\
1.00 & 2.00 & 3.00 \\
1.00 & 2.00 & 3.00 \\
1.00 & 2.00 & 3.00 \\
\end{bmatrix}
\]
Example 2

This example shows how to factor the 9 by 9 general sparse matrix \( A \) from Example 1, solve the system \( A^T x = b \) with three right-hand sides, and compute the determinant of \( A \). The default values for pivot processing are used for IPARM. Input matrix \( A \) is stored in profile-in skyline storage mode. Output matrix \( A \) is in \( LU \) factored form with \( U^T \) on the diagonal, and is stored in diagonal-out skyline storage mode. It is the same as output matrix \( A \) in Example 1.

**Call Statement and Input:**

\[
\begin{array}{cccccccccccc}
N & AU & NU & IDU & AL & NL & IDL & IPARM & RPARM & AUX & NAUX & BX & LDBX & MBX \\
\hline
9 & 33 & 13 & 35 & 57 & 12 & 3 \\
\hline
\end{array}
\]

\[
\begin{align*}
AU &= (2.0, 2.0, 4.0, 0.0, 4.0, 0.0, 2.0, 0.0, 4.0, 0.0, 6.0, 0.0, 2.0, 0.0, 4.0, 0.0, 8.0, 8.0, 2.0, 4.0, 6.0, 8.0, 2.0, 4.0, 6.0, 4.0, 0.0, 0.0, 0.0, 8.0, 0.0, 2.0, 4.0, 6.0, 0.0, 2.0, 4.0, 6.0, 8.0, 0.0, 2.0, 4.0, 6.0, 8.0, 8.0, 8.0, 0.0, 8.0, 8.0, 10.0, 0.0, 8.0, 8.0, 16.0, 0.0) \\
IDU &= (1, 3, 6, 9, 13, 16, 21, 25, 33, 34) \\
AL &= (0.0, 2.0, 0.0, 2.0, 4.0, 0.0, 2.0, 4.0, 0.0, 2.0, 4.0, 6.0, 0.0, 2.0, 4.0, 6.0, 0.0, 2.0, 4.0, 6.0, 8.0, 0.0, 4.0, 6.0, 8.0, 8.0, 0.0, 16.0, 0.0, 0.0) \\
IDL &= (1, 3, 6, 10, 12, 17, 21, 26, 35, 36) \\
IPARM &= (1, 110, 0, 1, 0, \ldots, 0, \ldots, 0, \ldots, 0, \ldots, 0, \ldots, 0, \ldots, 0, 9, \ldots, 0, 0, 0, 0, 0) \\
RPARM &= (\text{not relevant}) \\
\end{align*}
\]

\[
\begin{bmatrix}
10.00 & 20.00 & 30.00 \\
20.00 & 40.00 & 60.00 \\
28.00 & 56.00 & 84.00 \\
30.00 & 60.00 & 90.00 \\
40.00 & 80.00 & 120.00 \\
30.00 & 60.00 & 90.00 \\
44.00 & 88.00 & 132.00 \\
28.00 & 56.00 & 84.00 \\
60.00 & 120.00 & 180.00 \\
\vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots \\
\end{bmatrix}
\]

**Output:**

\[
\begin{align*}
AU &= \text{(same as output AU in Example 1)} \\
IDU &= \text{(same as output IDU in Example 1)} \\
AL &= \text{(same as output AL in Example 1)} \\
IDL &= \text{(same as output IDL in Example 1)} \\
IPARM &= (1, 110, 0, 1, 0, \ldots, 0, \ldots, 0, \ldots, 0, \ldots, 0, \ldots, 0, \ldots, 0, 9, \ldots, 0, 0, 0, 0, 0) \\
RPARM &= (\ldots, 0, \ldots, 0, \ldots, 0, \ldots, 0, \ldots, 0, \ldots, 0, \ldots, 0, \ldots, 0, \ldots, 0, \ldots, 0, \ldots, 0, \ldots, 0, \ldots, 0) \\
BX &= \text{(same as output BX in Example 1)}
\end{align*}
\]

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:
CALL DGKFS(AU, IDU, AL, RPARM, BX, IPARM, AUX, NAUX, BX, LDBX, MBX)

Call Statement and Input:

\[
\begin{bmatrix}
-2.0 & -2.0 & -2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
-2.0 & -4.0 & -4.0 & -2.0 & -2.0 & 0.0 & 0.0 & 0.0 & -2.0 \\
-2.0 & -4.0 & -6.0 & -4.0 & -4.0 & 0.0 & -2.0 & 0.0 & -4.0 \\
-2.0 & -4.0 & -6.0 & -6.0 & -6.0 & -2.0 & -4.0 & 0.0 & -6.0 \\
0.0 & 0.0 & 0.0 & -2.0 & -4.0 & -4.0 & -4.0 & -2.0 & -4.0 \\
0.0 & -2.0 & -4.0 & -6.0 & -8.0 & -6.0 & -8.0 & -4.0 & -10.0 \\
0.0 & 0.0 & 0.0 & -2.0 & -4.0 & -6.0 & -8.0 & -6.0 & -8.0 \\
0.0 & 0.0 & 0.0 & -2.0 & -4.0 & -6.0 & -8.0 & -8.0 & -8.0 \\
-2.0 & -4.0 & -6.0 & -6.0 & -8.0 & -6.0 & -10.0 & -8.0 & -16.0
\end{bmatrix}
\]

Output matrix \(A\), shown here, is in \(LU\) factored form with \(U^1\) on the diagonal, and is stored in diagonal-out skyline storage mode. Matrix \(A\) is:

\[
\begin{bmatrix}
-0.5 & -2.0 & -2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & -0.5 & -2.0 & -2.0 & 0.0 & 0.0 & 0.0 & -2.0 \\
1.0 & 1.0 & -0.5 & -2.0 & -2.0 & 0.0 & 0.0 & -2.0 \\
1.0 & 1.0 & 1.0 & -0.5 & -2.0 & -2.0 & 0.0 & -2.0 \\
0.0 & 0.0 & 0.0 & 1.0 & -0.5 & -2.0 & -2.0 & -2.0 \\
0.0 & 1.0 & 1.0 & 1.0 & -0.5 & -2.0 & -2.0 & -2.0 \\
0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 1.0 & -0.5 & -2.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & -0.5
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
-0.5 & -2.0 & -2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & -2.0 & -4.0 & -8.0 & -10.0 & -12.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0
\end{bmatrix}
\]
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:

\[
\begin{pmatrix}
2.0 & 2.0 & 2.0 & 0.0 & 0.0 & 0.0 \\
2.0 & 4.0 & 4.0 & 2.0 & 2.0 & 0.0 \\
2.0 & 4.0 & 6.0 & 4.0 & 4.0 & 0.0 \\
2.0 & 4.0 & 6.0 & 6.0 & 6.0 & 2.0 \\
0.0 & 0.0 & 0.0 & 2.0 & 4.0 & 4.0 \\
0.0 & 2.0 & 4.0 & 6.0 & 8.0 & 6.0 \\
\end{pmatrix}
\]

Output matrix $A1$, shown here, is in $LU$ factored form with $U^T$ on the diagonal, and is stored in diagonal-out skyline storage mode. Output matrix $A1$ is:

\[
\begin{pmatrix}
0.5 & 2.0 & 2.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 0.5 & 2.0 & 2.0 & 2.0 & 0.0 \\
1.0 & 1.0 & 0.5 & 2.0 & 2.0 & 0.0 \\
1.0 & 1.0 & 1.0 & 0.5 & 2.0 & 2.0 \\
0.0 & 0.0 & 0.0 & 1.0 & 0.5 & 2.0 \\
0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.5 \\
\end{pmatrix}
\]

Call Statement and Input:

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
N & AU & NU & IDU & AL & NL & IDL & IPARM & RPARM & AUX & NAUX & BX & LDBX & MBX \\
\hline
6 & 33 & 35 & 36 & 35 & 36 & 35 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
\end{array}
\]

CALL DGKFS(6, AU, 33, IDU, AL, 35, IDL, IPARM, RPARM, AUX, NAUX, BX, LDBX, MBX)

Output:

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
AU & = & \text{(same as input AU in Example 1)} & & & & & & & & & & & & \\
IDU & = & (1, 2, 4, 7, 10, 14, 17) & & & & & & & & & & & & \\
AL & = & \text{(same as input AL in Example 1)} & & & & & & & & & & & & \\
IDL & = & (1, 2, 4, 7, 11, 13, 18) & & & & & & & & & & & & \\
IPARM & = & (1, 11, 0, 0, 0, \ldots, \ldots, \ldots, 0, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots) & & & & & & & & & & & & \\
RPARM & = & \text{(not relevant)} & & & & & & & & & & & & \\
BX & = & \text{(not relevant)} & & & & & & & & & & & & \\
LDBX & = & \text{(not relevant)} & & & & & & & & & & & & \\
MBX & = & \text{(not relevant)} & & & & & & & & & & & & \\
\hline
\end{array}
\]
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 $A_2$, shown here, is made up of the output factored matrix $A_1$ plus the three remaining unfactored rows and columns of matrix $A$. Matrix $A_2$ is:

\[
\begin{bmatrix}
0.5 & 2.0 & 2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 0.5 & 2.0 & 2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 2.0 \\
1.0 & 1.0 & 0.5 & 2.0 & 2.0 & 0.0 & 2.0 & 0.0 & 4.0 \\
1.0 & 1.0 & 1.0 & 0.5 & 2.0 & 2.0 & 4.0 & 0.0 & 6.0 \\
0.0 & 0.0 & 0.0 & 1.0 & 0.5 & 2.0 & 4.0 & 2.0 & 4.0 \\
0.0 & 1.0 & 1.0 & 1.0 & 0.5 & 8.0 & 4.0 & 10.0 & \\
0.0 & 0.0 & 0.0 & 2.0 & 4.0 & 6.0 & 8.0 & 6.0 & 8.0 \\
0.0 & 0.0 & 0.0 & 2.0 & 4.0 & 6.0 & 8.0 & 8.0 & 10.0 \\
2.0 & 4.0 & 6.0 & 6.0 & 8.0 & 6.0 & 10.0 & 8.0 & 16.0
\end{bmatrix}
\]

Both parts of input matrix $A_2$ are stored in diagonal-out skyline storage mode.

Output matrix $A_2$ is the same as output matrix $A$ in Example 1 and is stored in diagonal-out skyline storage mode.

Call Statement and Input:

\[
\text{CALL DGKFS}( \text{N}, \text{AU}, \text{NU}, \text{IDU}, \text{AL}, \text{NL}, \text{IDL}, \text{IPARM}, \text{RPARM}, \text{AUX}, \text{NAUX}, \text{BX}, \text{LDBX}, \text{MBX})
\]

\[
\begin{align*}
\text{AU} &= \text{(same as output AU in Example 4)} \\
\text{IDU} &= \text{(same as input IDU in Example 1)} \\
\text{AL} &= \text{(same as output AL in Example 4)} \\
\text{IDL} &= \text{(same as input IDL in Example 1)} \\
\text{IPARM} &= (1,1,6,0,0,\ldots,0,\ldots,0,\ldots,0,\ldots,0) \\
\text{RPARM} &= (\text{not relevant}) \\
\end{align*}
\]

Output:

\[
\begin{align*}
\text{AU} &= \text{(same as output AU in Example 1)} \\
\text{IDU} &= \text{(same as output IDU in Example 1)} \\
\text{AL} &= \text{(same as output AL in Example 1)} \\
\text{IDL} &= \text{(same as input IDL in Example 1)} \\
\text{IPARM} &= (1,1,6,0,0,\ldots,0,\ldots,0,0,0,0,3) \\
\text{RPARM} &= (\text{not relevant}) \\
\text{BX} &= \text{(same as input)} \\
\text{LDBX} &= \text{(same as input)} \\
\text{MBX} &= \text{(same as input)}
\end{align*}
\]

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:

\[
\text{CALL DGKFS}( \text{N}, \text{AU}, \text{NU}, \text{IDU}, \text{AL}, \text{NL}, \text{IDL}, \text{IPARM}, \text{RPARM}, \text{AUX}, \text{NAUX}, \text{BX}, \text{LDBX}, \text{MBX})
\]

\[
\text{CALL DGKFS}( 9, \text{AU}, 33, \text{IDU}, \text{AL}, 35, \text{IDL}, \text{IPARM}, \text{RPARM}, \text{AUX}, 49, \text{BX}, 9, 1 )
\]
AU = (0.5, 2.0, 0.5, 2.0, 0.5, 2.0, 0.5, 2.0, 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, 2.0, 0.5, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0)
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, 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, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0)
IDL = (1, 3, 6, 10, 12, 17, 21, 26, 35, 36)
IPARM = (1, 2, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 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)

**Purpose**

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**

<table>
<thead>
<tr>
<th>Language</th>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL DSKFS (n, a, na, idiag, iparm, rparm, aux, naux, bx, ldbx, mbx)</td>
<td></td>
</tr>
<tr>
<td>C and C++</td>
<td>dskfs (n, a, na, idiag, iparm, rparm, aux, naux, bx, ldbx, mbx);</td>
<td></td>
</tr>
</tbody>
</table>

**On Entry**

- \( n \) is the order of symmetric sparse matrix \( A \). Specified as: an 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: an 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 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 827.

  \[
  \begin{align*}
  IPARM(2) &= 0 \\
  IPARM(3) &= 0 \\
  IPARM(4) &= 0 \\
  IPARM(5) &= 0 \\
  IPARM(10) &= 0 \\
  IPARM(11) &= -1 \\
  IPARM(12) &= -1 \\
  IPARM(13) &= -1 \\
  IPARM(14) &= -1 \\
  IPARM(15) &= 0 \\
  RPARM(10) &= 10^{-12}
  \end{align*}
  \]

  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:

<table>
<thead>
<tr>
<th>Type of Computation</th>
<th>Gaussian Elimination $Ax = b$</th>
<th>Gaussian Elimination $Ax = b$ and Determinant(A)</th>
<th>Cholesky Decomposition $Ax = b$</th>
<th>Cholesky Decomposition $Ax = b$ and Determinant(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factor and Solve</td>
<td>0</td>
<td>10</td>
<td>100</td>
<td>110</td>
</tr>
<tr>
<td>Factor Only</td>
<td>1</td>
<td>11</td>
<td>101</td>
<td>111</td>
</tr>
<tr>
<td>Solve Only</td>
<td>2</td>
<td>N/A</td>
<td>102</td>
<td>N/A</td>
</tr>
</tbody>
</table>

- 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 827.

- 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 827.

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 827. 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 830.
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 integers, where:

IPARM(1) = 0 or 1
IPARM(2) = 0, 1, 2, 10, 11, 100, 101, 102, 110, or 111
If IPARM(2) = 0, 10, 100, 102, or 110, then IPARM(3) = 0
If IPARM(2) = 1, 11, 101, or 111, then 0 \( \leq \) IPARM(3) \( \leq \) 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
$10^{-15} \leq \text{IPARM(10)} \leq 1$.

  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
    $|\text{RPARM}(10+i)|$ should be a sufficiently large nonzero value to avoid overflow
    when calculating the reciprocal of the pivot. For Gaussian elimination, the
    suggested value is $10^{-15} \leq |\text{RPARM}(10+i)| \leq 1$. For Cholesky decomposition,
    the value must be $\text{RPARM}(10+i) > 0$.

    If IPARM(10+i) ≠ 1, RPARN(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: an 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**

**For 32-bit integer arguments**
- You can use \( naux \geq n \).

**For 64-bit integer arguments**
- You can use \( naux \geq 2n \).

However, for optimal performance:

**For 32-bit integer arguments**
- Use \( naux \geq 3n \).

**For 64-bit integer arguments**
- Use \( naux \geq 4n \).

If you are doing a **factor and solve** or a **solve only**:

**For 32-bit integer arguments**
- Use \( naux \geq 3n + 4mbx \).

**For 64-bit integer arguments**
- Use \( naux \geq 4n + 4mbx \).

For further details on error handling and the special factor-only case, see "Notes" on page 827.

\( bx \) 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: an integer; \( ldbx \geq n \) and:

- If \( mbx \neq 0 \), then \( ldbx > 0 \).
- If \( mbx = 0 \), then \( ldbx \geq 0 \).

\( mbx \) 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: an integer; \( mbx \geq 0 \).
\(a\) is the array, referred to as \(A\), containing the upper triangular part of symmetric sparse matrix \(A\) in \(LDL^T\) or \(R^TR\) 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 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) \geq 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 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 $k$, $a_{ik}/d_{kk}$ for Gaussian elimination and $a_{ik}/r_{ik}$ 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: $detbas \times 10^{detpwr}$, where $1 \leq detbas < 10$. 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 $Ax = b$. 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. 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 \textit{iparm} and \textit{rparm} are defined for the five pivot regions handled by this subroutine. The limits of the regions are based on \texttt{RPARM(10)}, as shown in \textbf{Figure 14}. The pivot values in each region are:

- **Region 1:** pivot < -\texttt{RPARM(10)}
- **Region 2:** -\texttt{RPARM(10)} ≤ pivot < 0
- **Region 3:** pivot = 0
- **Region 4:** 0 < pivot ≤ \texttt{RPARM(10)}
- **Region 5:** pivot > \texttt{RPARM(10)}

\textbf{Figure 14. Five Pivot Regions}

4. The \texttt{IPARM(4)} and \texttt{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 126 and "Diagonal-Out Skyline Storage Mode" on page 124. Those descriptions use different array and variable names from the ones used here. To relate the two sets, use the following table:

<table>
<thead>
<tr>
<th>Name Here</th>
<th>Name in the Storage Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{A}</td>
<td>\texttt{AU}</td>
</tr>
<tr>
<td>\texttt{nu}</td>
<td>\texttt{nu}</td>
</tr>
<tr>
<td>\texttt{IDIAG}</td>
<td>\texttt{IDU}</td>
</tr>
</tbody>
</table>

5. Following is an illustration of the portion of matrix \( A \) factored in the partial factorization when \texttt{IPARM(3)} > 0. In this case, the subroutine assumes that rows and columns 1 through \texttt{IPARM(3)} are already factored and that rows and columns \texttt{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 $\text{naux}$ dynamically returned to your program. For details, see “Using Auxiliary Storage in ESSL” on page 51.

**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:

$$A = LDL^T$$

where:

- $D$ is a diagonal matrix.
- $L$ is a lower triangular matrix.

The transformed matrix $A$, factored into its $LDL^T$ 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 $L^T$ 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:

$$A = R^TR$$

where $R$ is an upper triangular matrix.

The transformed matrix $A$, factored into its $R^TR$ 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 $\text{IPARM}(3)$ rows and columns are already factored in the input matrix. It factors the remaining $n-\text{IPARM}(3)$ rows and columns in matrix $A$. (See “Notes” on page 827 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 \( Ax = b \), 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 [11 on page 1364], [19 on page 1364], [32 on page 1365], [56 on page 1366], [83 on page 1368]. 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 \( 1 \) and \( 1 \)\( \text{IPARM}(10+i) = 1 \), the pivot value is replaced with \( \text{RPARM}(10+i) \), 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 \( \text{ERRSET} \) 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 68.

**Input-Argument Errors**
1. \( n < 0 \)
2. \( na < 0 \)
3. \( \text{IDIAG}(n+1) > na+1 \)
4. \( \text{IDIAG}(i+1) = \text{IDIAG}(i) \) for \( i = 1, n \)
5. \( \text{IDIAG}(i+1) > \text{IDIAG}(i)+i \) and \( \text{IPARM}(4) = 0 \) for \( i = 1, n \)
6. \( \text{IDIAG}(i) > \text{IDIAG}(i-1)+i \) and \( \text{IPARM}(4) = 1 \) for \( i = 2, n \)
7. \( \text{IPARM}(1) \neq 0 \) or \( 1 \)
8. \( \text{IPARM}(2) \neq 0, 1, 2, 10, 11, 100, 101, 102, 110, \) or \( 111 \)
9. \( \text{IPARM}(3) < 0 \)
10. \( \text{IPARM}(3) > n \)
11. \( \text{IPARM}(3) > 0 \) and \( \text{IPARM}(2) \neq 1, 11, 101, \) or \( 111 \)
12. \( \text{IPARM}(4), \text{IPARM}(5) \neq 0 \) or \( 1 \)
13. \( \text{IPARM}(2) = 0, 1, 10, \) or \( 11 \) and:
   - \( \text{IPARM}(10) \neq 0 \) or \( 1 \)
   - \( \text{IPARM}(11), \text{IPARM}(12) \neq -1, 0, \) 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.

Examples

Example 1

This example shows how to factor a 9 by 9 symmetric sparse matrix \(A\) and solve the system \(Ax = b\) with three right-hand sides. It uses Gaussian elimination. The default values are used for IPARM and RPARM. Input matrix \(A\), shown here, is stored in diagonal-out skyline storage mode. Matrix \(A\) is:

\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 2.0 & 2.0 & 2.0 & 1.0 & 1.0 & 0.0 & 1.0 & 0.0 \\
1.0 & 2.0 & 3.0 & 3.0 & 2.0 & 2.0 & 0.0 & 2.0 & 0.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 3.0 & 3.0 & 0.0 & 3.0 & 0.0 \\
0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 4.0 & 1.0 & 4.0 & 0.0 \\
0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 2.0 & 5.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & 3.0 & 3.0 & 2.0 \\
0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 3.0 & 7.0 & 3.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 0.0
\end{bmatrix}
\]

Output matrix \(A\), shown here, is in LDL\(^T\) factored form with \(D\) on the diagonal, and is stored in diagonal-out skyline storage mode. Matrix \(A\) is:

\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 \\
\end{bmatrix}
\]

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 )
```
Example 2

This example shows how to factor the 9 by 9 symmetric sparse matrix $A$ from Example 1, solve the system $Ax = b$ with three right-hand sides, and compute the determinant of $A$. It uses Gaussian elimination. The default values for pivot processing are used for $IPARM$. Input matrix $A$ is stored in profile-in skyline storage mode. Output matrix $A$ is in $LDL^T$ factored form with $D^{-1}$ on the diagonal, and is stored in diagonal-out skyline storage mode. It is the same as output matrix $A$ in Example 1.

Call Statement and Input:

```fortran
CALL DSKFS( 9, A, 33, IDIAG, IPARM, RPARAM, AUX, NAUX, BX, LDBX, MBX )
```

### Example 1

Input matrix $A$ is stored in diagonal-out skyline storage mode. It is the same as $A$ used in Example 1.

```plaintext
A = (1.0, 2.0, 1.0, 3.0, 2.0, 1.0, ..., 1.0, 1.0, 4.0, 3.0, 2.0, 1.0, 4.0, 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, ..., 10.00, 20.00, 30.00)

RPARAM = (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)

IDIAG = (same as input)

IPARM = (0, ..., 7.0, ..., 10.00, 20.00, 30.00)

RPARAM = (8, 10.00, 20.00, 30.00)

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
1.00 2.00 3.00
```

Example 2
Example 3

This example shows how to factor a 9 by 9 negative-definite symmetric sparse matrix \( A \), solve the system \( Ax = b \) with three right-hand sides, and compute the determinant of \( A \). It uses Gaussian elimination. (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. Matrix \( A \) is:

\[
\begin{bmatrix}
-1.0 & -1.0 & -1.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
-1.0 & -2.0 & -2.0 & -2.0 & -1.0 & 0.0 & -1.0 & 0.0 & 0.0 \\
-1.0 & -2.0 & -3.0 & -3.0 & -2.0 & -2.0 & 0.0 & -2.0 & 0.0 \\
-1.0 & -2.0 & -3.0 & -4.0 & -3.0 & -3.0 & 0.0 & -3.0 & 0.0 \\
0.0 & -1.0 & -2.0 & -3.0 & -4.0 & -4.0 & -1.0 & -4.0 & 0.0 \\
0.0 & -1.0 & -2.0 & -3.0 & -4.0 & -5.0 & -2.0 & -5.0 & -1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & -1.0 & -2.0 & -3.0 & -3.0 & -2.0 \\
0.0 & -1.0 & -2.0 & -3.0 & -4.0 & -5.0 & -3.0 & -7.0 & -3.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -1.0 & -2.0 & -3.0 & -4.0 \\
\end{bmatrix}
\]

Output matrix \( A \), shown here, is in \( LDL^T \) factored form with \( D \) on the diagonal, and is stored in diagonal-out skyline storage mode. Matrix \( A \) is:

\[
\begin{bmatrix}
-1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & -1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 1.0 & 0.0 \\
1.0 & 1.0 & -1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 1.0 & 0.0 \\
1.0 & 1.0 & 1.0 & -1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 1.0 & 1.0 & 1.0 & -1.0 & 1.0 & 1.0 & 1.0 & 0.0 \\
0.0 & 1.0 & 1.0 & 1.0 & -1.0 & 1.0 & 1.0 & 1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & -1.0 & 1.0 & 1.0 \\
\end{bmatrix}
\]
Example 4

This example shows how to factor the first six rows and columns, referred to as matrix $A_1$, of the 9 by 9 symmetric sparse matrix $A$ from Example 1 and compute the determinant of $A_1$. It uses Gaussian elimination. Input matrix $A_1$, shown here, is stored in diagonal-out skyline storage mode. Input matrix $A_1$ is:

$$
\begin{bmatrix}
0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & -1.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & -1.0 & 1.0
\end{bmatrix}
$$

Call Statement and Input:

```plaintext
CALL DSKFS(9, A, 33, IDIAG, IPARM, RPARM, AUX, NAUX, BX, LDBX, MBX,
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, . . .
            . . ., . . ., . . .)

            . . ., . . ., . . ., . . .)

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)

IDIAG = (same as input)

IPARM = (1, 10, 0, 0, 0, . . ., 1, 0, -1, -1, -1, -1, 8,
            . . ., . . ., 9, 0, 0, 0)

RPARM = ( . . ., . . ., 10^{15}, . . .
            . . ., 7.0, -1.0, 0.0, . . ., . . .)

BX =

$$
\begin{bmatrix}
-1.00 & -2.00 & -3.00 \\
-1.00 & -2.00 & -3.00 \\
-1.00 & -2.00 & -3.00 \\
-1.00 & -2.00 & -3.00 \\
-1.00 & -2.00 & -3.00 \\
\end{bmatrix}
$$
Output matrix $A_1$, shown here, is in $LDL^T$ factored form with $D^1$ on the
diagonal, and is stored in diagonal-out skyline storage mode. Output matrix
$A_1$ is:

$$
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
\end{bmatrix}
$$

Call Statement and Input:

```
CALL DSKFS (6, A, 33, IDIAG, IPARM, RPARAM, AUX, 27, BX, LDBX, MBX)
```

- $A$ = (same as input $A$ in Example 1)
- IDIAG = $(1, 2, 4, 7, 11, 15, 20)$
- IPARM = $(1, 11, 0, 0, 0, \ldots, 0, \ldots, \ldots, \ldots)$
- RPARAM = (not relevant)
- BX = (not relevant)
- LDBX = (not relevant)
- MBX = (not relevant)

Output:

- $A$ = $(1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,$
  $1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,$
  $0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,$
  $0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)$
- IDIAG = (same as input)
- IPARM = $(1, 11, 0, 0, 0, \ldots, 0, \ldots, \ldots, \ldots, 6,$
  $\ldots, \ldots, 0, 0, 0, 6)$
- RPARAM = $(5.0, 1.0, 0.0, 0.0, \ldots, 0.0, 0.0, 0.0, 0.0, 0.0)$
- 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 $A_2$, shown here, is made up of the
output factored matrix $A_1$ plus the three remaining unfactored rows and
columns of matrix $A$. Matrix $A_2$ is:

$$
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 1.0 & 0.0 & 0.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 2.0 & 0.0 & 0.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 3.0 & 0.0 & 0.0 \\
0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 4.0 & 0.0 & 0.0 \\
0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 2.0 & 5.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & 3.0 & 3.0 & 2.0 \\
0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 3.0 & 7.0 & 3.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 0.0 \\
\end{bmatrix}
$$
Both parts of input matrix $A_2$ are stored in diagonal-out skyline storage mode.

Output matrix $A_2$ is the same as output matrix $A$ in Example 1 and is stored in diagonal-out skyline storage mode.

Example 6

This example shows how to solve the system $Ax = b$ with one right-hand side for a symmetric 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. It specifies Gaussian elimination, as used in Example 1. Here, output matrix $A$ 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, 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, ..., 0, ..., 8,
          ..., 0, 0, 0, 3)
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, ..., 0, ..., 8,
          ..., 0, 0, 0, 3)
RPARM = (not relevant)
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 symmetric 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. It specifies Gaussian elimination, as used in Example 1. Here, output matrix $A$ 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)
IDIAG = (1, 3, 6, 10, 14, 19, 22, 29, 33, 34)
IPARM = (1, 2, 0, 1, 1, ..., 0, ..., 0, ..., 0,
          ..., 0, ..., 0, 3)
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)
```
Example 7

This example shows how to factor a 9 by 9 symmetric sparse matrix $A$ and solve the system $Ax = b$ with four right-hand sides. It uses Cholesky decomposition. Input matrix $A$, shown here, is stored in profile-in skyline storage mode Matrix $A$ 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 $R^tR$ 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:

$$
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 1.0 \\
1.0 & .5 & 1.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 1.0 \\
1.0 & 1.0 & .333 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 1.0 \\
0.0 & 0.0 & 1.0 & .25 & 1.0 & 1.0 & 1.0 & 0.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & .2 & 1.0 & 1.0 & 0.0 & 1.0 \\
0.0 & 0.0 & 1.0 & 1.0 & 1.0 & .167 & 1.0 & 1.0 & 1.0 \\
0.0 & 0.0 & 1.0 & 1.0 & 1.0 & .143 & 1.0 & 1.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & .125 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & .111 & \, .
\end{bmatrix}
$$

Call Statement and Input:

```
CALL DSKFS( 9, A, 34, IDIAG, IPARM, RPARM, AUX, NAUX, BX, LDBX, MBX \\
            N A NA IDIAG IPARM RPARM AUX NAUX BX LDBX MBX             
            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:
\[ A = (1.0, 1.0, .5, 1.0, 1.0, .333, 1.0, .25, 1.0, 1.0, 1.0, .2, 1.0, 1.0, 1.0, .167, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, .125, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, .143, 1.0, 1.0, .125, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, .111) \]

\[ \text{IDIAG} = \text{(same as input)} \]

\[ \text{IPARM} = (1, 110, 0, 1, 1, \ldots, \ldots, 0, \ldots, \ldots, \ldots, 0, 0, 0, 0, 9) \]

\[ \text{RPARM} = (\ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, 0, 0, 0, 0, 9.89, 1.32, 11.0, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots) \]

\[
\begin{bmatrix}
1.00 & 2.00 & 3.00 & 4.00 \\
1.00 & 2.00 & 3.00 & 4.00 \\
1.00 & 2.00 & 3.00 & 4.00 \\
1.00 & 2.00 & 3.00 & 4.00 \\
1.00 & 2.00 & 3.00 & 4.00 \\
1.00 & 2.00 & 3.00 & 4.00 \\
1.00 & 2.00 & 3.00 & 4.00 \\
\ldots & \ldots & \ldots & \ldots
\end{bmatrix}
\]

\[ \text{BX} = \]
DSRIS (Iterative Linear System Solver for a General or Symmetric Sparse Matrix Stored by Rows)

Purpose

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:

$$Ax = b$$

where $A$, $x$, and $b$ contain long-precision real numbers.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL DSRIS (stor, init, n, ar, ja, ia, b, x, iparm, rparm, aux1, aux2, naux1, naux2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>dsris (stor, init, n, ar, ja, ia, b, x, iparm, rparm, aux1, aux2, naux1, naux2);</td>
</tr>
</tbody>
</table>

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 $init = 'T'$, 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 $init = 'T'$. 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 'T' or 'S'.
\( n \) is the order of the linear system \( Ax = b \) and the number of rows and columns in sparse matrix \( A \).

Specified as: an integer; \( n \geq 0 \).

\( ar \) is the sparse matrix \( A \) of order \( n \), stored by rows in an array, referred to as \( AR \). The \textit{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 integers; \( 1 \leq (JA \text{ elements}) \leq n \). 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 integers; \( IA(i+1) \geq IA(i) \) 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, \textit{IPARM}(i), where:

- \textit{IPARM}(1) controls the number of iterations.
  - If \textit{IPARM}(1) > 0, \textit{IPARM}(1) is the maximum number of iterations allowed.
  - If \textit{IPARM}(1) = 0, the following default values are used:
    
    \[
    \begin{align*}
    \text{IPARM}(1) &= 300 \\
    \text{IPARM}(2) &= 4 \\
    \text{IPARM}(4) &= 4 \\
    \text{IPARM}(5) &= 1 \\
    \text{RPARM}(1) &= 10^{-6} \\
    \text{RPARM}(2) &= 1
    \end{align*}
    \]
  - \textit{IPARM}(2) is the flag used to select the iterative procedure used in this subroutine.
    - If \textit{IPARM}(2) = 1, the conjugate gradient (CG) method is used. Note that this algorithm should only be used with positive definite symmetric matrices.
    - If \textit{IPARM}(2) = 2, the conjugate gradient squared (CGS) method is used.
    - If \textit{IPARM}(2) = 3, the generalized minimum residual (GMRES) method, restarted after \( k \) steps, is used.
    - If \textit{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) ≠ 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).
  - xj is the solution found at the j-th iteration.
  - rj and r0 are the preconditioned residuals obtained at iterations j and 0, respectively. (The residual at iteration j is given by b-Axj.)

  If IPARM(5) = 1, the iterative method is stopped when:

  \[ \frac{\| r_j \|_2}{\| x_j \|_2} < \epsilon \]

  **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:

  \[ \frac{\| r_j \|_2}{\| r_0 \|_2} < \epsilon \]

  If IPARM(5) = 3, the iterative method is stopped when:

  \[ \frac{\| x_j - x_{j-1} \|_2}{\| x_j \|_2} < \epsilon \]

  **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 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) has the following meaning, where:
- if RPARM(1) > 0, then RPARM(1) is the relative accuracy ε used in the stopping criterion.
- if RPARM(1) = 0, then the solver is forced to evaluate at most IPARM(1) iterations.

See [5 on page 844](#)

RPARM(2), see On Return

RPARM(3) has the following meaning, where:
- If IPARM(4) ≠ 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) ≥ 0
If IPARM(4) = 3, RPARM(3) > 0

**aux1**

is working storage for this subroutine, where:

If *init* = ‘T’, 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: an integer, where:

In these formulas *nw* has the following value:

If *stor* = ‘G’, then *nw* = 1A(n+1)-1+n.
If *stor* = ‘U’ or ‘L’, then *nw* = 2(1A(n+1)-1).

**For 32-bit integer arguments**

If IPARM(4) = 1, use *naux1* ≥ (3/2)*nw + (1/2)*n + 20.
If IPARM(4) = 2, use *naux1* ≥ (3/2)*nw + (3/2)*n + 20.
If IPARM(4) = 3, 4, or 5, use *naux1* ≥ 3*nw + n + 20.

**For 64-bit integer arguments**

If IPARM(4) = 1, use *naux1* ≥ 2*nw + n + 40.
If IPARM(4) = 2, use *naux1* ≥ 2*nw + 2n + 40.
If IPARM(4) = 3, 4, or 5, use *naux1* ≥ 4*nw + 4n + 40.
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: an 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 naux2 ≥ 4n.
If IPARM(2) = 2, use naux2 ≥ 7n.
If IPARM(2) = 3, use naux2 ≥ (k+2)n+k(k+4)+1, where k = IPARM(3).
If IPARM(2) = 4, use naux2 ≥ 7n.
If IPARM(2) = 5, use naux2 ≥ 9n.

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 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 integers.

rparm

is an array of parameters, RPARM(i), where:
RPARAM(1) is unchanged.
RPARAM(2) contains the estimate of the error of the solution. If the process converged, RPARAM(2) ≤ ε.
RPARAM(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 **aux1** 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](#)) 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 A, vector x, and vector b must have no common elements; otherwise, results are unpredictable.

5. The algorithm computes a sequence of approximate solution vectors 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 RPARAM(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 122.

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 51.
The linear system:

\[ Ax = b \]

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 \( AR, IA, \text{ and } JA \). 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 [44 on page 1366], [67 on page 1367], [99 on page 1369], [105 on page 1369], [108 on page 1369], and [114 on page 1370].

When you call this subroutine to solve a system for the first time, you specify \( \text{init} = 'I' \). After that, you can solve the same system any number of times by calling this subroutine each time with \( \text{init} = 'S' \). 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 68.

- For error 2110, if \( RPARM(1) > \), 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 \( \text{init} = 'S' \), but the data contained in \( \text{aux1} \) was computed for a different algorithm. An attention message is issued. Processing continues, and the contents of \( \text{aux1} \) are reset correctly.

• For error 2134, return code 3 indicates that the data contained in \( \text{aux1} \) is not consistent with the input sparse matrix. The subroutine has been called with \( \text{init} = 'S' \), and \( \text{aux1} \) contains an incomplete factorization and internal data storage for the input matrix \( A \) that was computed by a previous call to the subroutine when \( \text{init} = 'I' \). This error indicates that \( \text{aux1} \) 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 \( \text{IPARM}(4) = 1 \) 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. \( \text{stor} \neq 'G', 'U', \text{or} 'L' \)
3. \( \text{init} \neq 'I' \text{ or} 'S' \)
4. \( \text{IA}(n+1) < 1 \)
5. \( \text{IA}(i+1)-\text{IA}(i) < 0 \), for any \( i = 1, n \)
6. \( \text{IPARM}(1) < 0 \)
7. \( \text{IPARM}(2) \neq 1, 3, 4, \text{or} 5 \)
8. \( \text{IPARM}(3) = 0 \text{ and} \text{IPARM}(2) = 3 \)
9. \( \text{IPARM}(4) \neq 1, 3, 4, \text{or} 5 \)
10. \( \text{RPARM}(1) < 0 \)
11. \( \text{RPARM}(3) = 0 \text{ and} \text{IPARM}(4) = 3 \)
12. \( \text{naux1} \) 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 \( \text{naux2} \neq 0 \), and \( \text{naux2} \) is too small—that is, less than the minimum required value. Return code 7 is returned for \( \text{naux2} \) if error 2015 is recoverable.

**Examples**

**Example 1**

This example finds the solution of the linear system \( Ax = b \) for the sparse matrix \( A \), 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 \( \text{RPARM}(1) \). The algorithm is allowed to perform 20 iterations. The process converges after 9 iterations. Matrix \( A \) is:

\[
\begin{bmatrix}
2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 1.0 & 2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 0.0 & 0.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 1.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & -1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & -1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & -1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0
\end{bmatrix}
\]
Call Statement and Input:

\[
\begin{bmatrix}
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & -1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & -1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 \\
\end{bmatrix}
\]

Example 2

This example finds the solution of the linear system \(Ax = b\) for the same sparse matrix \(A\) used in Example 1. It also uses the same right-hand side in \(b\) and the same initial guesses in \(x\). 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:

\[
\begin{bmatrix}
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & -1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & -1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 \\
\end{bmatrix}
\]

\[
\text{STOR INIT N AR JA IA B X IPARM RPARM AUX1 NAUX1 AUX2 NAUX2}
\]

\[
\text{CALL DSRIS( 'G', 'I', 9, AR, JA, IA, B, X, IPARM, RPARM, AUX1, 98, AUX2, 63 )}
\]

\[
\begin{align*}
AR &= (2.0, 2.0, -1.0, 1.0, 2.0, 1.0, 2.0, -1.0, 1.0, 2.0, -1.0, 1.0, 2.0, -1.0, 1.0, 2.0) \\
JA &= (1, 2, 3, 1, 3, 4, 5, 6, 5, 6, 5, 6, 7, 8, 9, 8, 9) \\
IA &= (1, 2, 4, 9, 12, 15, 18, 21, 23) \\
B &= (2.0, 1.0, 3.0, 2.0, 2.0, 2.0, 2.0, 2.0, 3.0) \\
X &= (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
\end{align*}
\]

IPARM(1) = 20
IPARM(2) = 2
IPARM(3) = 0
IPARM(4) = 1
IPARM(5) = 10
RPARM(1) = 1.0-7
RPARM(3) = 1.0

Output:

\[
X = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
\]

IPARM(6) = 9
RPARM(2) = 0.290-16

Chapter 10. Linear Algebraic Equations

847
**Example 3**

This example finds the solution of the linear system $Ax = b$ for the sparse matrix $A$, 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 $A$ is:

$$
\begin{bmatrix}
2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
-1.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & -1.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & -1.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & -1.0 & 0.0 & 2.0 & 0.0 & -1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & -1.0 & 0.0 & 2.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -1.0 & 0.0 & 2.0
\end{bmatrix}
$$

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, NAUX1, AUX2, NAUX2 )

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, 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, 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, 0.0)
IPARM(1) = 20
IPARM(2) = 1
IPARM(3) = 0
IPARM(4) = 4
IPARM(5) = 1
RPARM(1) = 1.0-7
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 $Ax = b$ for the same sparse matrix $A$ used in Example 3. However, matrix $A$ 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, -1.0, 2.0, -1.0, 2.0, -1.0, 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, 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, 0.0)
IPARM(1) = 20
IPARM(2) = 3
IPARM(3) = 5
IPARM(4) = 3
IPARM(5) = 1
RPARM(1) = 1.0-7
RPARM(3) = 2.0

Output:
X = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 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)

Purpose

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:

$$Ax = b$$

where $A$, $x$, and $b$ contain long-precision real numbers.

Note:

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 122, 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 1325.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL DSMCG (m, nz, ac, ka, lda, b, x, iparm, rparm, aux1, aux2, naux1, aux2, naux2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>dsmcg (m, nz, ac, ka, lda, b, x, iparm, rparm, aux1, naux1, aux2, naux2);</td>
</tr>
</tbody>
</table>

On Entry

$m$ is the order of the linear system $Ax = b$ and the number of rows in sparse matrix $A$.

Specified as: an integer; $m \geq 0$.

$nz$ is the maximum number of nonzero elements in each row of sparse matrix $A$.

Specified as: an 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 integers, where $1 \leq (\text{elements of} \ KA) \leq m$.

$lda$ is the leading dimension of the arrays specified for $ac$ and $ka$.

Specified as: an 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 $x$ 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:

  $\| r \|_2 / \| x \|_2 < \varepsilon$

  where $r = b - Ax$ is the residual, and $\varepsilon$ is the desired relative accuracy. $\varepsilon$ is
  stored in $RPARM(1)$.

  If $IPARM(2) = 1$, the conjugate gradient iterative procedure is stopped when:

  $\| r \|_2 / \lambda \| x \|_2 < \varepsilon$

  where $\lambda$ is an estimate to the minimum eigenvalue of the iteration matrix. $\lambda$
  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:

  $\| r \|_2 / \lambda \| x \|_2 < \varepsilon$

  where $\lambda$ 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 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: an 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 ≥ 3m.
If IPARM(2) = 1 and IPARM(1) ≠ 0, use naux1 ≥ 3m+2(IPARM(1)).
If IPARM(2) = 1 and IPARM(1) = 0, use naux1 ≥ 3m+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: an integer. When IPARM(3) = 10 or -10, naux2 must have at least the following value:

For 32-bit integer arguments
naux2 ≥ m(nz-1)1.5+2(m+6).

For 64-bit integer arguments
naux2 ≥ m(nz-1)2.0+3(m+6).
On Return

\( x \) is the vector 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, \( \text{IPARM}(i) \), where:
\( \text{IPARM}(1) \) is unchanged.
\( \text{IPARM}(2) \) is unchanged.
\( \text{IPARM}(3) \) is unchanged.
\( \text{IPARM}(4) \) contains the number of iterations performed by this subroutine.
Returned as: a one-dimensional array of length 4, containing integers.

\( rparm \)
is an array of parameters, \( \text{RPARM}(i) \), where:
\( \text{RPARM}(1) \) is unchanged.
\( \text{RPARM}(2) \) is unchanged if \( \text{IPARM}(2) = 0 \) or 2. If \( \text{IPARM}(2) = 1 \), \( \text{RPARM}(2) \) contains \( \lambda \), an estimate of the smallest eigenvalue of the iteration matrix.
\( \text{RPARM}(3) \) contains the estimate of the error of the solution. If the process converged, \( \text{RPARM}(3) \leq \epsilon \).
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 \( \text{IPARM}(3) = 10 \), \( aux2 \) contains the incomplete Cholesky factorization of matrix \( A \).
If \( \text{IPARM}(3) = -10 \), \( aux2 \) is unchanged.
See “Notes” for additional information on \( aux2 \). Returned as: an area of storage, containing long-precision real numbers.

Notes

1. When \( \text{IPARM}(3) = -10 \), this subroutine uses the incomplete Cholesky factorization in \( aux2 \), computed in an earlier call to this subroutine. When \( \text{IPARM}(3) = 10 \), 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 \( \text{IPARM}(3) = 10 \) on the first invocation. The incomplete factorization is stored in \( aux2 \). You may save computing time on subsequent calls by setting \( \text{IPARM}(3) = -10 \). 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 \( \epsilon \) (\( \text{RPARM}(1) \)) 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:
\[ \| b - Ax \|_2 / \lambda \| x \|_2 < \epsilon \]

where \( \lambda \) 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 \( \epsilon \), the algorithm takes fewer iterations to converge to a solution. If you specify a smaller \( \epsilon \), 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 117.

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 51.

**Function**

The sparse positive definite or negative definite linear system:

\[ Ax = b \]

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 [73 on page 1367] and [80 on page 1368], [44 on page 1366].

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 (see “DSRSM (Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode)” on page 1323).

**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 68.

- 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. \( m < 0 \)
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.

### Examples

#### Example 1

This example finds the solution of the linear system \( Ax = b \) for the sparse matrix \( A \), which is stored in compressed-matrix storage mode in arrays \( AC \) and \( KA \). The system is solved using the conjugate gradient method. Matrix \( A \) is:

\[
\begin{bmatrix}
2.0 & 0.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & -1.0 & 2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
-1.0 & 0.0 & 0.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & -1.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & -1.0 & 2.0 & -1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -1.0 & 2.0 & -1.0 & 0.0 \\
\end{bmatrix}
\]
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) = 1.0-7

\[
\begin{bmatrix}
2.0 & -1.0 & 0.0 \\
2.0 & -1.0 & 0.0 \\
-1.0 & 2.0 & 0.0 \\
-1.0 & 2.0 & -1.0 \\
\end{bmatrix}
\]

\[
AC = \begin{bmatrix}
1 & 4 & . \\
2 & 3 & . \\
2 & 3 & . \\
1 & 4 & 5 \\
\end{bmatrix}
\]

\[
KA = \begin{bmatrix}
4 & 5 & 6 \\
5 & 6 & 7 \\
6 & 7 & 8 \\
7 & 8 & 9 \\
8 & 9 \\
\end{bmatrix}
\]

\[
B = (1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0)
\]

\[
X = (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
\]

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 \(Ax = b\) for the same sparse matrix \(A\) 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) = 1.0D-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) = 0.1000D-15
DSDCG (Sparse Positive Definite or Negative Definite Symmetric Matrix Iterative Solve Using Compressed-Diagonal Storage Mode)

Purpose

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:

\[ Ax = b \]

where \( A, x, \) and \( b \) contain long-precision real numbers.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL DSDCG (iopt, m, nd, ad, lda, la, b, x, iparm, rparm, aux1, aux2, auxx1, auxx2, auxx2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>dsdcg (iopt, m, nd, ad, lda, la, b, x, iparm, rparm, aux1, auxx1, auxx2, auxx2);</td>
</tr>
</tbody>
</table>

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 \( A0 \).

Specified as: an 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: an integer; \( m \geq 0 \).

\( nd \)

is the number of nonzero diagonals stored in the columns of array \( A0 \), the number of columns in the array \( A0 \), and the number of elements in array \( LA \).

Specified as: an 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 \( A0 \), 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: an 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 118.

Specified as: a one-dimensional array of (at least) length \( nd \), containing integers, where \( 1-m \leq \text{elements of } LA \leq m-1 \).

\( b \) 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:

    \[
    \begin{align*}
    \text{IPARM}(1) &= 300 \\
    \text{IPARM}(2) &= 1 \\
    \text{IPARM}(3) &= 0 \\
    \text{RPARM}(1) &= 10^{-6}
    \end{align*}
    \]

- IPARM\((2)\) is the flag used to select the stopping criterion.
  - If IPARM\((2) = 0\), the conjugate gradient iterative procedure is stopped when:
    \[
    \frac{\| r \|_2}{\| x \|_2} < \varepsilon
    \]
    where \( r = b - Ax \) is the residual and \( \varepsilon \) is the desired relative accuracy. \( \varepsilon \) is stored in RPARM\((1)\).
  - If IPARM\((2) = 1\), the conjugate gradient iterative procedure is stopped when:
    \[
    \frac{\| r \|_2}{\lambda \| x \|_2} < \varepsilon
    \]
    where \( \lambda \) is an estimate to the minimum eigenvalue of the iteration matrix. \( \lambda \) 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:
    \[
    \frac{\| r \|_2}{\lambda \| x \|_2} < \varepsilon
    \]
    where \( \lambda \) 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 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: an 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 ≥ 3m.
- If IPARM(2) = 1 and IPARM(1) ≠ 0, use naux1 ≥ 3m+2(IPARM(1)).
- If IPARM(2) = 1 and IPARM(1) = 0, use naux1 ≥ 3m+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: an integer. When IPARM(3) = 10 or -10, naux2 must have at least the following value:

**For 32-bit integer arguments**

\[ \text{naux2} \geq m(3nd + 2) + 8 \]

**For 64-bit integer arguments**

\[ \text{naux2} \geq m(4nd + 3) + 12 \]

**On Return**

\( x \) is the vector \( x \) of length \( m \), containing the solution of the system \( Ax = b \). 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 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 \( \lambda \), 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) \( \leq \varepsilon \).

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 \( \varepsilon \) (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:

\[
\frac{\| b - Ax \|_2}{\| x \|_2} < \varepsilon
\]

where \( \lambda \) 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 \( \varepsilon \), the algorithm takes fewer iterations to converge to a solution. If you specify a smaller \( \varepsilon \), 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 117.

6. On output, array \( A0 \) 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 \( A0 \) 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 51.

**Function**

The sparse positive definite or negative definite linear system:

\[ Ax = b \]

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 \( A0 \) 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 [73 on page 1367] and [80 on page 1368]. [44 on page 1366].

**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 68.

- 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 $\langle y, Ay \rangle$ 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.

**Examples**

**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 $A0$ and $LA$. The system is solved using the two-term conjugate gradient method. In this example, IOPT = 0. Matrix $A$ is:
Call Statement and Input:

\[
\begin{bmatrix}
2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
-1.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & -1.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & -1.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
\end{bmatrix}
\]

Example 2

This example finds the solution of the linear system \(Ax = b\) for the same sparse matrix \(A\) 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:

\[
\begin{bmatrix}
2.0 & 0.0 & -1.0 \\
2.0 & 0.0 & -1.0 \\
2.0 & -1.0 & -1.0 \\
2.0 & -1.0 & -1.0 \\
2.0 & -1.0 & 0.0 \\
2.0 & -1.0 & 0.0 \\
\end{bmatrix}
\]

\(AD = (0, -2, 2)\)

\(B = (1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0)\)

\(X = (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)\)

Output:

\(X = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)\)

\(IPARM(4) = 5\)

\(RPARM(2) = 0\)

\(RPARM(3) = 0.460-16\)
\[
\begin{bmatrix}
2.0 & -1.0 \\
2.0 & -1.0 \\
2.0 & -1.0 \\
2.0 & -1.0 \\
2.0 & -1.0 \\
2.0 & -1.0 \\
\end{bmatrix}
\]

\[
\begin{align*}
AD &= \begin{bmatrix}
2.0 & -1.0 \\
2.0 & -1.0 \\
2.0 & -1.0 \\
2.0 & -1.0 \\
2.0 & -1.0 \\
2.0 & -1.0 \\
\end{bmatrix} \\
LA &= (0, -2) \\
B &= (1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0) \\
X &= (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
\end{align*}
\]

Output:
\[
X = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
\]
\[
IPARM(4) = 1 \\
RPARM(2) = 0 \\
RPARM(3) = 0.890-16
\]
DSMGCG (General Sparse Matrix Iterative Solve Using Compressed-Matrix Storage Mode)

Purpose

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, \((A + A^T)/2\), is positive definite. The sparse matrix is stored in compressed-matrix storage mode. Matrix \(A\) and vectors \(x\) and \(b\) are used:

\[ Ax = b \]

where \(A\), \(x\), and \(b\) contain long-precision real numbers.

Note:

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 122, 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 1323.

Syntax

**Fortran**

```
CALL DSMGCG (m, nz, ac, ka, lda, b, x, iparm, rparm, aux1, aux2, naux1, naux2)
```

**C and C++**

```
dsmgcg (m, nz, ac, ka, lda, b, x, iparm, rparm, aux1, aux2, naux1, naux2);
```

On Entry

- \(m\) is the order of the linear system \(Ax = b\) and the number of rows in sparse matrix \(A\).
  - Specified as: an integer; \(m \geq 0\).

- \(nz\) is the maximum number of nonzero elements in each row of sparse matrix \(A\).
  - Specified as: an 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 integers, where \(1 \leq (\text{elements of} \ KA) \leq m\).

- \(lda\) is the leading dimension of the arrays specified for \(ac\) and \(ka\).
  - Specified as: an 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 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:
    
    \[
    \begin{align*}
    IPARM(1) &= 300 \\
    IPARM(2) &= 0 \\
    IPARM(3) &= 10 \\
    RPARM(1) &= 10^{-6}
    \end{align*}
    \]

- \( IPARM(2) \) 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 \textit{On Return}.

Specified as: an array of (at least) length 4, containing integers, where:

\[
\begin{align*}
IPARM(1) &\geq 0 \\
IPARM(2) &\geq 0 \\
IPARM(3) &= 0, 10, \text{ or } -10
\end{align*}
\]

\( rparm \)

is an array of parameters, \( RPARM(i) \), where:

- \( RPARM(1) > 0 \), is the relative accuracy \( \epsilon \) used in the stopping criterion. The iterative procedure is stopped when:
  
  \[
  \frac{\| b-Ax \|}{\| x \|} < \epsilon
  \]

- \( RPARM(2) \) is reserved.

- \( RPARM(3) \), see \textit{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 aux1.

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: an 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 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: an integer. When $IPARM(3) = 10$, naux2 must have at least the following value:

For 32-bit integer arguments

$naux2 \geq 3 + 2m + 1.5nz(m)$

For 64-bit integer arguments

$naux2 \geq 12 + 3m + 2nz(m)$

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 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, \( RPARM(3) \leq RPARM(1) \)

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 \( \epsilon \) (\( RPARM(1) \)) 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:

\[
\| b-Ax \|_2 / \| x \|_2 < \epsilon
\]

As a result, if you specify a larger \( \epsilon \), the algorithm takes fewer iterations to converge to a solution. If you specify a smaller \( \epsilon \), 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 117.

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 51.
Function

The linear system:

\[ Ax = b \]

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 [105 on page 1369] and [107 on page 1369]. [44 on page 1366].

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 (see “DSRSM (Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode)” on page 1323).

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 68.

- 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. \( \text{IPARM}(1) < 0 \)
7. \( \text{IPARM}(2) < 0 \)
8. \( \text{IPARM}(3) \neq 0, 10, \text{or} -10 \)
9. \( \text{RPARM}(1) < 0 \)
10. \( \text{RPARM}(2) < 0 \)
11. Error 2015 is recoverable or \( \text{naux1} \neq 0 \), and \( \text{naux1} \) is too small—that is, less than the minimum required value. Return code 5 is returned if error 2015 is recoverable.
12. \( \text{naux2} \) is too small—that is, less than the minimum required value. Return code 5 is returned if error 2015 is recoverable.

**Examples**

**Example 1**

This example finds the solution of the linear system \( Ax = b \) for the sparse matrix \( A \), which is stored in compressed-matrix storage mode in arrays \( AC \) and \( KA \). The system is solved using the conjugate gradient squared method. Matrix \( A \) is:

\[
\begin{bmatrix}
2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 1.0 & 2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 0.0 & 0.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 1.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & -1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & -1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0
\end{bmatrix}
\]

**Note:** For input matrix \( KA \), \( . \) indicates any value between 1 and 9.

**Call Statement and Input:**

\[
\begin{array}{cccccccccccc}
M & NZ & AC & KA & LDA & B & X & \text{IPARM} & \text{RPARM} & \text{AUX1} & \text{NAUX1} & \text{AUX2} & \text{NAUX2} \\
9 & 3 & AC & KA & 9 & B & X & \text{IPARM} & \text{RPARM} & \text{AUX1} & 63 & \text{AUX2} & 0
\end{array}
\]

\[
\begin{align*}
\text{IPARM}(1) &= 20 \\
\text{IPARM}(2) &= 0 \\
\text{IPARM}(3) &= 0 \\
\text{RPARM}(1) &= 1.0 \cdot 7
\end{align*}
\]

\[
\begin{bmatrix}
2.0 & 0.0 & 0.0 \\
2.0 & -1.0 & 0.0 \\
1.0 & 2.0 & 0.0 \\
1.0 & 2.0 & -1.0
\end{bmatrix}
\]

\[
\begin{bmatrix}
1.0 & 2.0 & 0.0
\end{bmatrix}
\]

\[
\begin{bmatrix}
1.0 & 2.0 & 0.0
\end{bmatrix}
\]
Example 2

This example finds the solution of the linear system $Ax = b$ for the same sparse matrix $A$ 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:

```
CALL DSMCGC( 9, 3, AC, KA, 9, B, X, IPARM, RPARM, AUX1, NAUX1, AUX2, NAUX2
```

```
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, 2.0, 3.0)
X = (0.0, 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, 1.0)
IPARM(4) = 2
RPARM(3) = 0.290D-15
```
DSDGCG (General Sparse Matrix Iterative Solve Using Compressed-Diagonal Storage Mode)

Purpose

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, \((A + A^T)/2\), is positive definite. The sparse matrix is stored in compressed-diagonal storage mode. Matrix \(A\) and vectors \(x\) and \(b\) are used:

\[ Ax = b \]

where \(A\), \(x\), and \(b\) contain long-precision real numbers.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL DSDGCG (m, nd, ad, lda, la, b, x, iparm, rparm, aux1, aux2, naux1, aux2, naux2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>dsdgcg (m, nd, ad, lda, la, b, x, iparm, rparm, aux1, aux2, naux1, aux2, naux2);</td>
</tr>
</tbody>
</table>

On Entry

- \(m\) is the order of the linear system \(Ax = b\) and the number of rows in sparse matrix \(A\).
  Specified as: an integer; \(m \geq 0\).

- \(nd\) is the number of nonzero diagonals stored in the columns of array \(A0\), the number of columns in array \(A0\), and the number of elements in array \(LA\).
  Specified as: an 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 \(A0\), 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: an 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 \(A0\). For an explanation of how diagonal numbers are stored, see "Compressed-Diagonal Storage Mode" on page 118.
  Specified as: a one-dimensional array of (at least) length \(nd\), containing integers, where \(1-m \leq \text{elements of} \ LA \leq (m-1)\).

- \(b\)
  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.
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:

  
  \[
  \begin{align*}
  \text{IPARM(1) } & = 300 \\
  \text{IPARM(2) } & = 0 \\
  \text{IPARM(3) } & = 10 \\
  \text{RPARM(1) } & = 10^{-6}
  \end{align*}
  \]

- 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 integers, where:

  \[
  \begin{align*}
  \text{IPARM(1) } & \geq 0 \\
  \text{IPARM(2) } & \geq 0 \\
  \text{IPARM(3) } & = 0, 10, \text{ or } -10
  \end{align*}
  \]

`rparm` is an array of parameters, RPARM(i), where:
If RPARM(1) > 0, is the relative accuracy $\epsilon$ used in the stopping criterion. The iterative procedure is stopped when:

\[
\| b-Ax \|_2 / \| x \|_2 < \epsilon
\]
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 \textit{naux1}.

Specified as: an area of storage, containing long-precision real numbers.

\textit{naux1} is the size of the work area specified by \textit{aux1}—that is, the number of elements in \textit{aux1}.

Specified as: an integer, where:

If \textit{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, \textit{naux1} > 0 and must have at least the following value, where:

If IPARM(2) = 0, use \textit{naux1} \geq 7m.

If IPARM(2) > 0, use \textit{naux1} \geq (k+2)m+k(k+4)+1, where \(k = \text{PARM}(2)\).

\textit{aux2} is a storage work area used by this subroutine. If IPARM(3) = -10, \textit{aux2} must contain the incomplete LU factorization of matrix \(A\), computed in an earlier call to DSDGCG. The size of \textit{aux2} is specified by \textit{naux2}.

Specified as: an area of storage, containing long-precision real numbers.

\textit{naux2} is the size of the work area specified by \textit{aux2}—that is, the number of elements in \textit{aux2}.

Specified as: an integer. When IPARM(3) = 10 or -10, \textit{naux2} must have at least the following value:

\textbf{For 32-bit integer arguments}

\[ \textit{naux2} \geq 3 + 2m + 1.5nd(m) \]

\textbf{For 64-bit integer arguments}

\[ \textit{naux2} \geq 12 + 3m + 2nd(m) \]

\textbf{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.

\textbf{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 integers.

\textbf{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 $\epsilon$ (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:

$$\| b - Ax \|_2 / \| x \|_2 < \epsilon$$

As a result, if you specify a larger $\epsilon$, the algorithm takes fewer iterations to converge to a solution. If you specify a smaller $\epsilon$, 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 118.

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 51.

Function

The linear system:

$$Ax = b$$
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 [105 on page 1369] and [107 on page 1369]. [44 on page 1366].

**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 68.

- For error 2110, return code 1 indicates that the subroutine exceeded \( \text{IPARM}(1) \) 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 \( \text{IPARM}(3) = -10 \), and \( aux2 \) contains an incomplete factorization of the input matrix \( A \) that was computed by a previous call to the subroutine when \( \text{IPARM}(3) = 10 \). 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 \( \text{IPARM}(3) = 0 \) 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. \( \text{IPARM}(1) < 0 \)
7. \( \text{IPARM}(2) < 0 \)
8. \( \text{IPARM}(3) \neq 0, 10, \) or -10
9. \( \text{RPARM}(1) < 1.0 \)
10. 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.
11. \( \text{naux2} \) is too small—that is, less than the minimum required value. Return code 5 is returned if error 2015 is recoverable.

**Examples**

**Example 1**

This example finds the solution of the linear system \( \mathbf{A} \mathbf{x} = \mathbf{b} \) for the sparse matrix \( \mathbf{A} \), which is stored in compressed-diagonal storage mode in arrays \( \mathbf{AD} \) and \( \mathbf{LA} \). The system is solved using the conjugate gradient squared method. Matrix \( \mathbf{A} \) is:

\[
\begin{pmatrix}
2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 0.0 & 0.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 \\
0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 \\
0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 2.0 & 0.0 & -1.0 \\
0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 2.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 2.0 \\
\end{pmatrix}
\]

Call Statement and Input:

\[
\begin{array}{cccccccccccc}
M & ND & AD & LDA & LA & B & X & IPARM & RPARM & AUX1 & NAUX1 & AUX2 & NAUX2 \\
\hline
9 & 3 & AD & 9 & LA & B & X & IPARM & RPARM & AUX1 & 63 & AUX2 & 0 \\
\end{array}
\]

\( \text{CALL DSDCGC( 9, 3, AD, 9, LA, B, X, IPARM, RPARM, AUX1, 63, AUX2, 0 )} \)

- \( \text{IPARM(1) = 20} \)
- \( \text{IPARM(2) = 0} \)
- \( \text{IPARM(3) = 0} \)
- \( \text{RPARM(1) = 1.0-7} \)

\[
\begin{pmatrix}
2.0 & -1.0 & 0.0 \\
2.0 & -1.0 & 0.0 \\
2.0 & -1.0 & 0.0 \\
2.0 & -1.0 & 0.0 \\
\end{pmatrix}
\]

\( AD = \)

\[
\begin{pmatrix}
2.0 & -1.0 & 1.0 \\
2.0 & -1.0 & 1.0 \\
2.0 & -1.0 & 1.0 \\
2.0 & 0.0 & 1.0 \\
2.0 & 0.0 & 1.0 \\
\end{pmatrix}
\]

\( \text{LA} = (0, 2, -4) \)

\( \mathbf{B} = (1, 1, 1, 1, 2, 2, 2, 3, 3) \)

\( \mathbf{X} = (0, 0, 0, 0, 0, 0, 0, 0, 0) \)

Output:

\( \mathbf{X} = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0) \)

- \( \text{IPARM(4) = 8} \)
- \( \text{RPARM(3) = 0.308D-17} \)

**Example 2**

This example finds the solution of the linear system \( \mathbf{A} \mathbf{x} = \mathbf{b} \) for the same sparse matrix \( \mathbf{A} \) as in Example 1, which is stored in compressed-diagonal storage mode in arrays \( \mathbf{AD} \) and \( \mathbf{LA} \). 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:
CALL DSGC(E( 9 , 3 , AD , 9 , LA , B , X , IPARM , RPARM , AUX1 , 109 , AUX2 , 46 )

IPARM(1) = 20
IPARM(2) = 5
IPARM(3) = 10
RPARM(1) = 1.0-7
AD = (same as input AD in Example 1)
LA = (same as input LA in Example 1)
B = (1, 1, 1, 2, 2, 2, 3, 3)
X = (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)
IPARM(4) = 6
RPARM(3) = 0.250D-15
Linear Least Squares Subroutines

This contains the linear least squares subroutine descriptions.
SGESVD, DGESVD, CGESVD, ZGESVD, SGESDD, DGESDD, CGESDD, and ZGESDD (Singular Value Decomposition for a General Matrix)

Purpose

These subroutines compute the singular value decomposition of a general matrix $A$, optionally computing the left and/or right singular vectors. The singular value decomposition is written:

For SGESVD, DGESVD, SGESDD, and DGESDD, $A = U\Sigma V^T$, where $U^T = U^{-1}$ and $V^T = V^{-1}$.

For CGESVD, ZGESVD, CGESDD, and ZGESDD, $A = U\Sigma V_H$, where $U_H = U^{-1}$ and $V_H = V^{-1}$.

In the formulas above:

- $U$ and $V$ are general matrices whose first $\min(m, n)$ columns are the left and right singular vectors of $A$.
- $\Sigma$ is a diagonal matrix whose $\min(m, n)$ diagonal elements are the singular values of $A$.

<table>
<thead>
<tr>
<th>Table 182. Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A, U, vt, work$</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
<tr>
<td>Short-precision complex</td>
</tr>
<tr>
<td>Long-precision complex</td>
</tr>
</tbody>
</table>

$^a$LAPACK

Syntax

Fortran

CALL SGESVD | DGESVD (jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, info)

CALL CGESVD | ZGESVD (jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, info)

CALL SGESDD | DGESDD (jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, iwork, info)

CALL CGESDD | ZGESDD (jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, iwork, info)

C and C++

sgesvd | dgesvd (jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, info);

cgesvd | zgesvd (jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, info);

sgesdd | dgesdd (jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, iwork, info);

cgesdd | zgesdd (jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, iwork, info);
| LAPACK | info = LAPACKE_sgesvd | LAPACKE_dgesvd (matrix_layout, jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, superb); |
| LAPACK | info = LAPACKE_cgesvd | LAPACKE_zgesvd (matrix_layout, jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, superb); |
| LAPACK | info = LAPACKE_sgesdd | LAPACKE_dgesdd (matrix_layout, jobz, m, n, a, lda, s, u, ldu, vt, ldvt); |
| LAPACK | info = LAPACKE_cgesdd | LAPACKE_zgesdd (matrix_layout, jobz, m, n, a, lda, s, u, ldu, vt, ldvt); |

**On Entry**

(matrix_layout)

indicates whether the input and output matrices are stored in row major order or column major order, where:
- If matrix_layout = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If matrix_layout = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. It must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR

(jobu)

indicates the options for computing all or part of matrix U, where:
- If jobu = 'A', all m columns (the left singular vectors) are returned in array u.
- If jobu = 'S', the first min(m, n) columns (the left singular vectors) are returned in array u.
- If jobu = 'O', the first min(m, n) columns (the left singular vectors) are overwritten on the array a.
- If jobu = 'N', no columns (no left singular vectors) are computed.

(jobvt)

indicates the options for computing all or part of V\(^T\) (for SGESVD/DGESVD) or V\(^H\) (for CGESVD/ZGESVD), where:
- If jobvt = 'A', all n rows (the right singular vectors) are returned in array vt.
- If jobvt = 'S', the first min(m, n) rows (the right singular vectors) are returned in array vt.
- If jobvt = 'O', the first min(m, n) rows (the right singular vectors) are overwritten in array a.
- If jobvt = 'N', no rows (no right singular vectors) are computed.

(jobz)

indicates the options for computing all or part of U and V\(^T\) (for SGESDD/DGESDD) or V\(^H\) (for CGESDD/ZGESDD), where:
- If jobz = 'A', all m columns (the left singular vectors) are returned in array u and all n rows (the right singular vectors) are returned in array vt.
- If jobz = 'S', the first min(m, n) columns (the left singular vectors) are returned in array u and the first min(m, n) rows (the right singular vectors) are returned in array vt.
- If jobz = 'O' and m ≥ n, the first n columns (the left singular vectors) are overwritten in the array a and all rows (the right singular vectors) are returned in array vt.
If jobz = 'O' and \( m < n \), all columns (the left singular vectors) are returned in array \( u \) and first \( m \) rows (the right singular vectors) are overwritten on the array \( a \).

- If jobz = 'N', no columns (the left singular vectors) or rows (the right singular vectors) are computed.

\( m \) is the number of rows in matrix \( A \).
Specified as: an integer; \( m \geq 0 \).

\( n \) is the number of columns in matrix \( A \).
Specified as: an integer; \( n \geq 0 \).

\( a \) 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 182 on page 881.

\( lda \) is the leading dimension of the array specified for \( A \).
Specified as: an integer; \( lda > 0 \) and \( lda \geq m \).

\( s \) See "On Return"
\( u \) See "On Return"

\( ldu \) is the leading dimension of the array specified for \( U \).
Specified as: an integer; \( ldu > 0 \) and:

- If jobu = 'A' or 'S', \( ldu \geq \max(1, m) \).
- If jobz = 'A' or 'S', \( ldu \geq \max(1, m) \).
- If jobz = 'O' and \( m < n \), \( ldu \geq \max(1, m) \).

\( vt \) See "On Return"

\( ldvt \) is the leading dimension of the array specified for \( VT \).
Specified as: an integer; \( ldvt > 0 \) and:

- If jobvt = 'A', \( ldvt \geq \max(1, n) \).
- If jobvt = 'S', \( ldvt \geq \max(1, \min(m, n)) \).
- If jobz = 'A', \( ldvt \geq \max(1, n) \).
- If jobz = 'S', \( ldvt \geq \max(1, \min(m, n)) \).
- If jobz = 'O', and \( m \geq n \), \( ldvt \geq \max(1, n) \).

\( superb \) is a work area of size \( \min(m, n) - 1 \).
Specified as: an area of storage containing numbers of the data type indicated in Table 182 on page 881.

\( work \) is the work area used by these subroutines, where:

- If lwork = 0, \( work \) is ignored.
- If lwork ≠ 0, the size of \( work \) is determined as follows:
If $lwork \neq -1$, $work$ is (at least) of length $lwork$.

If $lwork = -1$, $work$ is (at least) of length 1.

Specified as: an area of storage, containing numbers of the data type indicated in Table 182 on page 881.

$lwork$

is the number of elements in array WORK.

Specified as: an integer; where:

- If $lwork = 0$, the subroutine dynamically allocates the workspace needed for use during this computation. The work area is deallocated before control is returned to the calling program.
- If $lwork = -1$, the subroutine performs a workspace query and returns the optimal required size of $work$ in $work_1$. No computation is performed and the subroutine returns after error checking is complete.
- Otherwise:

For SGESVD and DGESVD

\[ lwork \geq \max(1, 3\min(m, n) + \max(m,n), 5\min(m, n)) \]

Note: If $jobu = 'N$' or $jobvt = 'N'$, depending on $m$, $n$ and the implementation, $lwork \geq \max(1, 5\min(m, n))$ may be sufficient and less than the value shown above.

For CGESVD and ZGESVD

\[ lwork \geq \max(1, 2\min(m, n) + \max(m,n)) \]

Note: If $jobu = 'N$' or $jobvt = 'N'$, depending on $m$, $n$ and the implementation, $lwork \geq \max(1, 3\min(m, n))$ may be sufficient and less than the value shown above.

For SGESDD and DGESDD

If $jobz = 'N'$, $lwork \geq 3\min(m, n) + \max(m(m, n), 7\min(m, n))$

If $jobz = 'O$':

\[ lwork \geq 3\min(m, n) + \max(m(m, n), 5\min(m, n)\min(m, n) + 4\min(m, n)) \]

If $jobz = 'S$', $lwork \geq 4\min(m, n)\min(m, n) + 7\min(m, n)$

If $jobz = 'A$', $lwork \geq 4\min(m, n)\min(m, n) + 6\min(m, n) + \max(m, n)$

For CGESDD and ZGESDD

If $jobz = 'N$', $lwork \geq 2\min(m, n) + \max(m, n)$

If $jobz = 'O$', $lwork \geq 2\min(m, n)\min(m, n) + 2\min(m, n) + \max(m, n)$

If $jobz = 'S$', $lwork \geq \min(m, n)\min(m, n) + 3\min(m, n)$

If $jobz = 'A$', $lwork \geq \min(m, n)\min(m, n) + 2\min(m, n) + \max(m, n)$

$iwork$

is a storage work area of size $8\min(m,n)$.

Specified as: an integer array.

$rwork$

For CGESVD and ZGESVD

$rwork$ is a work area of size (at least) $5\min(m,n)$.

For CGESDD and ZGESDD

$rwork$ is a work area of size (at least) $\max(1, lrwork)$. 
If jobz = 'N', lwork ≥ 5*min(m,n).
If max(m, n) >> min(m, n), lwork ≥ 5*min(m,n)*min(m, n) + 5*min(m,n).
Otherwise, lwork ≥ max(5*min(m,n)*min(m, n) + 5*min(m,n), 2*max(m, n)*min(m,n) + 2*min(m, n)*min(m,n) + min(m, n)).

Note:
LRWORK depends on whether or not max(m,n) >> min(m, n). ESSL is compatible with LAPACK 3.6.1 where max(m,n) >> min(m, n) is defined as max(m,n) ≥ 17.0*min(m, n)/9.0. If this definition changes in future LAPACK releases, IBM will consider updating these subroutines. If IBM updates these subroutines, the updates could require modifications of the calling application program.

Specified as: an area of storage containing numbers of the data type indicated in Table 182 on page 881

On Return
a is overwritten as follows:
If jobu = 'O' or jobz = 'O' and m ≥ n, u is not referenced. Instead, a is overwritten with the first min(m, n) columns of U. These are the left singular vectors, stored column-wise.
If jobu = 'O' or jobz = 'O' and m < n, vt is not referenced. Instead, a is overwritten with the first min(m, n) rows of V^T (for SGESVD, DGESVD, SGESDD, or DGESDD) or V^H (for CGESVD, ZGESVD, CGESDD, or ZGESDD).
These are the right singular vectors, stored row-wise.
If jobu ≠ 'O' and jobvt ≠ 'O' and jobz ≠ 'O', the contents of a are overwritten on return. Returned as: an lda by (at least) n array, containing numbers of the data type indicated in Table 182 on page 881

s is the vector s containing the non-negative singular values in descending order in the first min(m, n) elements of s.
Returned as: a one-dimensional array of (at least) length min(m, n) containing numbers of the data type indicated in Table 182 on page 881

u If jobu = 'A' or if jobz = 'A' or if jobz = 'O' and m < n, u contains the m by m matrix U.
If jobu = 'S' or if jobz = 'S', u contains first min(m, n) columns of U. These are the left singular vectors, stored column-wise.
If jobu = 'O' or 'N' or jobz = 'N' or jobz = 'O' and m ≥ n, u is not referenced.
Returned as: an ldu by (at least) m (if jobu = 'A' or if jobz = 'A' or if jobz = 'O' and m < n) or min(m,n) (if jobu = 'S' or if jobz = 'S') array containing numbers of the data type indicated in Table 182 on page 881

vt For SGESVD, DGESVD, SGESDD, and DGESDD:
- If jobvt = 'A' or jobz = 'A' or jobz = 'O' and m ≥ n, vt contains the matrix V^T of order n.
- If jobvt = 'S' or jobz = 'S', vt contains the first min(m, n) rows of V^T. These are the right singular vector, stored row-wise.
- If jobvt = 'O' or 'N' or jobz = 'N' or jobz = 'O' and m < n, vt is not referenced.
Note: These subroutines return $V^T$ instead of $V$

For CGESVD, ZGESVD, CGESDD, and ZGESDD:
- If $\text{jobvt} = 'A'$ or $\text{jobz} = 'A'$ or $\text{jobz} = 'O'$ and $m \geq n$, $vt$ contains the matrix $V^H$ of order $n$.
- If $\text{jobvt} = 'S'$ or $\text{jobz} = 'S'$, $vt$ contains the first $\min(m, n)$ rows of $V^H$. These are the right singular vector, stored row-wise.
- If $\text{jobvt} = 'O'$ or 'N' or $\text{jobz} = 'O'$ or $\text{jobz} = 'O'$ and $m < n$, $vt$ is not referenced.

Note: These subroutines return $V^H$ instead of $V$

Returned as: an ldvt by (at least) $n$ array, containing numbers of the data type indicated in Table 182 on page 881

$\text{superb}$ is a work area used by this subroutine.
Returned as: an area of storage. If info > 0, $\text{superb}$ contains the unconverged superdiagonal elements of the upper bidiagonal matrix $B$ whose diagonal is in $s$.

$\text{work}$ is a work area used by this subroutine if lwork ≠ 0, where:
- If lwork ≠ 0 and lwork ≠ -1, its size is (at least) of length lwork.
- If lwork = -1, its size is (at least) of length 1.
If lwork = 0, $\text{work}$ is not referenced.
Returned as: an area of storage where:
- If lwork ≥ 1 or lwork = -1, then work is set to the optimal lwork value.
For SGESVD or DGESVD, if lwork ≥ 1 and info > 0, work contains the unconverged superdiagonal elements of an upper bidiagonal matrix $B$, whose diagonal is in array $S$ (not necessarily sorted). $B$ satisfies $A = UBV^T$, so it has the same singular values as $A$, and singular vectors related by $U$ and $V^T$.

$\text{rwork}$ is a work area used by this subroutine.
Returned as: an area of storage where:
For CGESVD and ZGESVD, if info > 0, rwork contains the unconverged superdiagonal elements of an upper bidiagonal matrix $B$ whose diagonal is in array $S$ (not necessarily sorted). $B$ satisfies $A = UBV^H$, so it has the same singular values as $A$, and singular vectors related by $U$ and $V^H$.

info has the following meaning:
- If info = 0, the subroutine completed successfully.
For SGESVD, DGESVD, CGESVD, ZGESVD, LAPACK_sgesvd, LAPACK_dgesvd, LAPACK_cgesvd, and LAPACK_zgesvd:
- If info > 0, info specifies how many superdiagonals of an intermediate bidiagonal form $B$ did not converge to zero.
For SGESDD, DGESDD, CGESDD, ZGESDD, LAPACKE_sgesdd,
LAPACKE_dgesdd, LAPACKE_cgesdd, and LAPACKE_zgesdd:

If info > 0, the algorithm did not converge to zero and the update
process failed. The singular values and singular vectors did not
converge.

Returned as:

• For SGESVD, DGESVD, CGESVD, ZGESVD, SGESDD, DGESDD, CGESDD,
  and ZGESDD, returned as: an integer; info ≥ 0.

• For LAPACKE_sgesvd, LAPACKE_dgesvd, LAPACKE_cgesvd,
  LAPACKE_zgesvd, LAPACKE_sgesdd, LAPACKE_dgesdd,
  LAPACKE_cgesdd, and LAPACKE_zgesdd, returned as an integer function
  value; info ≥ 0.

Returned as: an integer, info ≥ 0.

Notes

1. These subroutines accept lowercase letters for the jobu, jobvt, and jobz arguments.
2. In your C program, argument info must be passed by reference.
3. When you specify jobu = 'O' or 'N' or jobz = 'O' or 'N', you must specify a
dummy argument for u.
4. When you specify jobvt = 'O' or 'N' or jobz = 'O' or 'N', you must specify a
dummy argument for vt.
5. You cannot specify both jobu = 'O' and jobvt = 'O'.
6. a, s, u, vt, work, rwork, and iwork must have no common elements; otherwise,
  results are unpredictable.
7. For best performance, specify lwork = 0.

Function

These subroutines compute the singular value decomposition of a general matrix
A, optionally computing the left and/or right singular vectors. The singular value
decomposition is written:

For SGESVD, DGESVD, SGESDD, and DGESDD, $A = U \Sigma V^T$, where $U^T = U^{-1}$ and
$V^T = V^{-1}$

For CGESVD, ZGESVD, CGESDD, and ZGESDD, $A = U \Sigma V^H$, where $U^H = U^{-1}$ and
$V^H = V^{-1}$

In the formulas above:

• $U$ and $V$ are general matrices whose first min(m, n) columns are the left and
  right singular vectors of $A$.
• $\Sigma$ is a diagonal matrix whose min(m, n) diagonal elements are the singular
  values of $A$.

The computation involves the following steps:
1. If necessary, scale $A$
2. If necessary, compute QR or LQ factorization
3. Bidiagonalize the matrix
4. For SGESVD, DGESVD, CGESVD, and ZGESD, compute the singular values and, optionally, the left and/or right singular vectors from the bidiagonalized matrix.
   For SGESDD, DGESDD, CGESDD, and ZGESDD, compute the singular values and, optionally, the left and/or right singular vectors from the bidiagonalized matrix using the divide and conquer algorithm.
5. If necessary, update the singular vectors
6. If necessary, undo scaling

If m or n is 0, no computation is performed and the subroutine returns after doing some parameter checking.

See references [73 on page 1367][104 on page 1369].

Error conditions

Resource Errors
   lwork = 0 and unable to allocate work space

Computational Errors
   For SGESVD, DGESVD, CGESVD, and ZGESD, at least info superdiagonals of an intermediate bidiagonal form B did not converge to zero.
   For SGESDD, DGESDD, CGESDD, and ZGESDD, singular values and/or singular vectors did not converge.

Input-Argument Errors
   1. jobu ≠ 'A', 'S', 'O', or 'N'
   2. jobvt ≠ 'A', 'S', 'O', or 'N'
   3. jobu = 'O' and jobvt = 'O'
   4. jobz ≠ 'A', 'S', 'O', or 'N'
   5. m < 0
   6. n < 0
   7. lda ≤ 0
   8. m > lda
   9. ldu ≤ 0
   10. m > ldu and (jobu = 'A' or jobz = 'S')
   11. m > ldu and (jobz = 'A' or jobz = 'S')
   12. m > ldu and (jobz = 'O' and m < n
   13. ldvt ≤ 0
   14. n > ldvt and jobvt = 'A'
   15. n > ldvt and jobz = 'A'
   16. n > ldvt and jobz = 'O' and m ≥ n
   17. min(m, n) > ldvt and jobvt = 'S'
   18. min(m, n) > ldvt and jobz = 'S'
   19. lwork ≠ 0 and lwork ≠ -1 and lwork < the required value
   20. The size of a work array is greater than 2147483647 when 32-bit integers are used.

Examples

Example 1
This example shows how to find the singular values only of the real general matrix \( A \).

Notes:
1. For DGESVD, because \( \text{jobu} = 'N' \), argument \( u \) is not referenced.
2. For DGESVD, because \( \text{jobvt} = 'N' \), argument \( vt \) is not referenced.
3. For DGESDD, because \( \text{jobz} = 'N' \), arguments \( u \) and \( vt \) are not referenced.
4. Because \( lwork = 0 \), the subroutine dynamically allocates \( \text{WORK} \).
5. \( iwork \) is an integer work array of size 64.

Call Statement and Input:

\[
\begin{align*}
\text{JOBU} & \quad \text{JOBVT} & \quad M & \quad N & \quad A & \quad \text{LDA} & \quad S & \quad U & \quad \text{LDU} & \quad VT & \quad \text{LDVT} & \quad \text{WORK} & \quad \text{LWORK} & \quad \text{INFO} \\
\text{CALL DGESVD}('N', 'N', 4, 4, A, 4, S, U, 1, VT, 1, WORK, 0, INFO) \\
\text{–or–} \\
\text{CALL DGESDD}('N', 4, 4, A, 4, S, U, 1, VT, 1, WORK, 0, IWORK, INFO)
\end{align*}
\]

\[
A = \begin{bmatrix}
1.0 & 1.0 & 0.0 & 0.0 \\
0.0 & 2.0 & 1.0 & 0.0 \\
0.0 & 0.0 & 3.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 4.0
\end{bmatrix}
\]

Output:

Array \( A \) is overwritten.

\[
S = \begin{bmatrix}
4.260007 \\
3.107349 \\
2.111785 \\
0.858542
\end{bmatrix}
\]

\( \text{INFO} = 0 \)

Example 2

This example shows how to find the singular values of the real general matrix \( A \) and return all its left and right singular vectors \( U \) and \( VT \) in arrays \( U \) and \( V^T \).

Note:
1. Because \( lwork = 0 \), the subroutine dynamically allocates \( \text{WORK} \).
2. \( iwork \) is an integer work array of size 64.

Call Statement and Input:

\[
\begin{align*}
\text{JOBU} & \quad \text{JOBVT} & \quad M & \quad N & \quad A & \quad \text{LDA} & \quad S & \quad U & \quad \text{LDU} & \quad VT & \quad \text{LDVT} & \quad \text{WORK} & \quad \text{LWORK} & \quad \text{INFO} \\
\text{CALL DGESVD}('A', 'A', 3, 3, A, 3, S, U, 3, VT, 3, WORK, 0, INFO) \\
\text{–or–} \\
\text{CALL DGESDD}('A', 3, 3, A, 3, S, U, 3, VT, 3, WORK, 0, IWORK, INFO)
\end{align*}
\]

\[
A = \begin{bmatrix}
1.0 & 2.0 & 3.0 \\
2.0 & 4.0 & 5.0 \\
3.0 & 5.0 & 6.0
\end{bmatrix}
\]

Output:
Array \( A \) is overwritten.

\[
S = \begin{bmatrix}
11.344814 \\
0.515729 \\
0.170915
\end{bmatrix}
\]

\[
U = \begin{bmatrix}
-0.327985 & -0.736976 & -0.591009 \\
-0.591009 & -0.327985 & 0.736976 \\
-0.736976 & 0.591009 & -0.327985
\end{bmatrix}
\]

\[
VT = \begin{bmatrix}
-0.327985 & -0.591009 & -0.736976 \\
0.736976 & -0.591009 & 0.327985 \\
-0.591009 & 0.736976 & -0.327985
\end{bmatrix}
\]

\( INFO = 0 \)

**Example 3**

This example shows how to find the singular values of the real general matrix \( A \). Additionally:

The first \( \min(m, n) \) columns of its left singular vectors \( U \) are returned in array \( U \).

The first \( \min(m, n) \) rows of its right singular vectors \( V^T \) are returned in array \( A \).

**Notes:**

1. Because \( jobvt = 'O' \), argument \( vt \) is not referenced.
2. Because \( lwork = 0 \), the subroutine dynamically allocates \( WORK \).
3. \( iwork \) is an integer work array of size 64.

**Call Statement and Input:**

```
CALL DGESVD( 'S', 'O', 2, 4, A, 2, S, U, 2, VT, 1, WORK, 0, INFO )
```

**Output:**

- For DGESVD:
  \[
  A = \begin{bmatrix}
  1.0 & 2.0 & 3.0 & 4.0 \\
  5.0 & 6.0 & 7.0 & 8.0
  \end{bmatrix}
  \]

  \[
  S = \begin{bmatrix}
  14.227407 \\
  1.257330
  \end{bmatrix}
  \]

- For DGESDD, \( A \) has been overwritten on output.
\[ U = \begin{bmatrix}
-0.376168 & -0.926551 \\
-0.926551 & 0.376168
\end{bmatrix} \]

- For DGESVD, \( VT \) is not referenced.
- For DGESDD, \( VT \) is:

\[ VT = \begin{bmatrix}
-0.352062 & -0.443626 & -0.535190 & -0.626754 \\
0.758981 & 0.321242 & -0.116498 & -0.554238
\end{bmatrix} \]

INFO = 0

**Example 4**

This example shows how to find the singular values only of the complex general matrix \( A \).

**Notes:**
1. For ZGESVD, because \( jobu = 'N' \), argument \( u \) is not referenced.
2. For ZGESVD, because \( jobvt = 'N' \), argument \( vt \) is not referenced.
3. For ZGESDD, because \( jobz = 'N' \), arguments \( u \) and \( vt \) are not referenced.
4. Because \( lwork = 0 \), the subroutine dynamically allocates \( WORK \).
5. \( iwork \) is an integer work array of size 64.
6. \( rwork \) is an integer work array of size 64.

**Call Statement and Input:**

```
CALL ZGESVD( 'N', 'N', 4, 4, A, LDA, S, U, LDU, VT, LDVT, WORK, LWORK, RWORK, INFO )
```

- or -

```
CALL ZGESDD( 'N', 4, 4, A, LDA, S, U, LDU, VT, LDVT, WORK, LWORK, RWORK, IWORK, INFO )
```

\[ A = \begin{bmatrix}
(1.0, 1.0) & (1.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (2.0, -1.0) & (1.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (0.0, 0.0) & (3.0, 1.0) & (1.0, 0.0) \\
(0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (4.0, -1.0)
\end{bmatrix} \]

**Output:**

Array \( A \) is overwritten.

\[ S = \begin{bmatrix}
4.389511 \\
3.276236 \\
2.346361 \\
1.221907
\end{bmatrix} \]

INFO = 0

**Example 5**

This example shows how to find the singular values of the complex general matrix \( A \) and its left and right singular vectors.

**Note:**
1. Because \( lwork = 0 \), the subroutine dynamically allocates \( WORK \).
2. \( iwork \) is an integer work array of size 64.

3. \( rwork \) is an integer work array of size 60.

Call Statement and Input:

```
CALL ZGESVD( 'A', 'A', 3, 3, A, 3, S, U, 3, VT, 3, WORK, 0, RMWORK, INFO )
```

\[ A = \begin{bmatrix}
(1.0, 1.0) & (2.0,-1.0) & (3.0, 0.0) \\
(2.0,-1.0) & (4.0, 1.0) & (5.0,-1.0) \\
(3.0, 0.0) & (5.0,-1.0) & (6.0, 1.0)
\end{bmatrix} \]

Output:

Array A is overwritten.

\[ S = \begin{bmatrix}
11.370686 \\
2.386257 \\
1.006620
\end{bmatrix} \]

\[ U = \begin{bmatrix}
(-0.3265, 0.0409) & (0.0558, 0.4814) & (0.3504,-0.7308) \\
(-0.5822, 0.0725) & (-0.0823,-0.7730) & (0.1017,-0.2026) \\
(-0.7396, 0.0233) & (0.0036, 0.4009) & (-0.2805, 0.4616)
\end{bmatrix} \]

\[ VT = \begin{bmatrix}
(-0.3290, 0.0000) & (-0.5867,-0.0004) & (-0.7367,-0.0688) \\
(0.4846, 0.0000) & (-0.7774,-0.0071) & (0.3987, 0.0425) \\
(-0.8105, 0.0000) & (-0.2267,-0.0041) & (0.5375, 0.0533)
\end{bmatrix} \]

INFO = 0

Example 6

This example shows how to find the singular values of the complex general matrix \( A \). Additionally:

The first \( \min(m, n) \) columns of its left singular vectors \( U \) are returned in array \( U \).

The first \( \min(m, n) \) rows of its right singular vectors \( V^H \) are returned in array \( A \).

Notes:

1. Because \( jobvt = 'O' \), argument \( vt \) is not referenced.
2. Because \( iwork = 0 \), the subroutine dynamically allocates \( WORK \).
3. \( rwork \) is a work array of size 60.
Example 7

This example shows how to find the singular values of the complex general matrix $A$. Additionally:

All the columns of its left singular vectors $U$ are returned in array $U$.

The first $m$ rows of its right singular vectors $V^H$ are returned in array $A$.

Notes:
1. Because $jobvt = 'O'$, argument $vt$ is not referenced.
2. Because $lwork = 0$, the subroutine dynamically allocates $WORK$.
3. $rwork$ is an integer work array of size 64.
4. $rwork$ is a work array of size 60.

Call Statement and Input:

```call
ZGESDD( 'O', 2, 3, A, 2, S, 2, U, 2, V, 1, WORK, 60, RWORK, 60, IWORK, INFO )
```

Output:

```
A =  
| (1.0, 1.0) (2.0, 0.0) |
| (3.0, 0.0) (4.0, -1.0) |
| (5.0, 1.0) (6.0, 0.0) |
| (7.0, 0.0) (8.0, -1.0) |
```

```
A =  
| (-0.0488, -0.0488) (-0.6265, 0.0670) (-0.7722, -0.0447) |
| (0.65909, 0.6590) (-0.2388, 0.1973) (0.1352, -0.1316) |
```

```
S =  
| 8.665569 |
| 1.374982 |
```
\[ \mathbf{U} = \begin{bmatrix} (-0.4231, 0.0000) & (0.9061, 0.0000) \\ (-0.9038, 0.0646) & (-0.4221, 0.0301) \end{bmatrix} \]

INFO = 0
SGEQRF, DGEQRF, CGEQRF, and ZGEQRF (General Matrix QR Factorization)

Purpose

This subroutine computes the QR factorization of a general matrix

\[ A = QR \]

where:

For SGEQRF and DGEQRF, \( Q \) is an orthogonal matrix.
For CGEQRF and ZGEQRF, \( Q \) is a unitary matrix.
For \( m \geq n \), \( R \) is an upper triangular matrix.
For \( m < n \), \( R \) is an upper trapezoidal matrix.

Table 183. Data Types

<table>
<thead>
<tr>
<th>( A, \tau, \text{work} )</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGEQRF (^5)</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGEQRF (^7)</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGEQRF (^5)</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGEQRF (^7)</td>
</tr>
</tbody>
</table>
| \(^5\)LAPACK

Syntax

Fortran

\[
\text{CALL SGEQRF | DGEQRF | CGEQRF | ZGEQRF} (m, n, a, lda, tau, work, lwork, info)
\]

C and C++

\[
sgeqrf | dgeqrf | cgeqrf | zgeqrf (m, n, a, lda, tau, work, lwork, info);
\]

LAPACK

\[
\text{info = LAPACK\_sgeqrf | LAPACK\_dgeqrf | LAPACK\_cgeqrf | LAPACK\_zgeqrf (matrix\_layout, m, n, a, lda, tau);}
\]

On Entry

\textit{matrix\_layout}

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If \textit{matrix\_layout} = LAPACK\_ROW\_MAJOR, the matrices are stored in row major order.
- If \textit{matrix\_layout} = LAPACK\_COL\_MAJOR, the matrices are stored in column major order.

Specified as: an integer. It must be LAPACK\_ROW\_MAJOR or LAPACK\_COL\_MAJOR

\textit{m}

is the number of rows in matrix \( A \) used in the computation.

Specified as: an integer; \( m \geq 0 \).

\textit{n}

is the number of columns in matrix \( A \) used in the computation.

Specified as: an integer; \( n \geq 0 \).

\textit{a}

is the \( m \) by \( n \) general matrix \( A \) whose QR factorization is to be computed.

Specified as: an \textit{lda} by (at least) \( n \) array, containing numbers of the data type indicated in Table 183
lda
is the leading dimension of the array specified for a.
Specified as: an integer; lda > 0 and lda ≥ m.
tau
See On Return
work
has the following meaning:
If lwork = 0, work is ignored.
If lwork ≠ 0, work is the work area used by this subroutine, where:
• If lwork ≠ -1, its size is (at least) of length lwork.
• If lwork = -1, its size is (at least) of length 1.
Specified as: an area of storage containing numbers of data type indicated in Table 183 on page 895
lwork
is the number of elements in array WORK.
Specified as: an integer; where:
• If lwork = 0, these subroutines dynamically allocate the work area used by this subroutine. The work area is deallocated before control is returned to the calling program. This option is an extension to the LAPACK standard.
• If lwork = -1, these subroutines perform a work area query and return the optimal size of work in work. No computation is performed and the subroutine returns after error checking is complete.
• Otherwise, it must be:
  lwork ≥ max(1, n)
info
See On Return
On Return
a
is the updated general matrix A, containing the results of the computation.
The elements on and above the diagonal of the array contain the min(m, n) × n upper trapezoidal matrix R (R is upper triangular if m ≤ n). The elements below the diagonal with τ represent the matrix Q as a product of min(m, n) elementary reflectors.
Returned as: an lda by (at least) n array, containing numbers of the data type indicated in Table 183 on page 895
tau
is the vector τ, of length min(m, n), containing the scalar factors of the elementary reflectors.
Returned as: a one-dimensional array of (at least) length min(m, n), containing numbers of the data type indicated in Table 183 on page 895
work
is the work area used by this subroutine if lwork ≠ 0, where:
If lwork ≠ 0 and lwork ≠ -1, its size is (at least) of length lwork.
If lwork = -1, its size is (at least) of length 1.
Returned as: an area of storage, where:
If $lwork \geq 1$ or $lwork = -1$, then $work_1$ is set to the optimal $lwork$ value and contains numbers of the data type indicated in Table 183 on page 895. Except for $work_1$, the contents of $work$ are overwritten on return.

**info** indicates that a successful computation occurred.

Retained as:

- For SGEQRF, DGEQRF, CGEQRF, and ZGEQRF, returned as: an integer; info $\geq 0$.
- For LAPACKE_sgeqrf, LAPACKE_dgeqrf, LAPACKE_cgeqrf, and LAPACKE_zgeqrf, returned as an integer function value; info $\geq 0$.

### Notes and Coding Rules

1. In your C program, argument info must be passed by reference.
2. The vectors and matrices used in the computation must have no common elements; otherwise, results are unpredictable.
3. For best performance specify $lwork = 0$.

### Function

Compute the $QR$ factorization of a general matrix $A$

$$A = QR$$

where:

The matrix $Q$ is represented as a product of elementary reflectors:

$$Q = H_1 H_2 \ldots H_k$$

where:

$k = \min(m, n)$

For each $i$:

- For SGEQRF and DGEQRF, $H_i = I - \tau vv^T$
- For CGEQRF and ZGEQRF, $H_i = I - \tau vv^H$

$\tau$ is a scalar, stored on return in $\tau_i$
$v$ is a real vector with $v_{1:i-1} = \text{zero}$, $v_i = \text{one}$.
$v_{i+1:m}$ is stored on return in $A_{i+1:m, i}$
$I$ is the identity matrix

For $m \geq n$, $R$ is an upper triangular matrix.
For $m < n$, $R$ is an upper trapezoidal matrix.

If $m = 0$ or $n = 0$, no computation is performed and the subroutine returns after doing some parameter checking.

See references [61 on page 1367](#), [8 on page 1363](#), [76 on page 1368](#), [59 on page 1367](#), [60 on page 1367](#).

### Error conditions

**Resource Errors**

- $lwork = 0$ and unable to allocate work space.

**Computational Errors**

- None.
Input-Argument Errors

1. $m < 0$
2. $n < 0$
3. $lda \leq 0$
4. $lda < m$
5. $lwork \neq 0$, $lwork \neq -1$, and $lwork < \max(1, n)$

Examples

Example 1

This example shows the QR factorization of a general matrix $A$ of size $6 \times 2$.

Note: Because $lwork = 0$, DGEQRF dynamically allocates the work area used by this subroutine.

Call Statements and Input:

```fortran
CALL DGEQRF (6, 2, A, 6, TAU, WORK, 0, INFO)
```

General matrix $A$ of size $6 \times 2$:

$$
A = \begin{bmatrix}
0.00000 & 2.00000 \\
2.00000 & -1.00000 \\
2.00000 & -1.00000 \\
0.00000 & 1.50000 \\
2.00000 & -1.00000 \\
2.00000 & -1.00000
\end{bmatrix}
$$

Output:

General matrix $A$ of size $6 \times 2$.

$$
A = \begin{bmatrix}
-4.00000 & 2.00000 \\
0.50000 & 2.50000 \\
0.50000 & 0.285714 \\
0.00000 & -0.428571 \\
0.50000 & 0.285714 \\
0.50000 & 0.285714
\end{bmatrix}
$$

Vector $\tau$ of length 2:

$$
\tau = \begin{bmatrix}
1.00000 \\
1.40000
\end{bmatrix}
$$

INFO = 0

Example 2

This example shows the QR factorization of a general matrix $A$ of size $4 \times 5$.

Note: Because $lwork = 0$, DGEQRF dynamically allocates the work area used by this subroutine.

Call Statements and Input:

```fortran
CALL DGEQRF (4, 5, A, 4, TAU, WORK, 0, INFO)
```

General matrix $A$ of size $4 \times 5$:
Output:
General matrix $A$ of size $4 \times 5$:

\[
A = \begin{bmatrix}
0.500000 & 0.500000 & 1.207107 & 0.000000 & 1.707107 \\
0.500000 & -1.500000 & -0.500000 & 2.414214 & 0.707107 \\
0.500000 & 0.500000 & 2.071070 & 0.000000 & 0.292893 \\
0.500000 & -1.500000 & -0.500000 & 2.414214 & 0.707107
\end{bmatrix}
\]

Vector $\tau$ of length 4:

\[
\begin{bmatrix}
1.500000 \\ 1.666667 \\ 1.989949 \\ 0.000000
\end{bmatrix}
\]

INFO = 0

Example 3

This example shows the QR factorization of a general matrix $A$ of size $6 \times 2$.

Note: Because $lwork = 0$, ZGEQRF dynamically allocates the work area used by this subroutine.

Call Statements and Input:

```
CALL ZGEQRF (6, 2, A, 6, TAU, WORK, 0, INFO)
```

General matrix $A$ of size $6 \times 2$:

\[
A = \begin{bmatrix}
(-1.800000, -0.900000) & (1.100000, -0.800000) \\
(-1.600000, 1.000000) & (1.700000, 1.400000) \\
(-1.000000, -0.300000) & (1.200000, 0.300000) \\
(1.100000, -0.100000) & (0.700000, -1.900000) \\
(0.500000, 0.700000) & (-0.200000, -1.500000) \\
(-1.500000, -0.700000) & (1.900000, -0.600000)
\end{bmatrix}
\]

Output:
General matrix $A$ of size $6 \times 2$:

\[
A = \begin{bmatrix}
(3.660601, 0.000000) & (-1.731956, -0.524504) \\
(0.255874, -0.225302) & (-3.865905, 0.000000) \\
(0.187102, 0.024101) & (1.135165, -0.000348) \\
(0.193177, 0.050152) & (0.057186, -0.451900) \\
(-0.19713, -0.110108) & (-0.065903, -0.220144) \\
(0.288000, 0.080724) & (0.110147, -0.200172)
\end{bmatrix}
\]

Vector $\tau$ of length 2:

\[
\begin{bmatrix}
1.491723, 0.245861) & (1.268358, 0.545419)
\end{bmatrix}
\]

INFO = 0

Example 4
This example shows the QR factorization of a general matrix $A$ of size $3 \times 4$.

**Note:** Because $lwork = 0$, ZGEQRF dynamically allocates the work area used by this subroutine.

**Call Statements and Input:**

```
M  N  A  LDA  TAU  WORK  LWORK  INFO
```

CALL ZGEQRF ( 3 , 4 , A , 3 , TAU , WORK , 0 , INFO)

General matrix $A$ of size $3 \times 4$:

$$
A = \begin{bmatrix}
(-1.60, 0.10) & (0.30, 1.70) & (0.30, 0.20) & (-0.50, -1.80) \\
(-1.20, 0.00) & (-0.90, -0.50) & (1.50, 0.80) & (1.50, -1.10) \\
(-0.10, 1.30) & (-1.10, 0.50) & (0.40, -1.30) & (1.60, 0.70)
\end{bmatrix}
$$

Output:

General matrix $A$ of size $3 \times 4$:

$$
A = \begin{bmatrix}
(2.39, 0.00) & (0.64, -0.32) & (-1.67, -0.71) & (-0.18, 0.88) \\
(0.30, 0.01) & (2.23, 0.00) & (-0.70, 0.81) & (-2.25, -0.01) \\
(0.03, -0.33) & (0.48, -0.28) & (0.65, 0.00) & (-1.00, -1.77)
\end{bmatrix}
$$

Vector $\tau$ of length 3:

$$
\tau = \begin{bmatrix}
(1.67, -0.04) & (1.35, 0.49) & (1.99, 0.14)
\end{bmatrix}
$$

INFO = 0
SGELS, DGELS, CGELS, and ZGELS (Linear Least Squares Solution for a General Matrix)

Purpose

SGELS and DGELS compute the linear least squares solution for a real general matrix $A$ or its transpose using a $QR$ factorization without column pivoting, where $A$ is assumed to have full rank.

CGELS and ZGELS compute the linear least squares solution for a complex general matrix $A$ or its conjugate transpose using a $QR$ factorization without column pivoting, where $A$ is assumed to have full rank.

The following options are provided:

- If $\text{transa} = 'N'$ and $m \geq n$: find the least squares solution of an overdetermined system; that is, solve the least squares problem: minimize $\|B - AX\|$
- If $\text{transa} = 'N'$ and $m < n$: find the minimum norm solution of an underdetermined system; that is, the problem is: $AX = B$
- For SGELS and DGELS:
  - If $\text{transa} = 'T'$ and $m \geq n$: find the minimum norm solution of an underdetermined system; that is, the problem is $A^TX = B$
  - If $\text{transa} = 'T'$ and $m < n$: find the least squares solution of an overdetermined system; that is, solve the least squares problem: minimize $\|B - A^TX\|$
- For CGELS and ZGELS:
  - If $\text{transa} = 'C'$ and $m \geq n$: find the minimum norm solution of an underdetermined system; that is, the problem is $A^HX = B$
  - If $\text{transa} = 'C'$ and $m < n$: find the least squares solution of an overdetermined system; that is, solve the least squares problem: minimize $\|B - A^HX\|$

<table>
<thead>
<tr>
<th>$A$, $B$, work</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGELS$^\circ$</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGELS$^\circ$</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGELS$^\circ$</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGELS$^\circ$</td>
</tr>
</tbody>
</table>

$^\circ$LAPACK

Syntax

**Fortran**
```fortran
CALL SGELS | DGELS | CGELS | ZGELS (transa, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
```

**C and C++**
```c
sgels | dgels | cgels | zgels (transa, m, n, nrhs, a, lda, b, ldb, work, lwork, info);
```

**LAPACK**
```c
info = LAPACKE_sgels | LAPACKE_dgels | LAPACKE_cgels | LAPACKE_zgels
(matrix_layout, transa, m, n, nrhs, a, lda, b, ldb);
```

**On Entry**

- $\text{matrix\_layout}$ indicates whether the input and output matrices are stored in row major order or column major order, where:
If matrix_layout = LAPACK_ROW_MAJOR, the matrices are stored in row major order.

If matrix_layout = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. It must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR

transa
indicate the form of matrix A to use in the computation, where:

If transa = 'N', matrix A is used.
If transa = 'T', matrix A^T is used.
If transa = 'C', matrix A^H is used.

Specified as: a single character, where:
• For SGELS and DGELS, it must be 'N' or 'T'.
• For CGELS and ZGELS, it must be 'N' or 'C'.

m is the number of rows in matrix A used in the computation.
Specified as: an integer; m ≥ 0.

n is the number of columns in matrix A used in the computation.
Specified as: an integer; n ≥ 0.

nrhs
is the number of right-hand sides; that is, the number of columns in matrix B used in the computation.
Specified as: an integer; nrhs ≥ 0.

a is the m by n coefficient matrix A.

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: an lda by (at least) n array, containing numbers of the data type indicated in Table 184 on page 901

lda
is the leading dimension of the array specified for a.
Specified as: an integer; lda > 0 and lda ≥ m.

b is the matrix B of right-hand side vectors.
If transa = 'N', matrix B has m rows and nrhs columns.
For DGELS and SGELS, if transa = 'T', matrix B has n rows and nrhs columns.
For CGELS and ZGELS, if transa = 'C', matrix B has n rows and nrhs columns.

Specified as: the ldb by (at least) nrhs array, containing numbers of the data type indicated in Table 184 on page 901

ldb
is the leading dimension of the array specified for b.
Specified as: an integer; ldb > 0 and ldb ≥ max(m,n).

work
has the following meaning:
If lwork = 0, work is ignored.
If \( lwork \neq 0 \), \( work \) is the work area used by this subroutine, where:

- If \( lwork \neq -1 \), its size is (at least) of length \( lwork \).
- If \( lwork = -1 \), its size is (at least) of length 1.

Specified as: an area of storage containing numbers of data type indicated in Table 184 on page 901

\( lwork \)

is the number of elements in array \( work \).

Specified as: an integer; where:

- If \( lwork = 0 \), these subroutines dynamically allocate the work area used by this subroutine. The work area is deallocated before control is returned to the calling program. This option is an extension to the LAPACK standard.
- If \( lwork = -1 \), these subroutines perform a work area query and return the optimal size of \( work \) in \( work \).
- Otherwise, it must be:
  
  \[ lwork \geq \max(1, mn + \max(mn, nrhs)) \]

  where \( mn = \min(m, n) \).

\( info \)

See On Return

On Return

\( a \)

is the updated general matrix \( A \). The matrix \( A \) is 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 184 on page 901

\( b \)

is the updated general matrix \( B \), containing the results of the computation. \( B \) is overwritten by the solution vectors, stored columnwise:

- If \( transa = 'N' \) and \( m \geq n \), rows 1 to \( n \) of \( B \) contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements \( n+1 \) to \( m \) in that column.
- If \( transa = 'N' \) and \( m < n \), rows 1 to \( n \) of \( B \) contain the minimum norm solution vectors.

For SGELS and DGELS:

- If \( transa = 'T' \) and \( m \geq n \), rows 1 to \( m \) of \( B \) contain the minimum norm solution vectors.

- If \( transa = 'T' \) and \( m < n \), rows 1 to \( m \) of \( B \) contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements \( m+1 \) to \( n \) in that column.

For CGELS and ZGELS:

- If \( transa = 'C' \) and \( m \geq n \), rows 1 to \( m \) of \( B \) contain the minimum norm solution vectors.

- If \( transa = 'C' \) and \( m < n \), rows 1 to \( m \) of \( B \) contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements \( m+1 \) to \( n \) in that column.

Returned as: an \( ldb \) by (at least) \( nrhs \) array, containing numbers of the data type indicated in Table 184 on page 901

Chapter 10. Linear Algebraic Equations  903
work

is the work area used by this subroutine if lwork ≠ 0, where:
If lwork ≠ 0 and lwork ≠ -1, its size is (at least) of length lwork.
If lwork = -1, its size is (at least) of length 1.
Returned as: an area of storage, where:
If lwork ≥ 1 or lwork = -1, then work, is set to the optimal lwork value and contains numbers of the data type indicated in Table 184 on page 901.
Except for work, the contents of work are overwritten on return.

info

indicates that a successful computation occurred.
Returned as:
• For SGELS, DGELS, CGELS, and ZGELS, returned as: an integer; info ≥ 0.
• For LAPACKE_sgels, LAPACKE_dgels, LAPACKE_cgels, and LAPACKE_zgels, returned as an integer function value; info ≥ 0.

Notes and Coding Rules
1. In your C program, argument info must be passed by reference.
2. All subroutines accept lowercase letters for the transa argument.
3. The vectors and matrices used in the computation must have no common elements; otherwise, results are unpredictable.
4. For best performance specify lwork = 0.

Function

SGELS and DGELS compute the linear least squares solution for a real general matrix A or its transpose using a QR factorization without column pivoting, where A is assumed to have full rank.

CGELS and ZGELS compute the linear least squares solution for a complex general matrix A or its conjugate transpose using a QR factorization without column pivoting, where A is assumed to have full rank.

The following options are provided:
• If transa = 'N' and m ≥ n: find the least squares solution of an overdetermined system; that is, solve the least squares problem: minimize || B - AX ||
• If transa = 'N' and m < n: find the minimum norm solution of an underdetermined system; that is, the problem is: AX = B
• For SGELS and DGELS:
  – If transa = 'T' and m ≥ n: find the minimum norm solution of an underdetermined system; that is, the problem is AᵀX = B
  – If transa = 'T' and m < n: find the least squares solution of an overdetermined system; that is, solve the least squares problem: minimize || B - AᵀX ||
• For CGELS and ZGELS:
  – If transa = 'C' and m ≥ n: find the minimum norm solution of an underdetermined system; that is, the problem is AᴴX = B
  – If transa = 'C' and m < n: find the least squares solution of an overdetermined system; that is, solve the least squares problem: minimize || B - AᴴX ||
If \((m = 0\) and \(n = 0\)) or \(nrhs = 0\), then no computation is performed and the subroutine returns after doing some parameter checking.

See reference 73 on page 1367.

**Error conditions**

**Resource Errors**

\(lwork = 0\) and unable to allocate work space.

**Computational Errors**

None.

**Input-Argument Errors**

1. For SGELS and DGELS, \(transa \neq 'N'\) or 'T'
   
   For CGELS and ZGELS, \(transa \neq 'N'\) or 'C'
2. \(m < 0\)
3. \(n < 0\)
4. \(nrhs < 0\)
5. \(lda < m\)
6. \(lda \leq 0\)
7. \(ldb < \max(m, n)\)
8. \(ldb \leq 0\)
9. \(lwork \neq 0, lwork \neq -1, \) and \(lwork \leq \max(1, mn + \max(mn, nrhs))\) where \(mn = \min(m, n)\)

**Examples**

**Example 1**

This example finds the least squares solution of an overdetermined real general system; that is, it solves the least squares problem: minimize \(\| BAX \|\). Matrix \(A\) is size \(6 \times 2\) and matrix \(B\) is size \(6 \times 3\).

**Note:** Because \(lwork = 0\), DGELS dynamically allocates the work area used by this subroutine.

**Call Statements and Input:**

```Fortran
CALL DGELS ( 'N', 6, 2, 3, A, 6, B, 6, WORK, 0, INFO )
```

General matrix \(A\) of size \(6 \times 2\):

\[
A = \begin{bmatrix}
0.00000000 & 2.00000000 \\
2.00000000 & -1.00000000 \\
2.00000000 & -1.00000000 \\
.00000000 & 1.50000000 \\
2.00000000 & -1.00000000 \\
2.00000000 & -1.00000000
\end{bmatrix}
\]

General matrix \(B\) of size \(6 \times 3\):

\[
B = \begin{bmatrix}
1.00000000 & 4.00000000 & 1.00000000 \\
1.00000000 & 1.00000000 & 2.00000000 \\
1.00000000 & -1.00000000 & 1.00000000 \\
1.00000000 & 3.00000000 & 2.00000000
\end{bmatrix}
\]
Output:
General matrix $A$ is overwritten.
Solution matrix $X$ overwrites $B$:

\[
\begin{bmatrix}
.780000000 & 1.000000000 & 1.025000000 \\
.560000000 & 2.000000000 & .800000000 \\
.042857143 & -1.285714286 & -.250000000 \\
.185714286 & .428571429 & 1.250000000 \\
.042857143 & .714285714 & -.250000000 \\
.042857143 & -1.285714286 & -.250000000
\end{bmatrix}
\]

INFO = 0

Example 2

This example finds the minimum norm solution of an underdetermined real general system $A^T X = B$. Matrix $A$ is size $6 \times 2$. On input, matrix $B$ is size $2 \times 1$, stored in array $b$ with leading dimension 6.

Note: Because $lwork = 0$, DGELS dynamically allocates the work area used by this subroutine.

Call Statements and Input:

```
CALL DGELS ( 'T', 6, 2, 1, A, 6, B, 6, WORK, 0, INFO )
```

General matrix $A$ of size $6 \times 2$:

\[
\begin{bmatrix}
.000000000 & 2.000000000 \\
2.000000000 & -1.000000000 \\
2.000000000 & -1.000000000 \\
.000000000 & 1.500000000 \\
2.000000000 & -1.000000000 \\
2.000000000 & -1.000000000
\end{bmatrix}
\]

General matrix $B$ of size $2 \times 1$:

\[
\begin{bmatrix}
1.000000000 \\
1.000000000 \\
. \\
. \\
. \\
.
\end{bmatrix}
\]

Output:
General matrix $A$ is overwritten.
Solution matrix $X$ overwrites $B$:

\[
\begin{bmatrix}
.480000000 \\
.125000000 \\
.125000000
\end{bmatrix}
\]
Example 3

This example finds the minimum norm solution of an underdetermined real general system $AX = B$. Matrix $A$ is size $3 \times 4$. On input, matrix $B$ is size $3 \times 4$, stored in array $b$ with leading dimension 4.

Note: Because $lwork = 0$, DGELS dynamically allocates the work area used by this subroutine.

Call Statements and Input:

```
CALL DGELS ( 'N', 3, 4, 4, A, 3, B, 4, WORK, 0, INFO )
```

General matrix $A$ of size $3 \times 4$:

```
 0.500000000  0.500000000  0.500000000  0.500000000  
 0.500000000 -1.500000000  0.500000000 -1.500000000  
 1.000000000  1.000000000  0.000000000  1.000000000  
```

General matrix $B$ of size $3 \times 4$:

```
 1.000000000  1.000000000  1.000000000  0.000000000  
 1.000000000 -1.000000000  2.500000000  1.000000000  
 1.000000000  1.000000000  3.000000000  0.000000000  
```

Output:

General matrix $A$ is overwritten.

Solution matrix $X$ overwrites $B$:

```
 1.000000000  0.000000000  3.500000000  0.500000000  
 0.000000000  0.500000000  -0.250000000 -0.250000000  
 1.000000000  1.000000000 -1.000000000  0.000000000  
 0.000000000  0.500000000  -0.250000000 -0.250000000  
```

INFO = 0

Example 4

This example finds the least squares solution of an overdetermined real general system; that is, it solves the least squares problem: minimize $\| B - A^T X \|$. Matrix $A$ is size $3 \times 4$. On input, matrix $B$ is size $4 \times 4$.

Note: Because $lwork = 0$, DGELS dynamically allocates the work area used by this subroutine.

Call Statements and Input:

```
CALL DGELS ( 'T', 3, 4, 4, A, 3, B, 4, WORK, 0, INFO )
```

General matrix $A$ of size $3 \times 4$:
General matrix $B$ of size $4 \times 4$:

$$B = \begin{bmatrix} 1.000000000 & 1.000000000 & 1.000000000 & .000000000 \\ 1.000000000 & -1.000000000 & 2.000000000 & 2.414213562 \\ 1.000000000 & 1.000000000 & 3.000000000 & .000000000 \\ 1.000000000 & -1.000000000 & 4.000000000 & -.414213562 \end{bmatrix}$$

Output:

General matrix $A$ is overwritten.

Solution matrix $X$ overwrites $B$:

$$X = \begin{bmatrix} 2.000000000 & 1.000000000 & 6.121320344 & .500000000 \\ .000000000 & 1.000000000 & .707106781 & -.500000000 \\ .000000000 & .000000000 & 1.414213562 & -.2.000000000 \end{bmatrix}$$

INFO = 0

Example 5

This example finds the minimum norm solution of an underdetermined complex general system $AX = B$. Matrix $A$ is size $3 \times 4$. Matrix $B$ is size $3 \times 3$, stored in array $b$ with leading dimension 4.

Note: Because $\text{lwork} = 0$, ZGELS dynamically allocates the work area used by this subroutine.

Call Statements and Input:

```fortran
CALL ZGELS ( 'N' , 3 , 4 , 3 , A , 3 , B , 4 , WORK , 0 , INFO )
```

General matrix $A$ of size $3 \times 4$:

$$A = \begin{bmatrix} (1.00, 0.00) & (-2.00, 1.00) & (-3.00, -1.00) & (4.00, -3.00) \\ (1.00, -1.00) & (2.00, 2.00) & (-3.00, 0.00) & (-4.00, -2.00) \\ (1.00, -2.00) & (-2.00, 3.00) & (-3.00, 1.00) & (4.00, -1.00) \end{bmatrix}$$

General matrix $B$ of size $3 \times 3$:

$$B = \begin{bmatrix} (1.00, 0.00) & (0.00, 1.00) & (1.00, 1.00) \\ (-1.00, 1.00) & (1.00, -1.00) & (0.00, 0.00) \\ (2.00, 1.00) & (1.00, 2.00) & (-1.00, -1.00) \end{bmatrix}$$

Output:

General matrix $A$ is overwritten.

Solution matrix $X$ overwrites $B$:

$$X = \begin{bmatrix} (-0.16, 0.15) & (-0.08, 0.18) & (0.16, -0.31) \\ (0.11, 0.02) & (0.21, -0.50) & (-0.38, 0.65) \end{bmatrix}$$
\[
\begin{pmatrix}
(-0.13, -0.32) & (0.16, 0.12) & (-0.27, -0.28) \\
(0.37, -0.05) & (0.04, 0.06) & (-0.19, 0.33)
\end{pmatrix}
\]

\[INFO = 0\]

**Example 6**

This example finds the least squares solution of an overdetermined complex general system \(A\); that is, it solves the least squares problem: minimize \(\|B - AX\|\). Matrix \(A\) is size \(6 \times 2\). Matrix \(B\) is size \(6 \times 1\).

**Note:** Because \(lwork = 0\), ZGELS dynamically allocates the work area used by this subroutine.

Call Statements and Input:

\[
\begin{align*}
\text{TRANSA} & \quad M & \quad N & \quad NRHS & \quad A & \quad LDA & \quad B & \quad LDB & \quad WORK & \quad LWORK & \quad INFO \\
\end{align*}
\]

\[
\text{CALL ZGELS ( 'N', 6, 2, 1, A, 6, B, 6, WORK, 0, INFO )}
\]

Matrix \(A\) is the same used as input in Example 3 for ZGEQRF.

General matrix \(B\) of \(6 \times 1\):

\[
B = \begin{bmatrix}
(6.0, 0.0) \\
(5.0, 0.0) \\
(4.0, 0.0) \\
(3.0, 0.0) \\
(2.0, 0.0) \\
(1.0, 0.0)
\end{bmatrix}
\]

**Output:**

General matrix \(A\) is overwritten.

Solution matrix \(X\) overwrites \(B\):

\[
B = \begin{bmatrix}
(-1.135350, 0.520298) \\
(0.944064, 0.624509) \\
(1.062824, -0.899701) \\
(2.570856, 1.687827) \\
(2.556854, 2.835820) \\
(-3.982815, -0.231572)
\end{bmatrix}
\]

\[INFO = 0\]

**Example 7**

This example finds the minimum norm solution of an underdetermined complex general system \(A^H X = B\). Matrix \(A\) is size \(3 \times 3\). Matrix \(B\) is size \(3 \times 2\).

**Note:** Because \(lwork = 0\), ZGELS dynamically allocates the work area used by this subroutine.

Call Statements and Input:

\[
\begin{align*}
\text{TRANSA} & \quad M & \quad N & \quad NRHS & \quad A & \quad LDA & \quad B & \quad LDB & \quad WORK & \quad LWORK & \quad INFO \\
\end{align*}
\]

\[
\text{CALL ZGELS ( 'C', 3, 3, 2, A, 3, B, 3, WORK, 0, INFO )}
\]

Matrix \(A\) is the same used as input in Example 4 for CPOSV.

Matrix \(B\) is the same used as input in Example 4 for CPOSV.
Output:

General matrix $A$ is overwritten.

Solution matrix $X$ overwrites $B$:

$$B = \begin{bmatrix}
(2.0, -1.0) & (2.0, 0.0) \\
(1.0, -1.0) & (0.0, 1.0) \\
(3.0, 0.0) & (1.0, -1.0)
\end{bmatrix}$$

INFO = 0

Example 8

This example finds the least squares solution of an overdetermined complex general system; that is, it solves the least squares problem: minimize $\| B - A^H X \|$. Matrix $A$ is size $2 \times 6$. Matrix $B$ is size $6 \times 1$.

Note: Because $lwork = 0$, ZGELS dynamically allocates the work area used by this subroutine.

Call Statements and Input:

```fortran
CALL ZGELS ('C', 2, 6, 1, A, 2, B, 6, WORK, 0, INFO)
```

General matrix $A$ of size $2 \times 6$:

$$A = \begin{bmatrix}
(2.0, 0.0) & (6.0, 0.0) & (10.0, 0.0) & (14.0, 0.0) & (18.0, 0.0) & (22.0, 0.0) \\
(4.0, 0.0) & (8.0, 0.0) & (12.0, 0.0) & (16.0, 0.0) & (20.0, 0.0) & (24.0, 0.0)
\end{bmatrix}$$

General matrix $B$ of size $6 \times 1$:

$$B = \begin{bmatrix}
(6.0, 0.0) \\
(5.0, 0.0) \\
(4.0, 0.0) \\
(3.0, 0.0) \\
(2.0, 0.0) \\
(1.0, 0.0)
\end{bmatrix}$$

Output:

General matrix $A$ is overwritten.

Solution matrix $X$ overwrites $B$:

$$B = \begin{bmatrix}
(-3.50, 0.00) \\
(3.25, 0.00) \\
(0.00, 0.00) \\
(0.00, 0.00) \\
(0.00, 0.00) \\
(0.00, 0.00)
\end{bmatrix}$$

INFO = 0
SGELSD, DGELSD, CGELSD, and ZGELSD (Linear Least Squares Solution for a General Matrix Using the Singular Value Decomposition)

Purpose

These subroutines compute the linear least squares solution for a general matrix $A$ using the singular value decomposition.

The following options are provided:

- If $m \geq n$: find the least squares solution of an overdetermined system; that is, solve the least squares problem: minimize $\| B - AX \|$
- If $m < n$: find the minimum norm solution of an undetermined system; that is, the problem is: $AX = B$

Table 185. Data Types

<table>
<thead>
<tr>
<th>$A, B, work$</th>
<th>$s, rcond, rwork$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SGELSD$^a$</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DGELSD$^a$</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>CGELSD$^a$</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>ZGELSD$^a$</td>
</tr>
</tbody>
</table>

$^a$LAPACK

Syntax

Fortran

CALL SGELSD | DGELSD ($m, n, nrhs, a, b, ldb, s, rcond, rank, work, lwork, iwork, info$)
CALL CGELSD | ZGELSD ($m, n, nrhs, a, b, ldb, s, rcond, rank, work, rwork, iwork, info$)  

C and C++

sgebs | dgebs ($m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, iwork, info$);

cgels | zgels ($m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, rwork, iwork, info$);

LAPACK

info = LAPACKE_sgebs | LAPACKE_dgebs ($matrix_layout, m, n, nrhs, a, lda, b, ldb, s, rcond, rank$);  
info = LAPACKE_cgels | LAPACKE_zgels ($matrix_layout, m, n, nrhs, a, lda, b, ldb, s, rcond, rank$);

On Entry

$matrix_layout$

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If $matrix_layout =$ LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If $matrix_layout =$ LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. It must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR

$m$ is the number of rows in matrix $A$ and $B$.

Specified as: an integer; $m \geq 0$.

$n$ is the number of columns in matrix $A$.  

Chapter 10. Linear Algebraic Equations  911
Specified as: an integer; \( n \geq 0 \).

\( nrhs \)

is the number of right-hand sides; that is, the number of columns in matrix \( B \).

Specified as: an integer; \( nrhs \geq 0 \).

\( a \)

is the \( m \) by \( n \) general matrix \( A \).

Specified as: an \( lda \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 185 on page 911.

\( lda \)

is the leading dimension of the array specified for \( A \).

Specified as: an integer; \( lda > 0 \) and \( lda \geq m \).

\( b \)

is the general matrix \( B \) containing the \( nrhs \) right-hand sides of the system. The right-hand sides, each of length \( m \), reside in the columns of matrix \( B \).

Specified as: the \( ldb \) by (at least) \( nrhs \) array, containing numbers of the data type indicated in Table 185 on page 911.

\( ldb \)

is the leading dimension of the array specified for \( b \).

Specified as: an integer; \( ldb \geq \max(m,n) \).

\( rcond \)

is used to determine the effective rank of matrix \( A \). Singular values of \( s_i \leq (rcond)(s_i) \) are treated as zero.

If \( rcond \) is less than or equal to zero or \( rcond \) is greater than or equal to one, then an \( rcond \) value of \( \epsilon \) is used, where \( \epsilon \) is the machine precision.

Specified as: a number of data type indicated in Table 185 on page 911.

\( work \)

is the work area used by this subroutine, where:

If \( lwork=0 \), work is ignored.

If \( lwork \neq 0 \), the size of \( work \) is determined as follows:

- If \( lwork \neq -1 \), work is (at least) of length \( lwork \).
- If \( lwork=-1 \), work is (at least) of length 1.

Specified as: an area of storage containing numbers of data type indicated in Table 185 on page 911.

\( lwork \)

is the number of elements in array \( work \).

Specified as: an integer; where:

- If \( lwork=0 \), these subroutines dynamically allocate the work area used by this subroutine. The work area is deallocated before control is returned to the calling program. This option is an extension to the LAPACK standard.
- If \( lwork=1 \), these subroutines perform a work area query and return the optimal size of \( work \) in \( work \). No computation is performed, and the subroutine returns after error checking is complete.
- Otherwise:

**For SGELSD and DGELSD**

\[
\text{\textit{lwork}} \geq 12r + 2(r)(smlsz) + 8(r)(nlvl) + (r)(nrhs) + (smlsz+1)^2, \text{ where:}
\]

\[
- r = \min(m,n)
\]
- smlsiz = 25
- nlvl = max(0, int(log₂(r/(smlsiz+1))) + 1)

For CGELSD and ZGELSD
\[ lwork \geq \max(1, m+n+r, 2r + (r)(nrhs)), \text{ where } r = \min(m, n). \]

**Note:** These formulas represent the minimum workspace required. For best performance, specify either \( lwork = -1 \) (to obtain the optimal size to use) or \( lwork = 0 \) (to direct the subroutine to dynamically allocate the workspace).

**rwork** is a work area of size \( \max(1, lrwork) \), where:
\[ lrwork \geq 10r + 2(r)(smlsiz) + 8(r)(nlvl) + 2(smlsiz)(nrhs) + \max((smlsiz+1)^2, n(1 + nrhs) + 2nrhs), \]
- \( r = \min(m, n) \)
- \( smlsiz = 25 \)
- \( nlvl = \max(0, \text{int}(\log_2(r/(smlsiz+1)))) + 1 \)

Specified as: an area of storage containing numbers of data type indicated in Table 185 on page 911

**iwork** is a work area of size \( \max(1, liwork) \), where:
\[ liwork \geq 3(r)(nlvl) + 11r \]
- \( r = \min(m, n) \)
- \( smlsiz = 25 \)
- \( nlvl = \max(0, \text{int}(\log_2(r/(smlsiz+1)))) + 1 \)

Specified as: an area of storage containing integers.

**info**
See **On Return**

**On Return**

- \( a \) The matrix \( A \) is 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 185 on page 911

- \( b \) is the updated general matrix \( B \), containing the results of the computation. \( B \) is overwritten by the \( n \) by \( nrhs \) solution matrix \( X \).
  - If \( m \geq n \) and \( \text{rank} = n \), rows 1 to \( n \) of \( B \) contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements \( n + 1 \) to \( m \) in that column.
  - If \( m < n \), rows 1 to \( n \) of \( B \) contain the minimum norm solution vectors.
  Returned as: an \( ldb \) by (at least) \( nrhs \) array, containing numbers of the data type indicated in Table 185 on page 911

- \( s \) is the vector \( s \) containing the singular values of matrix \( A \).
  Returned as: a one-dimensional array of (at least) length \( \min(m,n) \) containing numbers of the data type indicated in Table 185 on page 911

- \( \text{rank} \) is the effective rank of \( A \); that is the number of singular values that are greater than \( \text{rcond}(s) \).
  Returned as: an integer.
work is the work area used by this subroutine if lwork ≠ 0, where:
If lwork ≠ 0 and lwork ≠ -1, its size is (at least) of length lwork.
If lwork=-1, its size is (at least) of length 1.
Returned as: an area of storage, where:
If lwork ≥ 1 or lwork=-1, then work1 is set to the optimal lwork value and
contains numbers of the data type indicated in Table 185 on page 91.
Except for work1, the contents of work are overwritten on return.

rwork is a work area used by these subroutines.
Returned as: an area of storage where, if info = 0, rwork1 is set to the minimum
size of rwork.

iwork is a work area used by these subroutines.
Returned as: an area of storage where, if info = 0, iwork1 is set to the minimum
size of iwork.

info has the following meaning:
If info = 0, the subroutine completed successfully.
If 0 < info ≤ max(m, n), info specifies how many superdiagonals of an
intermediate bidiagonal form did not converge to zero.
If info = max(m, n) + 1, a singular value failed to converge. SGELSD, DGELSD,
CGELSD, and ZGELSD
Returned as:
- For SGELSD, DGELSD, CGELSD, and ZGELSD, returned as: an integer; info
  ≥ 0.
- For LAPACKE_sgelsd, LAPACKE_dgelsd, LAPACKE_cgelsd, and
  LAPACKE_zgelsd, returned as an integer function value; info ≥ 0.

Notes and Coding Rules
1. In your C program, arguments rank and info must be passed by reference.
2. a, b, s, work, rwork and iwork must have no common elements; otherwise, results
   are unpredictable.
3. For best performance, specify lwork = 0.

Function

These subroutines compute the linear least squares solution for a general matrix A
using the singular value decomposition.

The following options are provided:
• If m ≥ n: find the least squares solution of an overdetermined system; that is,
  solve the least squares problem: minimize ∥ B - AX ∥
• If m < n: find the minimum norm solution of an undetermined system; that is,
  the problem is: AX=B

See reference [34 on page 1365], [73 on page 1367].
Error conditions

Resource Errors
   iw\text{ork}=0 and unable to allocate work space.

Computational Errors
   • Superdiagonals of an intermediate bidiagonal form did not converge to zero.
   • A singular value failed to converge.

Input-Argument Errors
   1. \( m < 0 \)
   2. \( n < 0 \)
   3. \( nrhs < 0 \)
   4. \( lda \leq 0 \)
   5. \( lda < m \)
   6. \( ldb \leq 0 \)
   7. \( ldb < \text{max}(m,n) \)
   8. \( iw\text{ork} \neq 0 \) and \( iw\text{ork} \neq -1 \) and \( iw\text{ork} < \) the minimum required value
   9. \( rw\text{ork} \neq 0 \) and \( rw\text{ork} \neq -1 \) and \( rw\text{ork} < \) the minimum required value
   10. \( iw\text{ork} \neq 0 \) and \( iw\text{ork} \neq -1 \) and \( iw\text{ork} < \) the minimum required value

Examples

Example 1

This example finds the least squares solution of an overdetermined real general
system; that is, it solves the least squares problem: minimize \( \| B-AX \| \). Matrix
\( A \) is size \( 6 \times 2 \) and matrix \( B \) is size \( 6 \times 3 \).

Notes:
   • Because \( iw\text{ork}=0 \), DGELSD dynamically allocates the work area used by this
     subroutine.
   • \( iw\text{ork} \) is an integer work array of size 22.

Call Statements and Input:

\begin{verbatim}
<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>N</td>
<td>NRHS</td>
<td>A</td>
<td>LDA</td>
<td>B</td>
<td>LDB</td>
<td>S</td>
<td>RCOND</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
<td>A</td>
<td>6</td>
<td>B</td>
<td>6</td>
<td>5</td>
<td>RCOND</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>CALL DGELSD ( 6 , 2 , 3 , A , 6 , B , 6 , 5 , RCOND , RANK , WORK , 0 , IWORK , INFO )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\end{verbatim}

\begin{verbatim}
A = (same as input A in Example 1 for DGELS)  
B = (same as input B in Example 1 for DGELS)  
RCOND = .7450580-08
\end{verbatim}

Output:

General matrix \( A \) is overwritten.

Solution matrix \( X \) overwrites \( B \):

\begin{verbatim}
B =
   [ 0.78000000  1.00000000  1.02500000 
    0.56000000  2.00000000  0.80000000 
    0.04285714 -1.28571428 -0.25000000 
    0.18571429  0.42857143  1.25000000 
    0.04285714  0.71428571 -0.25000000 
    0.04285714 -1.28571428 -0.25000000 ]
\end{verbatim}
Example 2

This example finds the minimum norm solution of an underdetermined real general system $AX = B$. Matrix $A$ is size $3 \times 4$. On input, matrix $B$ is size $3 \times 3$, stored in array $b$ with leading dimension 4.

Notes:
- Because $lwork=0$, DGELSD dynamically allocates the work area used by this subroutine.
- $iwork$ is an integer work array of size 33.

Call Statements and Input:

```
CALL DGELSD (3, 4, 4, A, 3, B, 4, 4, S, RCOND, RANK, WORK, 0, IWORK, INFO)
```

Output:

General matrix $A$ is overwritten.

Solution matrix $X$ overwrites $B$:

$$
B = \begin{bmatrix}
1.000000000 & 0.000000000 & 3.500000000 & 0.500000000 \\
0.000000000 & 0.500000000 & -0.250000000 & -0.250000000 \\
1.000000000 & 1.000000000 & -1.000000000 & 0.000000000 \\
0.000000000 & 0.500000000 & -0.250000000 & -0.250000000
\end{bmatrix}
$$

Example 3

This example finds the least squares solution of an overdetermined complex general system; that is, it solves the least squares problem: minimize $\| B-AX \|$. Matrix $A$ is size $6 \times 2$ and matrix $B$ is size $6 \times 1$.

Notes:
- Because $lwork=0$, ZGELSD dynamically allocates the work area used by this subroutine.
- $rwork$ is a real work array of size 871.
- $iwork$ is an integer work array of size 22.

Call Statements and Input:

```
CALL ZGELSD (6, 2, 1, A, 6, B, 6, 5, S, RCOND, RANK, WORK, 0, RWORK, IWORK, INFO)
```
\( A = \) (same as input \( A \) in Example 6 for ZGELS)

\( B = \) (same as input \( B \) in Example 6 for ZGELS)

\( \text{RCOND} = .7450580-08 \)

**Output:**

General matrix \( A \) is overwritten.

Solution matrix \( X \) overwrites \( B \):

\[
\begin{bmatrix}
(-1.135350, & 0.520298) \\
(0.944064, & 0.624509) \\
(1.062824, & -0.899701) \\
(2.570856, & 1.687826) \\
(2.556854, & 2.835820) \\
(-3.982815, & -0.231572)
\end{bmatrix}
\]

\[
\begin{bmatrix}
4.781121271 \\
2.959878261
\end{bmatrix}
\]

\( \text{RANK} = 2 \)

\( \text{INFO} = 0 \)

**Example 4**

This example finds the minimum norm solution of an underdetermined complex general system \( AX = B \). Matrix \( A \) is size \( 3 \times 4 \). On input, matrix \( B \) is size \( 3 \times 3 \), stored in array \( b \) with leading dimension 4.

Notes:

- Because \( lwork=0 \), ZGELSD dynamically allocates the work area used by this subroutine.
- \( rwork \) is a real work array of size 1081.
- \( iwork \) is an integer work array of size 33.

Call Statements and Input:

\[
\begin{bmatrix}
\end{bmatrix}
\]

```
CALL ZGELSD (3, 4, 3, A, 3, B, 4, S, RCOND, RANK, WORK, LWORK, RWORK, IWORK, INFO)
```

\( A = \) (same as input \( A \) in Example 5 for ZGELS)

\( B = \) (same as input \( B \) in Example 5 for ZGELS)

\( \text{RCOND} = -1 \)

**Output:**

General matrix \( A \) is overwritten.

Solution matrix \( X \) overwrites \( B \):

\[
\begin{bmatrix}
(-0.16, & 0.15) & (-0.08, & 0.18) & (0.16, & -0.31) \\
(0.11, & 0.02) & (0.21, & -0.50) & (-0.38, & 0.65) \\
(-0.13, & -0.32) & (0.16, & 0.12) & (-0.27, & -0.28) \\
(0.37, & -0.05) & (0.04, & 0.06) & (-0.19, & 0.33)
\end{bmatrix}
\]

\[
\begin{bmatrix}
9.895527537 \\
4.876518979 \\
1.816066467
\end{bmatrix}
\]

\( \text{RANK} = 3 \)
INFO = 0
SGESVF and DGESVF (Singular Value Decomposition for a General Matrix)

Purpose

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 186. Data Types

<table>
<thead>
<tr>
<th>$A$, $B$, $s$, $aux$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGESVF</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGESVF</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SGESVF</th>
<th>DGESVF ($i\text{opt}$, $a$, $lda$, $b$, $ldb$, $nb$, $s$, $m$, $n$, $aux$, $naux$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>sgesvf</td>
<td>dgesvf ($i\text{opt}$, $a$, $lda$, $b$, $ldb$, $nb$, $s$, $m$, $n$, $aux$, $naux$);</td>
</tr>
</tbody>
</table>

On Entry

$i\text{opt}$

indicates the type of computation to be performed, where:

- If $i\text{opt} = 0$ or 10, singular values are computed.
- If $i\text{opt} = 1$ or 11, singular values and $V$ are computed.
- If $i\text{opt} = 2$ or 12, singular values, $V$, and $U^TB$ are computed.

Specified as: an integer; $i\text{opt} = 0$, 1, 2, 10, 11, or 12.

- If $i\text{opt} < 10$, singular values are unordered.
- If $i\text{opt} \geq 10$, singular values are sorted in descending order and, if applicable, the columns of $V$ and the rows of $U^TB$ 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 186](#).

$lda$

is the leading dimension of the array specified for $a$.

Specified as: an integer; $lda > 0$ and $lda \geq \max(m, n)$.

$b$

has the following meaning, where:

- If $i\text{opt} = 0$, 1, 10, or 11, this argument is not used in the computation.
- If $i\text{opt} = 2$ 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 186](#).

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, $AX = B$. (The $nb$
column vectors of \( B \) contain right-hand sides for \( nb \) distinct linear least squares problems.) However, if the matrix \( U^T \) 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: an 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: an 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: an integer; \( m \geq 0. \)

**n**

is the number of columns in matrix \( A \) and the number of elements in vector \( s \).

Specified as: an 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 186 on page 919.

**naux**

is the size of the work area specified by \( aux \)—that is, the number of elements in \( aux. \)

Specified as: an 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 \( n+1, \) \( n+2, \ldots, \) \( m \) 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 186 on page 919.

\( b \) 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 \( U^T B. \)

If \( iopt = 12, \) the rows of \( U^T B \) are swapped to correspond to the sorted singular values. If \( m > n, \) rows \( n+1, \) \( n+2, \ldots, \) \( m \) 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 186 on page 919.

\( s \) 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 186 on page 919. \( s \geq 0, \) where:

If \( iopt < 10, \) the singular values are unordered in \( s. \)

If \( iopt \geq 10, \) the singular values are sorted in descending order in \( s; \) that is, \( s_1 \geq s_2 \geq \ldots \geq s_n \geq 0. \) If applicable, the columns of \( V \) and the rows of \( U^T B \) 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.

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 51.

Function

The singular value decomposition of a real general matrix is computed as follows:

\[ A = U \Sigma V^T \]

where:

\[ U^T U = V^T V = V V^T = I \]

\( A \) 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 \( A. \)

\( U^T B \) is an \( n \) by \( nb \) real general matrix. On output, \( U^T B \) overwrites the first \( n \) rows and \( nb \) columns of \( B. \)
Σ is an \( n \times n \) 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 [118 on page 1370].)
   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 [20 on page 1364].)
   a. Compute the QR decomposition of matrix \( A \) using Householder transformations; that is, \( A = QR \).
   b. Apply the Golub-Reinsch Algorithm to the matrix \( R \).
   If \( R = X W Y^T \) is the singular value decomposition of \( R \), the singular value decomposition of matrix \( A \) is given by:

\[
A = Q \begin{bmatrix} X \\ 0 \end{bmatrix} W Y^T
\]

where:

\[
U = Q \begin{bmatrix} X \\ 0 \end{bmatrix}
\]

\[\Sigma = W\]

\[V = Y\]

Also, see references [20 on page 1364], [69 on page 1367], [92 on page 1368], and pages 134 to 151 in reference [118 on page 1370]. 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 \( (s_p, j = n, n-1, ..., i+1) \) 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.
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.

Examples

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:

\[
\text{CALL DGESVF( 0, A, 4, DUMMY, 1, 0, S, 4, 3, AUX, 7 )}
\]

\[
A = \begin{bmatrix}
1.0 & 2.0 & 3.0 \\
4.0 & 5.0 & 6.0 \\
7.0 & 8.0 & 9.0 \\
10.0 & 11.0 & 12.0 \\
\end{bmatrix}
\]

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:

\[
\text{CALL DGESVF}( 1, A, 3, \text{DUMMY}, 1, 0, 5, 3, 3, \text{AUX}, 9 )
\]

\[
A = \begin{bmatrix}
2.0 & 1.0 & 1.0 \\
4.0 & 1.0 & 0.0 \\
-2.0 & 2.0 & 1.0 \\
\end{bmatrix}
\]

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 \( U^TB \) in preparation for solving the underdetermined system \( AX = B \), where:
• \( M \) is less than \( N \).
• \( \text{NAUX} \) is greater than or equal to \( 2N + \max(M, N, \text{NB}) = 9 \).
• 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:

\[
\text{CALL DGESVF}( 2, A, 3, B, 3, 1, S, 2, 3, \text{AUX}, 9 )
\]

\[
A = \begin{bmatrix}
1.0 & 2.0 & 2.0 \\
2.0 & 4.0 & 5.0 \\
\cdot & \cdot & \cdot \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
1.0 \\
4.0 \\
\cdot \\
\end{bmatrix}
\]

Output:

\[
A = \begin{bmatrix}
-0.304 & -0.894 & 0.328 \\
-0.608 & 0.447 & 0.656 \\
-0.733 & 0.000 & -0.680 \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
-4.061 \\
0.000 \\
\cdot \\
\end{bmatrix}
\]
Example 4

This example computes the singular values, \( s \), and matrices \( V \) and \( U^T B \) in preparation for solving the overdetermined system \( AX = B \), where:

- \( M \) is greater than \( N \).
- \( \text{NAUX} \) 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 \( U^T B \) is returned in \( B \).

Call Statement and Input:

```
CALL DGESVF( 2, A, 3, B, 3, 2, S, 3, 2, AUX, 7 )
```

\[
\begin{bmatrix}
1.0 & 4.0 \\
2.0 & 5.0 \\
3.0 & 6.0 \\
\end{bmatrix}
\]

\[
A = \begin{bmatrix}
1.0 & 4.0 \\
2.0 & 5.0 \\
3.0 & 6.0 \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
7.0 & 10.0 \\
8.0 & 11.0 \\
9.0 & 12.0 \\
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
0.922 & -0.386 \\
-0.386 & -0.922 \\
\end{bmatrix}
\]

\[
A = \begin{bmatrix}
0.922 & -0.386 \\
-0.386 & -0.922 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
-1.310 & -2.321 \\
-13.867 & -18.963 \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
-1.310 & -2.321 \\
-13.867 & -18.963 \\
\end{bmatrix}
\]

\[
X = (0.773, 9.508)
\]

Example 5

This example computes the singular values, \( s \), and matrices \( V \) and \( U^T B \) in preparation for solving the overdetermined system \( AX = B \). The singular values are sorted in descending order, and the columns of \( V \) and the rows of \( U^T B \) are swapped to correspond to the sorted singular values.

- \( M \) is greater than \( N \).
- \( \text{NAUX} \) 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 \( U^T B \) is returned in \( B \).

Call Statement and Input:

```
CALL DGESVF( 12, A, 3, B, 3, 2, S, 3, 2, AUX, 7 )
```

\[
\begin{bmatrix}
-0.714 \\
\end{bmatrix}
\]

\[
S = (7.342, 0.000, 0.305)
\]
\[ A = \begin{bmatrix} 1.0 & 4.0 \\ 2.0 & 5.0 \\ 3.0 & 6.0 \end{bmatrix} \]

\[ B = \begin{bmatrix} 7.0 & 10.0 \\ 8.0 & 11.0 \\ 9.0 & 12.0 \end{bmatrix} \]

Output:

\[ A = \begin{bmatrix} -0.386 & 0.922 \\ -0.922 & -0.386 \\ . & . \end{bmatrix} \]

\[ B = \begin{bmatrix} -13.867 & -18.963 \\ -1.310 & -2.321 \\ . & . \end{bmatrix} \]

\[ S = (9.508, 0.773) \]
SGESVS and DGESVS (Linear Least Squares Solution for a General Matrix Using the Singular Value Decomposition)

Purpose

These subroutines compute the minimal norm linear least squares solution of $AX = B$, where $A$ is a general matrix, using the singular value decomposition computed by SGESVF or DGESVF.

Table 187. Data Types

<table>
<thead>
<tr>
<th>V, UB, X, s, τ</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGESVS</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGESVS</td>
</tr>
</tbody>
</table>

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);

On Entry

$v$ is the orthogonal matrix $V$ of order $n$ in the singular value decomposition of matrix $A$. 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 187.

$ldv$

is the leading dimension of the array specified for $v$.

Specified as: an integer; $ldv > 0$ and $ldv \geq n$.

$ub$

is an $n$ by $nb$ matrix, containing $U^TB$. It is produced by a preceding call to SGESVF or DGESVF, where it corresponds to output argument $b$. On output, $U^TB$ 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 187.

$ldub$

is the leading dimension of the array specified for $ub$.

Specified as: an integer; $ldub > 0$ and $ldub \geq n$.

$nb$

is the number of columns in matrices $X$ and $U^TB$.

Specified as: an integer; $nb > 0$.

$s$

is the vector $s$ of length $n$, containing the singular values of matrix $A$. 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 187; $s_i \geq 0$.

$x$

See On Return.

$ldx$

is the leading dimension of the array specified for $x$. 

Chapter 10. Linear Algebraic Equations 927
Specified as: an integer; \( ldx > 0 \) and \( ldx \geq n \).

\( m \) is the number of rows in matrix \( A \).
Specified as: an 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: an integer; \( n \geq 0 \).

\( \tau \) is the error tolerance \( \tau \). Any singular values in vector \( s \) that are less than \( \tau \) are treated as zeros when computing matrix \( X \).
Specified as: a number of the data type indicated in Table 187 on page 927; \( \tau \geq 0 \). For more information on the values for \( \tau \), see “Notes .”

On Return

\( x \) is an \( n \) by \( nb \) matrix, containing the minimal norm linear least solutions of \( AX=B \). 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 187 on page 927.

Notes

1. \( V \), \( X \), \( s \), and \( U^TB \) can have no common elements; otherwise the results are unpredictable.
2. In problems involving experimental data, \( \tau \) should reflect the absolute accuracy of the matrix elements:

\[ \tau \geq \max(|\Delta_{ij}|) \]

where \( \Delta_{ij} \) are the errors in \( a_{ij} \). In problems where the matrix elements are known exactly or are only affected by roundoff errors:

\[ \tau \geq \varepsilon \left( \frac{\sqrt{mn}}{s_j} \right) \max(s_j) \quad \text{for} \quad j = (1, \ldots, n) \]

where:
\( \varepsilon \) 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 \[20 on page 1364], \[69 on page 1367], \[92 on page 1368\], and pages 134 to 151 in reference \[118 on page 1370\].

Function

The minimal norm linear least squares solution of \( AX=B \), 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:

\[ A = U\Sigma V^T \]

The linear least squares of solution \( X \), for \( AX=B \), is given by the following formula:

\[ X = V\Sigma^+U^TB \]
where:

\[ \Sigma + \] is the diagonal matrix with elements \( \sigma_j^+ \), where:

\[
\begin{align*}
\sigma_j^+ &= \frac{1.0}{\sigma_j} \quad \text{if } \sigma_j \geq \tau \text{ and } \sigma_j \neq 0 \\
\sigma_j^+ &= 0 \quad \text{for all other cases}
\end{align*}
\]

If \( m \) or \( n \) is equal to 0, no computation is performed. See references [20 on page 1364], [69 on page 1367], [92 on page 1368], and pages 134 to 151 in reference [118 on page 1370]. 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. \( \tau < 0 \)

**Examples**

**Example 1**

This example finds the linear least squares solution for the underdetermined system \( AX = B \), 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 \( U^T B \) is overwritten.

**Note:** This example corresponds to DGESVF **Example 3**

**Call Statement and Input:**

\[
\begin{align*}
V & \quad LDV & \quad UB & \quad LDUB & \quad NB & \quad S & \quad X & \quad LDX & \quad M & \quad N & \quad TAU \\
\end{align*}
\]

CALL DGESVS( V, 3, UB, 3, 1, S, X, 3, 2, 3, TAU )
\[
V = \begin{bmatrix}
-0.304 & -0.894 & 0.328 \\
-0.608 & 0.447 & 0.656 \\
-0.733 & 0.000 & -0.600
\end{bmatrix}
\]

\[
UB = \begin{bmatrix}
-4.061 \\
0.000 \\
-0.714
\end{bmatrix}
\]

\[
S = (7.342, 0.000, 0.305)
\]

\[\text{TAU} = 0.3993 \times 10^{-14}\]

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 \(AX = B\), using the singular value decomposition computed by DGESVF. Matrix \(A\) is:

\[
\begin{bmatrix}
1.0 & 4.0 \\
2.0 & 5.0 \\
3.0 & 6.0
\end{bmatrix}
\]

and where \(B\) is:

\[
\begin{bmatrix}
7.0 & 10.0 \\
8.0 & 11.0 \\
9.0 & 12.0
\end{bmatrix}
\]

On output, matrix \(U^TB\) is overwritten.

**Note:** This example corresponds to DGESVF Example 4

Call Statement:

```
CALL DGESVS( V, 3, UB, 3, 2, S, X, 2, 3, 2, TAU )
```

Input:

\[
V = \begin{bmatrix}
0.922 & -0.386 \\
-0.386 & -0.922 \\
\cdot & \cdot
\end{bmatrix}
\]

\[
UB = \begin{bmatrix}
-1.310 & -2.321 \\
-13.867 & -18.963 \\
\cdot & \cdot
\end{bmatrix}
\]

\[
S = (0.773, 9.508)
\]

\[\text{TAU} = 0.5171 \times 10^{-14}\]

Output:
\[
X = \begin{bmatrix}
-1.000 & -2.000 \\
2.000 & 3.000
\end{bmatrix}
\]
SGELLS and DGELLS (Linear Least Squares Solution for a General Matrix with Column Pivoting)

Purpose

These subroutines compute the minimal norm linear least squares solution of $AX \cong B$, using a QR decomposition with column pivoting.

Table 188. Data Types

<table>
<thead>
<tr>
<th>$A$, $B$, $X$, $rn$, $\tau$, $aux$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGELLS</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGELLS</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SGELLS</th>
<th>DGELLS (iopt, a, lda, b, ldb, x, ldx, rn, tau, m, n, nb, k, aux, naux)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>sgells</td>
<td>dgells (iopt, a, lda, b, ldb, x, ldx, rn, tau, m, n, nb, k, aux, naux);</td>
</tr>
</tbody>
</table>

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: an 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 188.

lda

is the leading dimension of the array specified for $a$.

Specified as: an integer; $lda > 0$ and $lda \geq m$.

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 188.

ldb

is the leading dimension of the array specified for $b$.

Specified as: an integer; $ldb > 0$ and $ldb \geq m$.

x

See On Return.

ldx

is the leading dimension of the array specified for $x$.

Specified as: an integer; $ldx > 0$ and $ldx \geq n$.

rn

See On Return.
tau

is the tolerance $\tau$, 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 188 on page 932; $\tau \geq 0$. For more information on how to select a value for $\tau$, see “Notes” on page 934.

$m$

is the number of rows in matrices $A$ and $B$.

Specified as: an integer; $m \geq 0$.

$n$

is the number of columns in matrix $A$ and the number of rows in matrix $X$.

Specified as: an integer; $n \geq 0$.

$nb$

is the number of columns in matrices $B$ and $X$ and the number of elements in vector $rr$.

Specified as: an 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 188 on page 932. 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: an 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:

For SGELLS

For SGELLS, it must have the following values:

For 32-bit integer arguments

$naux \geq 3n + \max(n, nb)$

For 64-bit integer arguments

$naux \geq 4n + \max(n, nb) + 3$

For DGELLS

For DGELLS, it must have the following values:

For 32-bit integer arguments

$naux \geq \lceil 2.5n \rceil + \max(n, nb)$

For 64-bit integer arguments

$naux \geq 3n + \max(n, nb)$

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 188 on page 932.

\( \text{rn} \) is the vector \( \text{rn} \) of length \( nb \), where:

- If \( iopt = 0 \) or \( k = 0 \), \( \text{rn} \) is not used in the computation.
- If \( iopt = 1 \), \( \text{rn} \) 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 188 on page 932.

\( k \) is the number of columns of matrix \( A \) used in the solution. Returned as: an integer; \( k = ( \text{the number of diagonal elements of matrix } R \text{ exceeding } \tau \text{ in magnitude}) \).

**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 \( m \), and the data area specified for \( aux \)
   - Matrices \( A \) and \( B \), vector \( m \), and the data area specified for \( aux \).
4. If the relative uncertainty in the matrix \( B \) is \( \rho \), then:

   \[
   \tau \; \geq \; \rho \| A \|_F
   \]

   See references [52 on page 1366], [73 on page 1367], and [92 on page 1368] for additional guidance on determining suitable values for \( \tau \).
5. When you specify \( iopt = 0 \), you must also specify a dummy argument for \( \text{rn} \). For more details, see Example 1.
6. You have the option of having the minimum required value for \( \text{naux} \) dynamically returned to your program. For details, see “Using Auxiliary Storage in ESSL” on page 51.

**Function**

The minimal norm linear least squares solution of \( AX=B \) 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 \( AX=B \). \( A \) is decomposed, using Householder transformations and column pivoting, into the following form:

\[
AP = QR
\]
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 $RX = Q^T B$ and reordering $X$ to correspond to the original columns of $A$.

If $k < n$, $R$ has the following form:

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix}$$

To find the minimal norm linear least squares solution, it is necessary to zero the submatrix $R_{12}$ using Householder transformations. See references [52 on page 1366], [73 on page 1367], and [92 on page 1368]. 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.

**Examples**

**Example 1**
This example solves the underdetermined system $AX\cong B$. On output, $A$ and $B$ are overwritten. DUMMY is used as a placeholder for argument $m$, which is not used in the computation.

**Call Statement and Input:**
```
CALL DGELLS( IOPT, A, LDA, B, LDB, X, LDX, RN, TAU, M, N, NB, K, AUX, NAUX )
```

$A = \begin{bmatrix} 1.0 & 2.0 & 2.0 \\ 2.0 & 4.0 & 5.0 \end{bmatrix}$

$B = \begin{bmatrix} 1.0 \\ 4.0 \end{bmatrix}$

$\tau = 0.0$

**Output:**

$X = \begin{bmatrix} -0.600 \\ -1.200 \\ 2.000 \end{bmatrix}$

$K = 2$

**Example 2**

This example solves the overdetermined system $AX\cong B$. On output, $A$ and $B$ are overwritten. DUMMY is used as a placeholder for argument $m$, which is not used in the computation.

**Call Statement and Input:**
```
CALL DGELLS( IOPT, A, LDA, B, LDB, X, LDX, RN, TAU, M, N, NB, K, AUX, NAUX )
```

$A = \begin{bmatrix} 1.0 & 4.0 \\ 2.0 & 5.0 \\ 3.0 & 6.0 \end{bmatrix}$

$B = \begin{bmatrix} 7.0 & 10.0 \\ 8.0 & 11.0 \\ 9.0 & 12.0 \end{bmatrix}$

$\tau = 0.0$

**Output:**

$X = \begin{bmatrix} -1.000 & -2.000 \\ 2.000 & 3.000 \end{bmatrix}$

$K = 2$

**Example 3**

This example solves the overdetermined system $AX\cong B$ and computes the Euclidean Norms of the residual vectors. On output, $A$ and $B$ are overwritten.

**Call Statement and Input:**
```
```
CALL DGELLS( 1, A, 3, B, 3, X, 2, RN, TAU, 3, 2, 2, K, AUX, 7 )

\[
A = \begin{bmatrix}
1.1 & -4.3 \\
2.0 & -5.0 \\
3.0 & -6.0
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
-7.0 & 10.0 \\
-8.0 & 11.0 \\
-9.0 & 12.0
\end{bmatrix}
\]

\[
TAU = 0.0
\]

Output:

\[
X = \begin{bmatrix}
0.543 & -1.360 \\
1.785 & -2.699
\end{bmatrix}
\]

\[
RN = \begin{bmatrix}
0.196 \\
0.275
\end{bmatrix}
\]

\[
K = 2
\]
Chapter 11. Eigensystem Analysis

The eigensystem analysis subroutines are described here.

Overview of the Eigensystem Analysis Subroutines

The eigensystem analysis subroutines provide solutions to the algebraic eigensystem analysis problem and the generalized eigensystem analysis problem. These subroutines correspond to the LAPACK routines described in reference [8 on page 1363].

Table 189. List of LAPACK Eigensystem Analysis Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGEV&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DGEV&lt;sup&gt;a&lt;/sup&gt;</td>
<td>SGEEV, DGEEV, CGEEV, ZGEEV, SGEEVX, DGEEVX, CGEEVX, and ZGEEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix)” on page 942</td>
</tr>
<tr>
<td>CGEV&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZGEV&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SGEEVX&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DGEEVX&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CGEEVX&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZGEEVX&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgeev&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACKE_dgeev&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgeev&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACKE_zgeev&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sggeevx&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACKE_dgeevx&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cggeevx&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACKE_zgeevx&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SSYEV&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DSYEV&lt;sup&gt;a&lt;/sup&gt;</td>
<td>SSYEV, DSYEV, CHEEV, ZHEEV, SSPEVX, DSPEVX, CHPEVX, ZHPEVX, SSYEVX, DSYEVX, CHEEVX, and ZHEEVX (Eigenvalues and, Optionally, the Eigenvectors of a Real Symmetric or Complex Hermitian Matrix)” on page 959</td>
</tr>
<tr>
<td>CHEEV&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZHEEV&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SSPEVX&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DSPEVX&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CHPEVX&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZHPEVX&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SSYEVX&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DSYEVX&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CHEEVEVX&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZHEEVEVX&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssyev&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACKE_dsyev&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cheev&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACKE_zheev&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sspevx&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACKE_dspevx&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chpevx&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACKE_zhpevx&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssyevx&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACKE_dsyevx&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cheevx&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACKE_zheevx&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SSPEVD&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DSPEVD&lt;sup&gt;a&lt;/sup&gt;</td>
<td>SSPEVD, DSPEVD, CHPEVD, ZHPEVD, SSYEVX, DSYEVX, CHEEVX, and ZHEEVD (Eigenvalues and, Optionally the Eigenvectors of a Real Symmetric or Complex Hermitian Matrix Using a Divide-and-Conquer Algorithm)” on page 978</td>
</tr>
<tr>
<td>CHPEVD&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZHPEVD&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SSYEVD&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DSYEVD&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CHEEVD&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZHEEVD&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sspevd&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACKE_dspevd&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chpevd&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACKE_zhpevd&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssyevd&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACKE_dsyevd&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cheevd&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACKE_zheevd&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SGGEV&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DGGEV&lt;sup&gt;a&lt;/sup&gt;</td>
<td>SGGEV, DGGEV, CGGEV, ZGGEV, SGEEVX, DGEEVX, CGEEVX, and ZGEEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix Generalized Eigenproblem)” on page 991</td>
</tr>
<tr>
<td>CGGGEV&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZGGGEV&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SGGEEVX&lt;sup&gt;a&lt;/sup&gt;</td>
<td>DGGEEVX&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CGGEEVX&lt;sup&gt;a&lt;/sup&gt;</td>
<td>ZGGGEVX&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgggev&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACKE_dgggev&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgggev&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACKE_zgggev&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgggev&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACKE_dgggev&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgggev&lt;sup&gt;a&lt;/sup&gt;</td>
<td>LAPACKE_zgggev&lt;sup&gt;a&lt;/sup&gt;</td>
<td></td>
</tr>
</tbody>
</table>
Table 189. List of LAPACK Eigensystem Analysis Subroutines (continued)

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSPGVX</td>
<td>DSPGVX</td>
<td>&quot;SSPGVX, DSPGVX, CHPGVX, ZHPGVX, SSYGVX, DSYGVX, CHEGVX, and ZHEGVX (Eigenvalues and, Optionally, the Eigenvectors of a Positive Definite Real Symmetric or Complex Hermitian Generalized Eigenproblem)&quot; on page 1008</td>
</tr>
<tr>
<td>CHPGVX</td>
<td>ZHPGVX</td>
<td></td>
</tr>
<tr>
<td>SYMGVX</td>
<td>DSYGYVX</td>
<td></td>
</tr>
<tr>
<td>CHEVGX</td>
<td>ZHEGYVX</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spgvxy</td>
<td>LAPACKE_dspgvxy</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chpgvxy</td>
<td>LAPACKE_zhpgvxy</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssygvy</td>
<td>LAPACKE_dsygyvxy</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chegvxy</td>
<td>LAPACKE_zhegyvxy</td>
<td></td>
</tr>
</tbody>
</table>

A LAPACK

Performance and Accuracy Considerations

1. The short precision subroutines provide increased accuracy by accumulating intermediate results in long precision when the AltiVec or VSX unit is not used. Occasionally, for performance reasons, these intermediate results are stored.

2. 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 64.
Eigensystem Analysis Subroutines

This contains the eigensystem analysis subroutine descriptions.
SGEEV, DGEEV, CGEEV, ZGEEV, SGEEVX, DGEEVX, CGEEVX, and ZGEEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix)

**Purpose**

- SGEEV, DGEEV, CGEEV, and ZGEEV compute the eigenvalues and, optionally, right eigenvectors and/or left eigenvectors of a general matrix.
- SGEEVX, DGEEVX, CGEEVX, and ZGEEVX compute the eigenvalues and, optionally, right eigenvectors, left eigenvectors, reciprocal condition numbers for eigenvalues, and reciprocal condition numbers for right eigenvectors of a general matrix.

For a right eigenvector $v$ of $A$:

$$Av = \lambda v$$

For a left eigenvector $u$ of $A$:

$$u^H A = \lambda u^H$$

The computed eigenvectors are normalized to have the Euclidean norm equal to one and the largest component real.

**Table 190. Data Types**

<table>
<thead>
<tr>
<th>$A$, vl, vr, work, wr, wi, w</th>
<th>scale, abnrn, rconde, rcondv, rwork</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SGEEV$^a$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SGEEVX$^a$</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DGEEV$^a$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DGEEVX$^a$</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>CGEEV$^a$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CGEEVX$^a$</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>ZGEEV$^a$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZGEEVX$^a$</td>
</tr>
</tbody>
</table>

$^a$ LAPACK

**Syntax**

**Fortran**

```fortran
CALL SGEEV | DGEEV (jobvl, jobvr, n, a, lda, vr, wi, vl, ldvl, vr, ldvr, work, lwork, info)
CALL CGEEV | ZGEEV (jobvl, jobvr, n, a, lda, vr, wi, vl, ldvl, vr, ldvr, work, lwork, rwork, info)
CALL SGEEVX | DGEEVX (balanc, jobvl, jobvr, sense, n, a, lda, vr, wi, vl, ldvl, vr, ldvr, ilo, ihi, scale, abnrn, rconde, rcondv, work, lwork, info)
CALL CGEEVX | ZGEEVX (balanc, jobvl, jobvr, sense, n, a, lda, vr, wi, vl, ldvl, vr, ldvr, ilo, ihi, scale, abnrn, rconde, rcondv, work, lwork, rwork, info)
```
### C and C++

```
sgeev | dgeev (jobvl, jobvr, n, a, lda, wr, vi, vl, ldvl, vr, ldvr, work, lwork, info);

cgeev | zgeev (jobvl, jobvr, n, a, lda, w, vl, ldvl, vr, ldvr, work, lwork, rwork, info);

sgeevx | dgeevx (balanc, jobvl, jobvr, sense, n, a, lda, wr, vi, vl, ldvl, vr, ldvr, ilo, ihi, scale, abnrn, rconde, rconde, work, lwork, rwork, info);

cgeevx | zgeevx (balanc, jobvl, jobvr, sense, n, a, lda, w, vl, ldvl, vr, ldvr, ilo, ihi, scale, abnrn, rconde, rconde, work, lwork, rwork, info);
```

### LAPACK

```
info = LAPACKE_sgeev | LAPACKE_dgeev (matrix_layout, jobvl, jobvr, n, a, lda, wr, vi, vl, ldvl, vr, ldvr);

info = LAPACKE_cgeev | LAPACKE_zgeev (matrix_layout, jobvl, jobvr, n, a, lda, w, vl, ldvl, vr, ldvr);

info = LAPACKE_sgeevx | LAPACKE_dgeevx (matrix_layout, balanc, jobvl, jobvr, sense, n, a, lda, wr, vi, vl, ldvl, vr, ldvr, ilo, ihi, scale, abnrn, rconde, rconde);

info = LAPACKE_cgeevx | LAPACKE_zgeevx (matrix_layout, balanc, jobvl, jobvr, sense, n, a, lda, w, vl, ldvl, vr, ldvr, ilo, ihi, scale, abnrn, rconde, rconde);
```

---

### On Entry

**matrix_layout**

- Indicates whether the input and output matrices are stored in row major order or column major order, where:
  - If `matrix_layout = LAPACK_ROW_MAJOR`, the matrices are stored in row major order.
  - If `matrix_layout = LAPACK_COL_MAJOR`, the matrices are stored in column major order.

  Specified as: an integer. It must be `LAPACK_ROW_MAJOR` or `LAPACK_COL_MAJOR`

**balanc**

- Indicates whether or not to scale `A` diagonally and whether or not to permute its rows and columns to improve the conditioning of its eigenvalues, where `balanc` can have any of the following values:
  - **N**: Neither diagonally scale nor permute `A`.
  - **P**: Permute `A`, but do not diagonally scale it.
  - **S**: Diagonally scale `A`, but do not permute it.
  - **B**: Both diagonally scale and permute `A`.

When diagonal scaling is specified, the subroutine replaces `A` with `DAD^{-1}` where `D` is a diagonal matrix chosen to make the rows and columns of `A` more equal in norm and the condition numbers of its eigenvalues and eigenvectors smaller.

When permuting is specified, the subroutine makes `A` more nearly upper triangular.

The computed reciprocal condition numbers correspond to the balanced matrix. In exact arithmetic, permuting rows and columns does not change the condition numbers, but diagonal scaling does change the condition numbers.

Specified as: a single character. It must be 'N', 'P', 'S', or 'B'.

---

Chapter 11. Eigensystem Analysis  943
**jobvl**
indicates whether or not to compute the left eigenvectors of $A$, where $jobvl$ can have either of the following values:

- **N**: Do not compute the left eigenvectors of $A$.
- **V**: Compute the left eigenvectors of $A$.

**Note:** If $sense = 'E$ or 'B', $jobvl$ must = 'V'.
Specified as: a single character. It must be 'N' or 'V'.

**jobvr**
indicates whether or not to compute the right eigenvectors of $A$, where $jobvr$ can have either of the following values:

- **N**: Do not compute the right eigenvectors of $A$.
- **V**: Compute the right eigenvectors of $A$.

**Note:** If $sense = 'E$ or 'B', $jobvr$ must = 'V'.
Specified as: a single character. It must be 'N' or 'V'.

**sense**
indicates which reciprocal numbers to compute (if any), where $sense$ can have any of the following values:

- **N**: Do not compute reciprocal condition numbers.
- **E**: Compute reciprocal condition numbers for eigenvalues only.
- **V**: Compute reciprocal condition numbers for right eigenvectors only.
- **B**: Compute reciprocal condition numbers for eigenvalues and right eigenvectors.

**Note:** If $sense = 'E$ or 'B', both $jobvl$ and $jobvr$ must equal 'V' (so that both left and right eigenvectors are also computed).
Specified as: a single character. It must be 'N', 'E', 'V', or 'B'.

**n**
is the order of the general matrix $A$.
Specified as: an integer; $n \geq 0$.

**a**
is the general matrix $A$ of order $n$.
Specified as: an $lda$ by (at least) $n$ array, containing numbers of the data type indicated in Table 190 on page 942.

**lda**
is the leading dimension of the array specified for $a$.
Specified as: an integer, where $lda > 0$ and $lda \geq n$.

**wr**
See **On Return**

**wi**
See **On Return**

**w**
See **On Return**

**ldvl**
is the leading dimension of the array specified for $vl$.
Specified as: an integer; $ldvl > 0$; if $jobvl = 'V', ldvl \geq n$. 

---

944 ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
ldvr

is the leading dimension of the array specified for vr.
Specified as: an integer; ldvr > 0; if jobvr = 'V', ldvr ≥ n.

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 190 on page 942.

lwork

is the number of elements in array WORK.
Specified as an integer, where:
- If lwork = 0, the subroutine dynamically allocates the workspace needed for use during this computation. The dynamically allocated workspace will be freed prior to returning control to the calling program.
- If lwork = -1, a workspace query is assumed. The subroutine only calculates the optimal size of the WORK array and returns this value as the first entry of the WORK array.
Otherwise:
- For SGEEV and DGEEV:
  - If jobvl = 'N' or jobvr = 'N', lwork ≥ max(1, 3n).
  - If jobvl = 'V' or jobvr = 'V', lwork ≥ max(1, 4n).
- For CGEEV and ZGEEV, lwork ≥ max(1, 2n).
- For SGEEVX and DGEEVX:
  - If sense = 'N' or 'E':
    - If jobvl = 'N' and jobvr = 'N', lwork ≥ max(1, 2n).
    - If jobvl = 'V' or jobvr = 'V', lwork ≥ max(1, 3n).
  - If sense = 'V' or 'B', lwork ≥ max(1, n(n + 6)).
- For CGEEVX and ZGEEVX:
  - If sense = 'N' or 'E', lwork ≥ max(1, 2n).
  - If sense = 'V' or 'B', lwork ≥ max(1, n^2 + 6n^2).

Note: These formulas represent the minimum workspace required. For best performance, specify either lwork = -1 (to obtain the optimal size to use) or lwork = 0 (to direct the subroutine to dynamically allocate the workspace).

rwork

is a storage work area of size 2n.
Specified as: an area of storage containing numbers of the data type indicated in Table 190 on page 942.

iwork

is a storage work area of size max(1, 2n-2).
If sense = 'N' or 'E', iwork is not referenced by the subroutine.
Specified as: an integer array.

On Return

a is the updated general matrix A of order n. On output, A is overwritten; that is, the original input is not preserved. If jobvl = 'V' or jobvr = 'V', A contains the Schur form of the balanced matrix.
Returned as: an _lda_ by (at least) _n_ array, containing numbers of the data type indicated in Table 190 on page 942

**wr** contains the real part of the computed eigenvalues. Returned as: a one-dimensional array of (at least) length _n_, containing numbers of the data type indicated in Table 190 on page 942

**wi** contains the imaginary part of the computed eigenvalues. Returned as: a one-dimensional array of (at least) length _n_, containing numbers of the data type indicated in Table 190 on page 942

**w** contains the computed eigenvalues. Returned as: a one-dimensional array of (at least) length _n_, containing numbers of the data type indicated in Table 190 on page 942

**vl** contains the left eigenvectors.

- If _jobvl_ = 'V', the left eigenvectors are stored one after another in the columns of _vl_, in the same order as their eigenvalues.
  - For SGEEV, DGEEV, SGEEVX and DGEEVX:
    - If the _jth_ eigenvalue is real, then the _jth_ column of _vl_ contains its eigenvector.
    - If the _jth_ and (_j+1)st eigenvalues form a complex conjugate pair, then the _jth_ and (_j+1)st columns of _vl_ contain the real and imaginary parts of the eigenvector corresponding to the _jth_ eigenvalue. The conjugate of this eigenvector is the eigenvector for the (_j+1)st eigenvalue.
- If _jobvl_ = 'N', _vl_ is not referenced.

Returned as: an array of size (ldvl, _n_) containing numbers of the data type indicated in Table 190 on page 942

**vr** contains the right eigenvectors.

- If _jobvr_ = 'V', the left eigenvectors are stored one after another in the columns of _vr_, in the same order as their eigenvalues.
  - For SGEEV, DGEEV, SGEEVX and DGEEVX:
    - If the _jth_ eigenvalue is real, then the _jth_ column of _vr_ contains its eigenvector.
    - If the _jth_ and (_j+1)st eigenvalues form a complex conjugate pair, then the _jth_ and (_j+1)st columns of _vr_ contain the real and imaginary parts of the eigenvector corresponding to the _jth_ eigenvalue. The conjugate of this eigenvector is the eigenvector for the (_j+1)st eigenvalue.
- If _jobvr_ = 'N', _vr_ is not referenced.

Returned as: an array of size (ldvr, _n_) containing numbers of the data type indicated in Table 190 on page 942

**ilo** has the following meaning:

If _balanc_ = 'N', _ilo_ = 1.

Otherwise, the value of _ilo_ is determined when _A_ is balanced.

The balanced _a_ _ij_ = 0 if _i_ > _j_ and _j_ = 1, ..., (_ilo-1) or _i_ = (_ihi+1), ..., _n_.

Returned as: an integer; 1 ≤ _ilo_ ≤ _n_.

**ihi** has the following meaning:
If \( balanc = 'N', ihi = n. \)
Otherwise, the value of \( ihi \) is determined when \( A \) is balanced.
The balanced \( a_{ij} = 0 \) if \( i > j \) and \( j = 1, ..., (ilo-1) \) or \( i = (ihi+1), ..., n. \)
Returned as: an integer; \( 1 \leq ihi \leq n. \)

\( scale \)
contains the details of the permutations and scaling factors applied when balancing \( A. \)
If \( p_j \) is the index of the row and column interchanged with row and column \( j, \)
and \( d_j \) is the scaling factor applied to row and column \( j, \) then:
• \( scale_j = p_j, \) for \( j = 1, ..., (ilo) \)
• \( scale_j = d_j, \) for \( j = ilo, ..., ihi \)
• \( scale_j = p_j, \) for \( j = (ihi+1), ..., n \)
Returned as: a one-dimensional array of (at least) length \( n \) containing numbers of the data type indicated in Table 190 on page 942.

\( abnrm \)
is the one-norm of the balanced matrix (the maximum of the sum of absolute values of elements of any column).
Returned as: a number of the data type indicated in Table 190 on page 942; \( abnrm \geq 0. \)

\( rconde \)
contains the computed reciprocal condition numbers of the eigenvalues, where \( rconde_j \) is the reciprocal condition number of the \( j \)th eigenvalue.
Returned as: an array of dimension \( n \) containing numbers of the data type indicated in Table 190 on page 942.

\( rcondv \)
contains the computed reciprocal condition numbers of the eigenvectors, where \( rcondv_j \) is the reciprocal condition number of the \( j \)th right eigenvector.
Returned as: an array of dimension \( n \) containing numbers of the data type indicated in Table 190 on page 942.

\( work \)
is the work area used by this subroutine if \( lwork \neq 0, \) where:
If \( lwork \neq 0 \) and \( lwork \neq -1, \) its size is (at least) of length \( lwork. \)
If \( lwork = -1, \) its size is (at least) of length \( 1. \)
Returned as: an area of storage, where:
If \( lwork \geq 1 \) or \( lwork = -1, \) then \( work \) is set to the optimal \( lwork \) value and contains numbers of the data type indicated in Table 190 on page 942.
Except for \( work_k, \) the contents of \( work \) are overwritten on return.

\( rwork \)
is a storage work area of size \( 2n. \)
Returned as: an area of storage containing numbers of the data type indicated in Table 190 on page 942.

\( iwork \)
is a storage work area of size \( 2n-2. \)
If \( sense = 'N' \) or \( 'E', iwork \) is not referenced by the subroutine.
Returned as: an integer array.

info has the following meaning:

If info = 0, the subroutine completed successfully.

If info > 0:

- For SGEEV, DGEEV, CGEEV, ZGEEV, LAPACKE_sgeev, LAPACKE_dgeev, LAPACKE_cgeev, and LAPACKE_zgeev, if info > 0, the QR algorithm failed to compute all the eigenvalues, and no eigenvectors were computed; elements (i+1):n of the eigenvalue arrays contain eigenvalues that have converged.

- For SGEEVX, DGEEVX, CGEEVX, ZGEEVX, LAPACKE_sgeevx, LAPACKE_dgeevx, LAPACKE_cgeevx, and LAPACKE_zgeevx, if info > 0, the QR algorithm failed to compute all the eigenvalues, and no eigenvectors or reciprocal condition numbers were computed; elements 1:(ilo-1) and (i+1):n of the eigenvalue arrays contain eigenvalues that have converged.

Returned as:

- For SGEEV, DGEEV, CGEEV, ZGEEV, SGEEVX, DGEEVX, CGEEVX, and ZGEEVX returned as: an integer; info ≥ 0.

- For LAPACKE_sgeev, LAPACKE_dgeev, LAPACKE_cgeev, LAPACKE_zgeev, LAPACKE_sgeevx, LAPACKE_dgeevx, LAPACKE_cgeevx, and LAPACKE_zgeevx, returned as an integer function value; info ≥ 0.

Notes

1. A, wr, wi, vl, vr, work, rwork, and iwork must have no common elements; otherwise, results are unpredictable.

2. When you specify jobvl = 'N', you must specify a dummy argument for vl.

3. When you specify jobvr = 'N', you must specify a dummy argument for vr.

4. When you specify sense = 'N', you must specify a dummy argument for rconde.

5. When you specify sense = 'N' or 'E', you must specify dummy arguments for rcondv and iwork.

6. In your C program, the ilo, ihi, abnrm, info arguments must be passed by reference.

7. These subroutines accept lowercase letters for the balanc, jobvl, jobvr, and sense arguments.

8. The vectors and matrices used in the computation must have no common elements; otherwise, results are unpredictable.

9. For best performance, specify lwork = 0.

Function

These subroutines compute the following for a general matrix A:

- SGEEV, DGEEV, CGEEV, ZGEEV, SGEEVX, DGEEVX, CGEEVX, and ZGEEVX compute:
  - eigenvalues
  - optionally, the right eigenvectors, left eigenvectors, or both

- SGEEVX, DGEEVX, CGEEVX, and ZGEEVX also compute:
  - optionally, the reciprocal condition numbers for the eigenvalues
  - optionally, the reciprocal condition numbers for the right eigenvectors
Computing eigenvalues only
The eigenvalues (only) of general matrix $A$ are computed as follows:
1. If necessary, scale the general matrix $A$.
2. Balance the general matrix $A$.
3. Reduce the balanced matrix to an upper Hessenberg matrix using the following types of transformations:
   - **SGEEV, DGEEV, SGEEVX and DGEEVX**
     - Orthogonal similarity transformations
   - **CGEEV, DGEEV, CGEEVX and ZGEEVX**
     - Unitary similarity transformations
4. Compute the eigenvalues of the upper Hessenberg matrix using the multi-shift QR algorithm or the implicit double-shift QR algorithm.
5. If specified for SGEEVX, DGEEVX, CGEEVX, and ZGEEVX, compute reciprocal condition numbers.
6. If necessary, undo scaling.

Computing eigenvalues and right eigenvectors or left eigenvectors or both
The eigenvalues and right eigenvectors or left eigenvectors, or both, of general matrix $A$ are computed as follows:
1. If necessary, scale the general matrix $A$.
2. Balance the general matrix $A$.
3. Reduce the balanced matrix to an upper Hessenberg matrix using the following types of transformations:
   - **SGEEV, DGEEV, SGEEVX and DGEEVX**
     - Orthogonal similarity transformations
   - **CGEEV, ZGEEV, CGEEVX and ZGEEVX**
     - Unitary similarity transformations
4. Accumulate the transformations.
5. Compute the eigenvalues of the upper Hessenberg matrix, and the appropriate eigenvectors of the corresponding balanced matrix, using the multi-shift QR algorithm or the implicit double-shift QR algorithm.
6. If specified, compute reciprocal condition numbers.
7. Undo balancing the eigenvectors; normalize the eigenvectors; and make the largest component real.
8. If necessary, undo scaling.

If $n = 0$, no computation is performed and the subroutine returns after doing some parameter checking.

For more information, see references [14 on page 1364], [15 on page 1364], [65 on page 1367], [66 on page 1367], and [74 on page 1367].

Error conditions

Resource Errors
$Iwork = 0$, and unable to allocate work area.

Computational Errors
1. Eigenvalue $(i)$ failed to converge.
   - For SGEEV, DGEEV, CGEEV and ZGEEV, elements $(i+1):n$ of $wr$ and $wi$
     contain eigenvalues that have converged. No eigenvectors have been computed.
   - For SGEEVX, DGEEVX, CGEEVX, and ZGEEVX, elements $1:(ilo-1)$ and $(i+1):n$
     contain eigenvalues that have converged. No eigenvectors or condition numbers have been computed.
• 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 2153 is set to be unlimited in the ESSL error option table.

2. The subroutine computed the eigenvalues using multiple algorithms.
   • Performance may be degraded.
   • 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 2613 is set to be unlimited in the ESSL error option table.

Input-Argument Errors
1. $balanc \neq 'N$', 'S', 'P', or 'B'
2. $jobvl \neq 'N'$, or 'V'
3. $jobvr \neq 'N$', or 'V'
4. $sense \neq 'N$', 'E', 'V', or 'B'
5. $(sense = 'E$ or $sense = 'B')$ and $(jobvl \neq 'V'$ or $jobvr \neq 'V')$
6. $n < 0$
7. $lda \leq 0$
8. $n > lda$
9. $ldvl \leq 0$
10. $ldvr \leq 0$
11. $jobvl \neq 'V'$ and $ldvl < n$
12. $jobvr \neq 'V'$ and $ldvr < n$
13. For SGEEV and DGEEV, if $lwork \neq 0$:
    • If $jobvl = 'N'$ and $jobvr = 'N'$, $lwork \leq \max(1, 3n)$.
    • If $jobvl = 'V'$ or $jobvr = 'V'$, $lwork < 4n$.
14. For CGEEV and ZGEEV, if $lwork \neq 0$ and $lwork < \max(1,2n)$.
15. $lwork \neq 0$ and $lwork \neq -1$ and $lwork <$ the minimum required value.
16. The size of a work array is greater than 2147483647 when 32-bit integers are used.

Examples

Example 1

This example shows how to find the eigenvalues only of a long-precision real general matrix $A$ of order 4, where:
• LDVL and LDVR are set to 1 to avoid an error condition.
• DUMMY1 is a placeholder for VL. VL is not used.
• DUMMY2 is a placeholder for VR. VR is not used.
• DUMMY3 is a placeholder for RCONDE. RCONDE is not used.
• DUMMY4 is a placeholder for RCONDV. RCONDV is not used.
• IDUMMY is a placeholder for IWORK. IWORK is not used.

Note:
1. This matrix is used in Example 5.5 in referenced text [74 on page 1367].
2. Because $lwork = 0$, the subroutine dynamically allocates WORK.
3. On output, \( A \) has been overwritten.

Call Statement and Input:

\[
\text{JOBVL, JOBVR, N, LDA, WR, WI, VL, LDVL, VR, LDVR, WORK, LWORK, INFO}
\]

CALL DGEEV( 'N', 'N', 4, 4, WR, WI, DUMMY1, 1, DUMMY2, 1, WORK, 0, INFO )

\[
\text{–or–}
\]

CALL DGEEVX( 'N', 'N', 'N', 'N', 4, 4, WR, WI, DUMMY1, 1, DUMMY2, 1, ILO, IHI, SCALE, ABNRM, RCONDE, RCONDV, WORK, LWORK, IWORK, INFO )

\[
A = \begin{bmatrix}
-2.0 & 2.0 & 2.0 & 2.0 \\
-3.0 & 3.0 & 2.0 & 2.0 \\
-2.0 & 0.0 & 4.0 & 2.0 \\
-1.0 & 0.0 & 0.0 & 5.0
\end{bmatrix}
\]

Output:

\[
\text{WR} = \begin{bmatrix}
1.000000 \\
2.000000 \\
3.000000 \\
4.000000
\end{bmatrix}
\]

\[
\text{WI} = \begin{bmatrix}
0.000000 \\
0.000000 \\
0.000000 \\
0.000000
\end{bmatrix}
\]

\[
\text{SCALE} = \begin{bmatrix}
1.000000 \\
1.000000 \\
1.000000 \\
1.000000
\end{bmatrix}
\]

\[
\text{ILO} = 1 \\
\text{IHI} = 4 \\
\text{ABNRM} = 11.0 \\
\text{INFO} = 0
\]

**Example 2**

For a long precision real general matrix \( A \) of order 4, this example shows the following, depending on the subroutine you are using:

- For DGEEV, this example shows how to find the eigenvalues and left and right eigenvectors of a general matrix \( A \).
- For DGEEVX, this example shows the eigenvalues, left and right eigenvectors, and reciprocal condition numbers for the eigenvalues and right eigenvectors of a balanced general matrix \( A \) of order 4.

**Note:**

1. This matrix is used in Example 5.5 in referenced text [74 on page 1367].
2. IWORK is an integer work array of size 6.
3. On output, \( A \) has been overwritten by the Schur form of the balanced matrix.

Call Statement and Input:
CALL DGEEV( 'V', 'V', 4, A, 4, WR, WI, VL, 1, VR, 1, WORK, 0, INFO )

–or–
CALL DGEEVX( 'B', 'V', 'V', 'B', 4, A, 4, WR, WI, VL, 4, VR, 4, ILO, IHI, SCALE, ABNRM, RCONDE, RCONDV, WORK, 0, IWORK, INFO )

\[
A = \begin{bmatrix}
-2.0 & 2.0 & 2.0 & 2.0 \\
-3.0 & 3.0 & 2.0 & 2.0 \\
-2.0 & 0.0 & 4.0 & 2.0 \\
-1.0 & 0.0 & 0.0 & 5.0
\end{bmatrix}
\]

Output:

• For DGEEV:

\[
A = \begin{bmatrix}
1.000000 & -6.949732 & 2.539925 & -0.707107 \\
0.000000 & 2.000000 & 1.060660 & -0.717805 \\
0.000000 & 0.000000 & 3.000000 & -0.780869 \\
0.000000 & 0.000000 & 0.000000 & 4.000000
\end{bmatrix}
\]

• For DGEEVX:

\[
A = \begin{bmatrix}
1.000000 & -6.949732 & -1.320184 & 0.103510 \\
0.000000 & 2.000000 & -2.415229 & 1.002262 \\
0.000000 & 0.000000 & 3.000000 & -0.780869 \\
0.000000 & 0.000000 & 0.000000 & 4.000000
\end{bmatrix}
\]

\[
WR = \begin{bmatrix}
1.000000 \\
2.000000 \\
3.000000 \\
4.000000
\end{bmatrix}
\]

\[
WI = \begin{bmatrix}
0.000000 \\
0.000000 \\
0.000000 \\
0.000000
\end{bmatrix}
\]

\[
VL = \begin{bmatrix}
-0.707107 & -0.408248 & 0.000000 & 0.000000 \\
0.707107 & 0.816497 & 0.408248 & 0.000000 \\
0.000000 & -0.408248 & -0.816497 & -0.447214 \\
0.000000 & 0.000000 & 0.408248 & 0.894427
\end{bmatrix}
\]

\[
VR = \begin{bmatrix}
-0.730297 & 0.625543 & -0.554700 & 0.500000 \\
-0.547723 & 0.625543 & -0.554700 & 0.500000 \\
-0.365148 & 0.417029 & -0.554700 & 0.500000 \\
-0.182574 & 0.208514 & -0.277350 & 0.500000
\end{bmatrix}
\]

\[
RCONDE = \begin{bmatrix}
0.087287 \\
0.053722 \\
0.096561 \\
0.282843
\end{bmatrix}
\]
Example 3

This example shows how to find the eigenvalues, left and right eigenvectors, and reciprocal condition numbers of a balanced long-precision real general matrix \( A \) of order 3.

Note:
1. This matrix is used in Example 5.4 in referenced text [74 on page 1367].
2. \( IWORK \) is an integer work array of size 4.
3. On output, \( A \) has been overwritten by the Schur form of the balanced matrix.

Call Statement and Input:

\[
\begin{align*}
\text{CALL DGEEVX('B', 'V', 'V', 'B', 3, A, 3, WR, WI, VL, VR, ILO, IHI,}
\end{align*}
\]

\[
\begin{align*}
\text{SCALE, ABNRM, RCONDE, RCONDV, WORK, LWORK, IWORK, INFO )}
\end{align*}
\]

\[
\begin{align*}
A = \begin{bmatrix}
8.0 & -1.0 & -5.0 \\
-4.0 & 4.0 & -2.0 \\
18.0 & -5.0 & -7.0
\end{bmatrix}
\end{align*}
\]

Output:

\[
\begin{align*}
\text{A} = \begin{bmatrix}
2.000000 & -6.928203 & -13.435029 \\
2.309401 & 2.000000 & -10.206207 \\
0.000000 & 0.000000 & 1.000000
\end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
\text{WR} = \begin{bmatrix}
2.000000 \\
2.000000 \\
1.000000
\end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
\text{WI} = \begin{bmatrix}
4.000000 \\
-4.000000 \\
0.000000
\end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
\text{VL} = \begin{bmatrix}
-0.877058 & 0.000000 & -0.816497 \\
0.263117 & -0.087706 & 0.408248 \\
0.350823 & 0.175412 & 0.408248
\end{bmatrix}
\end{align*}
\]
Example 4

This example shows how to find the eigenvalues and right eigenvectors of a long-precision complex general matrix \( A \) of order 4, where:

- \( \text{LDVL} \) is set to 1 to avoid an error condition.
- \( \text{DUMMY1} \) is a placeholder for \( \text{VL} \). \( \text{VL} \) is not used.
- \( \text{DUMMY2} \) is a placeholder for \( \text{RCONDE} \). \( \text{RCONDE} \) is not used.
- \( \text{DUMMY3} \) is a placeholder for \( \text{RCONDV} \). \( \text{RCONDV} \) is not used.

**Note:**
1. This matrix is used in Example 6.5 in referenced text [74 on page 1367].
2. On output, \( A \) has been overwritten by the Schur form of the balanced matrix.
3. \( \text{RWORK} \) is a real array of length 8.

**Call Statement and Input:**

```fortran
CALL ZGEEV( 'N', 'V', 4, A, 4, W, DUMMY1, 1, VR, 4, WORK, 0, RWORK, INFO )
```

- or -

```fortran
CALL ZGEEVX( 'N', 'N', 'V', 'V', 4, A, 4, W, DUMMY1, 1, VR, 4, ILO, IHI, SCALE, ABNRM, RCONDE, RCONDV, WORK, 0, RWORK, INFO )
```

**A** =

\[
\begin{bmatrix}
0.316228 & -0.316228 & 0.408248 \\
0.632456 & 0.000000 & 0.816497 \\
0.000000 & -0.632456 & 0.408248 \\
0.301511 & 0.301511 & 0.192450 \\
1.671856 & 1.671856 & 1.174058 \\
0.500000 & 1.000000 & 0.500000
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.301511 \\
0.301511 \\
0.192450 \\
1.671856 \\
1.671856 \\
1.174058 \\
0.500000 \\
1.000000 \\
1.000000
\end{bmatrix}
\]

Output:

- For ZGEEV:

\[
\begin{bmatrix}
5.0 & 9.0 \\
3.0 & 3.0 \\
2.0 & 2.0 \\
1.0 & 1.0
\end{bmatrix}
\]
Example 5

For a long precision complex general matrix $A$ of order 4, this example shows the following, depending on the subroutine you are using:

- For ZGEEV, this example shows how to find the eigenvalues and left and right eigenvectors of a complex general matrix $A$.
- For ZGEEVX, this example shows the eigenvalues, left and right eigenvectors, and reciprocal condition numbers for the eigenvalues and right eigenvectors of a balanced complex general matrix $A$ of order 4.

Note:

1. This matrix is used in Example 6.5 in referenced text [74 on page 1367].
2. RWORK is a real array of length 8.
3. On output, $A$ has been overwritten by the Schur form of the balanced matrix.

Call Statement and Input:

```fortran
JOBVL JOBVR N A LDA W VL LDVL VR LDVR WORK LWORK RWORK INFO
CALL ZGEEV( 'V', 'V', 4, A, 4, W, DUMMY1, 1, VR, 4, WORK, 0, RWORK, INFO )
```

-or-

```fortran
BALANC JOBVL JOBVR SENSE N A LDA W VL LDVL VR LDVR ILO IHI
CALL ZGEEVX( 'P', 'V', 'V', 'B', 4, A, 4, W, VL, 4, VR, 4, ILO, IHI,
SCALE ABNRM RCONDE RCONDV WORK LWORK RWORK INFO
SCALE, ABNRM, RCONDE, RCONDV, WORK, 0, RWORK, INFO )
```
For ZGEEV:

\[ A = \begin{bmatrix}
(5.0, 9.0) & (5.0, 5.0) & (-6.0, -6.0) & (-7.0, -7.0) \\
(3.0, 3.0) & (6.0, 10.0) & (-5.0, -5.0) & (-6.0, -6.0) \\
(2.0, 2.0) & (3.0, 3.0) & (-1.0, 3.0) & (-5.0, -5.0) \\
(1.0, 1.0) & (2.0, 2.0) & (-3.0, -3.0) & (0.0, 4.0)
\end{bmatrix} \]

Output:

- For ZGEEV:

\[ A = \begin{bmatrix}
(2.0000, 6.0000) & (-4.8694, -0.6574) & (-6.7837, -7.6341) & (17.1461, -0.1118) \\
(0.0000, 0.0000) & (4.0000, 8.0000) & (0.1361, 1.7912) & (-2.7907, -2.0982) \\
(0.0000, 0.0000) & (0.0000, 0.0000) & (3.0000, 7.0000) & (-3.0172, 0.1977) \\
(0.0000, 0.0000) & (0.0000, 0.0000) & (0.0000, 0.0000) & (1.0000, 5.0000)
\end{bmatrix} \]

- For ZGEEVX:

\[ A = \begin{bmatrix}
(2.0000, 6.0000) & (-1.1081, 4.9368) & (-3.3663, 3.6542) & (-19.9524, 4.0936) \\
(0.0000, 0.0000) & (4.0000, 8.0000) & (0.1597, 0.5962) & (-2.1519, 5.6785) \\
(0.0000, 0.0000) & (0.0000, 0.0000) & (3.0000, 7.0000) & (0.8130, 4.9939) \\
(0.0000, 0.0000) & (0.0000, 0.0000) & (0.0000, 0.0000) & (1.0000, 5.0000)
\end{bmatrix} \]

\[ W = \begin{bmatrix}
(2.0000, 6.0000) \\
(4.0000, 8.0000) \\
(3.0000, 7.0000) \\
(1.0000, 5.0000)
\end{bmatrix} \]

\[ VR = \begin{bmatrix}
(0.3780, 0.0000) & (0.5774, 0.0000) & (0.5774, 0.0000) & (0.7559, 0.0000) \\
(0.7559, 0.0000) & (0.5774, 0.0000) & (0.5774, 0.0000) & (0.3780, 0.0000) \\
(0.3780, 0.0000) & (0.5774, 0.0000) & (0.5774, 0.0000) & (0.3780, 0.0000) \\
(0.3780, 0.0000) & (0.5774, 0.0000) & (0.5774, 0.0000) & (0.3780, 0.0000)
\end{bmatrix} \]

\[ VL = \begin{bmatrix}
(-0.5774, 0.0000) & (-0.3780, 0.0000) & (-0.3780, 0.0000) & (0.0000, 0.0000) \\
(0.0000, 0.0000) & (-0.3780, 0.0000) & (-0.3780, 0.0000) & (-0.5774, 0.0000) \\
(0.3780, 0.0000) & (0.3780, 0.0000) & (0.7559, 0.0000) & (0.5774, 0.0000) \\
(0.3780, 0.0000) & (0.3780, 0.0000) & (0.7559, 0.0000) & (0.5774, 0.0000)
\end{bmatrix} \]

\[ RCONDE = \begin{bmatrix}
0.2182 \\
0.2182 \\
0.2182 \\
0.2182
\end{bmatrix} \]

\[ RCONDV = \begin{bmatrix}
0.3089 \\
0.6450 \\
0.1770 \\
0.5504
\end{bmatrix} \]

\[ SCALE = \begin{bmatrix}
1.000000 \\
1.000000 \\
1.000000 \\
1.000000
\end{bmatrix} \]

\[ ILO = 1 \\
IHI = 4 \\
ABNRM = 29.5 \\
INFO = 0 \]

Example 6

This example shows how to find the eigenvalues, left and right eigenvectors, and reciprocal condition numbers for the eigenvalues and right eigenvectors of a balanced long-precision complex general matrix \( A \) of order 4.
Note:
1. This matrix is used in Example 6.5 in referenced text on page 1367.
2. RWORK is a real array of length 8.
3. On output, A has been overwritten by the Schur form of the balanced matrix.

Call Statement and Input:

CALL ZGEESX ( 'B', 'V', 'V', 'B', 4, A, LD, W, VL, LDVL, VR, LDVR, ILO, IHI,
SCALE, ABNRM, RCONDE, RCONDV, WORK, LWORK, RWORK, INFO )

A =
\[
\begin{pmatrix}
(5.0, 9.0) & (5.0, 5.0) & (-6.0, -6.0) & (-7.0, -7.0) \\
(3.0, 3.0) & (6.0, 10.0) & (-5.0, -5.0) & (-6.0, -6.0) \\
(2.0, 2.0) & (3.0, 3.0) & (-1.0, 3.0) & (-5.0, -5.0) \\
(1.0, 1.0) & (2.0, 2.0) & (-3.0, -3.0) & (0.0, 4.0)
\end{pmatrix}
\]

Output:

A =
\[
\begin{pmatrix}
(2.0000, 6.0000) & (0.2165, -4.9088) & (6.7861, -7.6319) & (-16.4572, 4.8125) \\
(0.0000, 0.0000) & (4.0000, 8.0000) & (0.1841, 1.7868) & (1.5401, -3.1335) \\
(0.0000, 0.0000) & (0.0000, 0.0000) & (3.0000, 7.0000) & (-0.6773, -2.9469) \\
(0.0000, 0.0000) & (0.0000, 0.0000) & (0.0000, 0.0000) & (1.0000, 5.0000)
\end{pmatrix}
\]

W =
\[
\begin{pmatrix}
(2.0000, 6.0000) \\
(4.0000, 8.0000) \\
(3.0000, 7.0000) \\
(1.0000, 5.0000)
\end{pmatrix}
\]

VR =
\[
\begin{pmatrix}
(0.3780, 0.0000) & (0.5774, 0.0000) & (0.5774, 0.0000) & (0.7559, 0.0000) \\
(0.7559, 0.0000) & (0.5774, 0.0000) & (0.5774, 0.0000) & (0.3780, 0.0000) \\
(0.3780, 0.0000) & (0.5774, 0.0000) & (0.0000, 0.0000) & (0.3780, 0.0000) \\
(0.3780, 0.0000) & (0.0000, 0.0000) & (0.5774, 0.0000) & (0.3780, 0.0000)
\end{pmatrix}
\]

VL =
\[
\begin{pmatrix}
(-0.5774, 0.0000) & (-0.3780, 0.0000) & (-0.3780, 0.0000) & (0.0000, 0.0000) \\
(0.0000, 0.0000) & (-0.3780, 0.0000) & (-0.3780, 0.0000) & (-0.5774, 0.0000) \\
(0.5774, 0.0000) & (0.3780, 0.0000) & (0.7559, 0.0000) & (0.5774, 0.0000) \\
(0.5774, 0.0000) & (0.7559, 0.0000) & (0.3780, 0.0000) & (0.5774, 0.0000)
\end{pmatrix}
\]

RCONDE =
\[
\begin{pmatrix}
0.1633 \\
0.2108 \\
0.2108 \\
0.2887
\end{pmatrix}
\]

RCONDV =
\[
\begin{pmatrix}
0.4507 \\
0.4293 \\
0.1317 \\
0.5114
\end{pmatrix}
\]

SCALE =
\[
\begin{pmatrix}
2.000000 \\
1.000000 \\
1.000000 \\
1.000000
\end{pmatrix}
\]
ILO = 1
IHI = 4
ABNRM = 27.3
INFO = 0
SSYEV, DSYEV, CHEEV, ZHEEV, SSPEVX, DSPEVX, CHPEVX, ZHPEVX, SSYEVX, DSYEVX, CHEEVX, and ZHEEVX (Eigenvalues and, Optionally, the Eigenvectors of a Real Symmetric or Complex Hermitian Matrix)

Purpose

These subroutines compute eigenvalues and, optionally, the eigenvectors of a real symmetric matrix or a complex Hermitian matrix:

- SSYEV and DSYEV compute all eigenvalues and, optionally, the eigenvectors of real symmetric matrix $A$, stored in lower or upper storage mode.
- CHEEV and ZHEEV compute all eigenvalues and, optionally, the eigenvectors of complex Hermitian matrix $A$, stored in lower or upper storage mode.
- SSPEVX and DSPEVX compute selected eigenvalues and, optionally, the eigenvectors of real symmetric matrix $A$, stored in lower or upper packed storage mode.
- CHPEVX and ZHPEVX compute selected eigenvalues and, optionally, the eigenvectors of complex Hermitian matrix $A$, stored in lower or upper packed storage mode.
- SSYEVX and DSYEVX compute selected eigenvalues and, optionally, the eigenvectors of real symmetric matrix $A$, stored in lower or upper storage mode.
- CHEEVX and ZHEEVX compute selected eigenvalues and, optionally, the eigenvectors of complex Hermitian matrix $A$, stored in lower or upper storage mode.

For SSYEV, DSYEV, CHEEV, and ZHEEV, eigenvalues are returned in vector $w$, and eigenvectors are returned in matrix $A$.

For SSPEVX, DSPEVX, CHPEVX, ZHPEVX, SSYEVX, DSYEVX, CHEEVX, and ZHEEVX eigenvalues are returned in vector $w$, and eigenvectors are returned in matrix $Z$.

The formula is shown below:

$$Az = wz$$

where $A = A^T$ or $A = A^H$.

Table 191. Data Types

<table>
<thead>
<tr>
<th>$vl, vu, abstol, w, rwork$</th>
<th>$A, Z, work$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SSYEV$^a$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SSPEVX$^a$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SSYEV$^a$</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DSYEV$^a$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DSPEVX$^a$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DSYEV$^a$</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>Short-precision complex</td>
<td>CHEEV$^a$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CHPEVX$^a$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CHEEV$^a$</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision complex</td>
<td>ZHEEV$^a$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZHPEVX$^a$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZHEEV$^a$</td>
</tr>
<tr>
<td>C and C++</td>
<td>Fortran</td>
<td>LAPACK</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
<td>--------</td>
</tr>
</tbody>
</table>
| CALL SYEV | CALL CHEEV | info = LAPACKE_ssyev
| CALL SYEV | CALL CHEEV | LAPACKE_dsyev
| CALL SYEV | CALL CHEEV | LAPACKE_sspevx
| CALL SYEV | CALL CHEEV | LAPACKE_zspevx
| CALL SYEV | CALL CHEEV | LAPACKE_ssyevx
| CALL SYEV | CALL CHEEV | LAPACKE_zsyevx
| CALL SYEV | CALL CHEEV | LAPACKE_zheevx
| CALL SYEV | CALL CHEEV | LAPACKE_zheevx |

### On Entry

**matrix_layout**

- indicates whether the input and output matrices are stored in row major order or column major order, where:
  - If `matrix_layout = LAPACK_ROW_MAJOR`, the matrices are stored in row major order.
  - If `matrix_layout = LAPACK_COL_MAJOR`, the matrices are stored in column major order.
If matrix_layout = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. It must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR

Jobz indicates the type of computation to be performed, where:
If jobz = 'N', eigenvalues only are computed.
If jobz = 'V', eigenvalues and eigenvectors are computed.
Specified as: a single character; jobz = 'N' or 'V'.

Range indicates the type of computation to be performed, where:
If range = 'A', all eigenvalues are to be found.
If range = 'V', all eigenvalues in the interval [vl, vu] are to be found.
If range = 'I', the il-th through iu-th eigenvalues are to be found.
Specified as: a single character; range = 'A', 'V', or 'I'.

Uplo indicates whether the upper or lower triangular part of the matrix A is referenced, where:
If uplo = 'U', the upper triangular part is referenced.
If uplo = 'L', the lower triangular part is referenced.
Specified as: a single character; uplo = 'U' or 'L'.

N is the order of matrix A used in the computation.
Specified as: an integer; n ≥ 0.

Ap is the real symmetric or complex Hermitian matrix A of order n. It is stored in an array, referred to as AP, where:
If uplo = 'U', it is stored in upper-packed storage mode.
If uplo = 'L', it is stored in lower-packed storage mode.
Specified as: one-dimensional array of (at least) length n(n + 1)/2, containing numbers of the data type indicated in Table 191 on page 959.

A is the real symmetric or complex Hermitian matrix A of order n.
If uplo = 'U', it is stored in upper storage mode.
If uplo = 'L', 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 191 on page 959.

LDA is the leading dimension of the array specified for A.
Specified as: an integer; lda > 0 and lda ≥ n.

Vl has the following meaning:
If range = 'V', it is the lower bound of the interval to be searched for eigenvalues.
If range ≠ 'V', this argument is ignored.
Specified as: a number of the data type indicated in Table 191 on page 959. If range = 'V', vl < vu.

vu has the following meaning:
If range = 'V', it is the upper bound of the interval to be searched for eigenvalues.
If range ≠ 'V', this argument is ignored.

Specified as: a number of the data type indicated in Table 191 on page 959. If range = 'V', vl < vu.

il has the following meaning:
If range = 'T', it is the index (from smallest to largest) of the smallest eigenvalue to be returned.
If range ≠ 'T', this argument is ignored.
Specified as: an integer; il ≥ 1.

iu has the following meaning:
If range = 'T', it is the index (from smallest to largest) of the largest eigenvalue to be returned.
If range ≠ 'T', this argument is ignored.
Specified as: an integer; min(il, n) ≤ iu ≤ n.

abstol
is the absolute tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a, b] of width less than or equal to:

\[ \text{abstol} + \epsilon (\max(|a|, |b|)) \]

where \( \epsilon \) is the machine precision. If abstol is less than or equal to zero, then \( \epsilon (\text{norm}(T)) \) is used in its place, where \( \text{norm}(T) \) is the one-norm of the tridiagonal matrix obtained by reducing \( A \) to tridiagonal form. For most problems, this is the appropriate level of accuracy to request.

For certain strongly graded matrices, greater accuracy can be obtained in very small eigenvalues by setting abstol to a very small positive number. However, if abstol is less than:

\[ \sqrt{\text{unfl}} \]

where unfl is the underflow threshold, then:

\[ \sqrt{\text{unfl}} \]

is used in its place.

Eigenvalues are computed most accurately when abstol is set to twice the underflow threshold—that is, \( 2(\text{unfl}) \).

If jobz = 'V', setting abstol to unfl (the underflow threshold) yields the most orthogonal eigenvectors.

Note:
1. The approximate values of the constants used for abstol are listed below:
For SSPEVX, CHPEVX, SSYEVX, and CHEEVX

- \( \epsilon = 0.119209289550781250 \times 10^{-6} \)
- \( unfl = 0.1175494351 \times 10^{-37} \)
- \( \sqrt{unfl} = 0.1084202172 \times 10^{-18} \)

For DSPEVX, ZHPEVX, DSYEVX, and ZHEEVX

- \( \epsilon = 0.222044604925031308 \times 10^{-15} \)
- \( unfl = 0.222507385850720138 \times 10^{-307} \)
- \( \sqrt{unfl} = 0.149166814624004135 \times 10^{-153} \)

2. The value of \( abstol \) can affect which algorithm is used to compute the eigenvalues and eigenvectors. See [Function](#).

Specified as: a number of the data type indicated in [Table 191 on page 959](#).

\( m \) See [On Return](#).

\( w \) See [On Return](#).

\( z \) See [On Return](#).

\( ldz \) is the leading dimension of the array specified for Z.

Specified as: an integer; \( ldz > 0 \) and, if \( jobz = 'V' \), \( ldz \geq n \).

\( work \) is a work area used by these subroutines, where:

For SSYEV, DSYEV, CHEEV, ZHEEV, SSYEVX, DSYEVX, CHEEVX, and ZHEEVX

- If \( lwork = 0 \), \( work \) is ignored.
- If \( lwork \neq 0 \), the size of \( work \) is determined as follows:
  - If \( lwork \neq -1 \), \( work \) is (at least) of length \( lwork \).
  - If \( lwork = -1 \), \( work \) is (at least) of length 1.

For SSPEVX and DSPEVX

Its size is \( 8n \).

For CHPEVX and ZHPEVX

Its size is \( 2n \).

Specified as: an area of storage containing numbers of the data type indicated in [Table 191 on page 959](#).

\( lwork \) is used to determine the size of the WORK array.

Specified as: an integer, where:

- If \( lwork = 0 \), the subroutine dynamically allocates the workspace needed for use during this computation. The dynamically allocated workspace will be freed prior to returning control to the calling program.
- If \( lwork = -1 \), a workspace query is assumed. The subroutine only calculates the optimal size of the WORK array and returns this value as the first entry of the WORK array.

Otherwise:

**For SSYEV and DSYEV**
\( lwork \geq \max(1, 3n-1) \).

**For CHEEV and ZHEEV**
\( lwork \geq \max(1, 2n-1) \).

**For SSYEVX and DSYEVX**
\( lwork \geq \max(1, 8n) \).

**For CHEEVX and ZHEEVX**
\( lwork \geq \max(1, 2n) \).

**Note:** These formulas represent the minimum workspace required. For best performance, specify either \( lwork = -1 \) (to obtain the optimal size to use) or \( lwork = 0 \) (to direct the subroutine to dynamically allocate the workspace).

**rwork**
is a work area:

**For CHEEV and ZHEEV**
rwork is a work area of size \( \max(1, 3n-2) \).

**For CHPEVX, ZHPEVX, CHEEVX, and ZHEEVX**
rwork is a work area of size \( 7n \).

Specified as: an area of storage containing real numbers of the data type indicated in Table 191 on page 959

**iwork**
is a work area of size \( 5n \).

Specified as: an area of storage containing integers.

**ifail**
See **On Return**

**On Return**

**ap** On exit, the matrix \( A \) is overwritten by values generated during the reduction to tridiagonal form.

If \( uplo = 'U' \), the diagonal and first superdiagonal of the tridiagonal matrix \( T \) overwrite the corresponding elements of \( A \).

If \( uplo = 'L' \), the diagonal and first subdiagonal of \( T \) overwrite the corresponding elements of \( A \).

Returned as: a one-dimensional array of (at least) length \( n(n + 1)/2 \), containing numbers of the data type indicated in Table 191 on page 959

**a**

**For SSYEV, DSYEV, CHEEV, and ZHEEV**

On exit:

- If \( jobz = 'V' \), the columns of \( a \) contain the orthonormal eigenvectors of the matrix \( A \) corresponding to the computed eigenvalues, with the \( i \)-th column of \( A \) holding the eigenvector associated with \( w(i) \).
- If \( jobz = 'N' \):
If uplo = 'U', the leading \( n \) by \( n \) upper triangular part of \( A \) is overwritten.

If uplo = 'L', the leading \( n \) by \( n \) lower triangular part of \( A \) is overwritten.

For SSYEVX, SSPEVX, DSPEVX, CHPEVX, ZHPEVX, DSYEVX, CHEEVX, and ZHEEVX

On exit, the matrix \( A \) is overwritten by values generated during the reduction to tridiagonal form.

If uplo = 'U', the diagonal and first superdiagonal of the tridiagonal matrix \( T \) overwrite the corresponding elements of \( A \).

If uplo = 'L', the diagonal and first subdiagonal of \( T \) overwrite the corresponding elements of \( A \).

Returned as: an array of dimension \( lda \) by (at least) \( n \), containing numbers of the data type indicated in Table 191 on page 959

\( m \) is the number of eigenvalues found.

Returned as: an integer; \( 0 \leq m \leq n \).

\( w \)

For SSYEV, DSYEV, CHEEV, and ZHEEV

is the vector \( w \) of length \( n \) containing the computed eigenvalues in ascending order.

For SSYEVX, SSPEVX, DSPEVX, CHPEVX, ZHPEVX, DSYEVX, CHEEVX, and ZHEEVX

is the vector \( w \), containing the computed eigenvalues in ascending order in the first \( m \) elements of \( w \).

Returned as: a one-dimensional array of (at least) length \( n \), containing numbers of the data type indicated in Table 191 on page 959

\( z \) has the following meaning, where:

If jobz = 'N', then \( z \) is ignored.

If jobz = 'V', the first \( m \) columns of \( z \) contain the orthonormal eigenvectors of the matrix \( A \) corresponding to the computed eigenvalues, with the \( i \)-th column of \( z \) holding the eigenvector associated with \( w(i) \). If an eigenvector fails to converge, then that column of \( z \) contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in ifail.

Note: You must ensure that at least max(1, \( m \)) columns are supplied in the array \( z \); if range = 'V', the exact value of \( m \) is not known in advance and an upper bound must be used.

Returned as: an \( ldz \) by (at least) max(1, \( m \)) array, containing numbers of the data type indicated in Table 191 on page 959

\( work \)

is a work area used by these subroutines.

Returned as: an area of storage where:

If lwork = -1, then work is (at least) of length 1 and work contains the calculated optimal size of the WORK array.

If lwork ≠ -1 and lwork ≠ 0, then work is (at least) of length lwork and work contains the value specified for lwork.
Except for \textit{work}, the contents of \textit{work} are overwritten on return.

\textit{ifail} has the following meaning:

If \textit{jobz} = 'N', \textit{ifail} is ignored.

If \textit{jobz} = 'V':
\begin{itemize}
  \item If \textit{info} = 0, the first \textit{m} elements of \textit{ifail} are zero.
  \item If \textit{info} > 0, \textit{ifail} contains the indices of the eigenvectors that failed to converge.
\end{itemize}

Returned as: an array of length \textit{n}, containing integers.

\textit{info} has the following meaning:

\begin{verbatim}
For SSYEV, DSYEV, CHEEV, ZHEEV, LAPACKE_ssyev, LAPACKE_dsyev,
LAPACKE_cheev, and LAPACKE_zheev
  If \textit{info} = 0, then the algorithm converged. This indicates a normal exit.
  If \textit{info} = i, the algorithm failed to converge. i off-diagonal elements of
  an intermediate tridiagonal form did not converge to zero.

For SSPEVX, DSPEVX, CHPEVX, ZHPEVX, SSYEVX, DSYEVX, CHEEVX,
and ZHEEVX, LAPACKE_sspevx, LAPACKE_dspevx, LAPACKE_chpevx,
LAPACKE_zhpevx, LAPACKE_ssyevx, LAPACKE_dsyevx, LAPACKE_cheevx,
and LAPACKE_zheevx
  If \textit{info} = 0, then all eigenvectors converged. This indicates a normal
  exit.
  If \textit{info} = i, then i eigenvectors failed to converge. Their indices are
  saved in array \textit{ifail}.
\end{verbatim}

Returned as:
\begin{itemize}
  \item For SSYEV, DSYEV, CHEEV, ZHEEV, SSYEVX, DSPEVX, CHPEVX,
        ZHPEVX, SSYEVX, DSYEVX, CHEEVX, and ZHEEVX returned as: an integer; \textit{info} \geq 0.
  \item For LAPACKE_ssyev, LAPACKE_dsyev, LAPACKE_cheev, LAPACKE_zheev,
        LAPACKE_ssyevx, LAPACKE_dspevx, LAPACKE_chpevx,
        LAPACKE_zhpevx, LAPACKE_ssyevx, LAPACKE_dspevx, LAPACKE_chpevx,
        LAPACKE_zhpevx, and LAPACKE_zheevx returned as an integer function value; \textit{info} \geq 0.
\end{itemize}

\textbf{Notes}

1. This subroutine accepts lowercase letters for the \textit{jobz}, \textit{range}, and \textit{uplo} arguments.
2. In your C program, the arguments \textit{info} and \textit{m} must be passed by reference.
3. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix \textit{A} are assumed to be zero, so you do not have to set these values.
4. \textit{A}, \textit{Z}, \textit{w}, \textit{ifail}, \textit{work}, \textit{rwork}, \textit{iwork} must have no common elements; otherwise, results are unpredictable.
5. For a description of how real symmetric matrices are stored in lower- or upper-packed storage mode, see \textit{“Lower-Packed Storage Mode”} on page 85 or \textit{“Upper-Packed Storage Mode”} on page 87, respectively. For a description of how complex Hermitian matrices are stored in lower- or upper-packed storage mode, see \textit{“Complex Hermitian Matrix”} on page 90.
6. For best performance specify \textit{lwork} = 0.
Function

For SSYEV, DSYEV, CHEEV, and ZHEEV

These subroutines compute all eigenvalues and, optionally, the eigenvectors of a real symmetric or complex Hermitian matrix $A$, stored in lower or upper storage mode. (If $n = 0$, the subroutine returns after completing parameter checking.)

The computation involves the following steps:

1. If necessary, scale the matrix $A$.
2. Reduce the matrix to real symmetric tridiagonal form.
3. Compute all eigenvalues and, optionally, the eigenvectors.
   - If $jobz = 'N'$, compute all the eigenvalues of the symmetric tridiagonal matrix using the Pal-Walker-Kahan variant of the QL or QR algorithm.
   - Otherwise, for $jobz = 'V'$, compute all eigenvalues and eigenvectors of the symmetric tridiagonal matrix using the implicit QL or QR method.
4. Rescale eigenvalues appropriately if matrix $A$ was scaled.

For SSYEVX, SSPEVX, DSPEVX, CHPEVX, ZHPEVX, DSYEVX, CHEEVX, and ZHEEVX

These subroutines compute selected eigenvalues and, optionally, the eigenvectors of a real symmetric or complex Hermitian matrix $A$, stored in lower-packed or upper-packed storage mode or in lower or upper storage mode. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues. (If $n = 0$, the subroutine returns after completing parameter checking.)

The computation involves the following steps:

1. If necessary, scale the matrix $A$.
2. Reduce the matrix to real symmetric tridiagonal form.
3. Compute the selected eigenvalues and, optionally, the eigenvectors. The algorithm used depends on the value specified for $abstol$ and whether or not all eigenvalues are requested.
   a. If $abstol \leq 0$ and all eigenvalues were requested (that is, $range = 'A'$ or $range = 'I'$ with $il = 1$ and $iu = n$), do the following:
      - If $jobz = 'N'$, compute all the eigenvalues of the symmetric tridiagonal matrix using the Pal-Walker-Kahan variant of the QL or QR algorithm.
      - Otherwise, for $jobz = 'V'$, compute all eigenvalues and eigenvectors of the symmetric tridiagonal matrix using the implicit QL or QR method.
   b. Otherwise, if $abstol > 0$, or if a subset of the eigenvalues was requested via $range = 'T'$ or $range = 'V'$, or if the previous step failed to compute all eigenvalues, do the following:
      1) Compute the requested eigenvalues using bisection. If $abstol \leq 0$, then $\epsilon(n\text{norm}(T))$ is used in its place, where $\text{norm}(T)$ is the one-norm of the tridiagonal matrix obtained by reducing $A$ to tridiagonal form.
      2) If the eigenvectors were also requested, compute the eigenvectors using inverse iteration.
4. Rescale eigenvalues appropriately if matrix $A$ was scaled.
For more information on these methods, see references [8 on page 1363], [34 on page 1365], and [74 on page 1367].

Error conditions

Resource Errors
1. \( lwork = 0 \), and unable to allocate work area.

Computational Errors
1. Bisection failed to converge for some eigenvalues. The eigenvalues may not be as accurate as the absolute and relative tolerances.
2. The number of eigenvalues computed does not match the number of eigenvalues requested.
3. No eigenvalues were computed because the Gershgorin interval initially used was incorrect.
4. Some eigenvectors failed to converge. The indices are stored in \( ifail \).
5. The subroutine computed the eigenvalues using multiple algorithms. Performance may be degraded.
6. If \( info = i \), the algorithm failed to converge. \( i \) indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

Note: The default for the number of allowable errors for error conditions 2154, 2155, 2156, 2157, 2162, and 2613 is set to be unlimited in the ESSL error option table; therefore, each computational error message may occur multiple times with processing continuing after each error.

Input-Argument Errors
1. \( jobz \neq 'N' \) or \( 'V' \)
2. \( range \neq 'A', 'V', \) or \( 'I' \)
3. \( uplo \neq 'U' \) or \( 'L' \)
4. \( n < 0 \)
5. \( range = 'V', n > 0, \) and \( vu \leq vl \)
6. \( range = 'I' \) and \( (i < 1 \) or \( i > \max(1, n)) \)
7. \( range = 'I' \) and \( (iu < \min(n, i) \) or \( iu > n) \)
8. \( lda \leq 0 \)
9. \( lda < n \)
10. \( ldz \leq 0 \)
11. \( jobz = 'V' \) and \( ldz < n \)
12. \( lwork = 0 \) and \( lwork \neq -1 \) and \( lwork < \) the minimum required value

Examples
Example 1
This example shows how to find the eigenvalues only of a real symmetric matrix \( A \) of order 4, stored in lower-packed storage mode.

Note: This matrix is Example 4.1 in referenced text [74 on page 1367].

Matrix \( A \) is:

\[
\begin{bmatrix}
5.0 & 4.0 & 1.0 & 1.0 \\
4.0 & 5.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 4.0 & 2.0 \\
1.0 & 1.0 & 2.0 & 4.0 \\
\end{bmatrix}
\]
Call Statement and Input:

```
JOBZ RANGE UPLO N AP VL VU IL IU ABSTOL M W Z LDZ WORK IWORK IFAIL INFO
```

CALL DSPEVX ('N', 'A', 'L', 4, AP, VL, VU, IL, IU, -1.0, M, W, Z, 4, WORK, IWORK, IFAIL, INFO)

```
AP = (5.0, 4.0, 1.0, 1.0, 5.0, 1.0, 1.0, 4.0, 2.0, 4.0)
```

Output:

```
M = 4
```

```
AP = (5.000000, -4.242641, 0.121320, 0.121320, 6.000000, 1.414214, 0.414214, 5.000000, 0.000000, 2.000000)
```

```
W = 
[1.000000]
[2.000000]
[5.000000]
[10.000000]
```

Z is not used when JOBZ = 'N'.

```
IFAIL = (0,0,0,,)
INFO = 0
```

**Example 2**

This example shows how to find the eigenvalues and eigenvectors of a real symmetric matrix $A$ of order 4, stored in upper-packed storage mode. This example also illustrates the use of the $il$ and $iu$ arguments when range = 'T'.

**Note:** This matrix is Example 4.1 in referenced text [74 on page 1367].

Matrix $A$ is the same matrix used for DSPEVX in [Example 1]

Call Statement and Input:

```
JOBZ RANGE UPLO N AP VL VU IL IU ABSTOL M W Z LDZ WORK IWORK IFAIL INFO
```

CALL DSPEVX ('V', 'I', 'U', 4, AP, VL, VU, 1, 3, -1.0, M, W, Z, 4, WORK, IWORK, IFAIL, INFO)

```
AP = (5.0, 4.0, 5.0, 1.0, 1.0, 4.0, 1.0, 1.0, 2.0, 4.0)
```

Output:

```
M = 3
```

```
AP = (1.000000, 0.000000, 6.000000, 0.414214, 2.828427, 7.000000, 0.224745, 0.224725, -2.449490, 4.000000)
```

```
W = 
[1.000000]
[2.000000]
[5.000000]
```

```
Z = 
[0.707107  0.000000 -0.316228]
[-0.707107  0.000000 -0.316228]
[0.000000  -0.707107  0.632456]
[0.000000    0.707107  0.632456]
```

```
IFAIL = (0,0,0,,)
INFO = 0
```

**Example 3**
This example shows how to find the eigenvalues and eigenvectors of a real symmetric matrix $A$ of order 4, stored in upper-packed storage mode. This example also illustrates the use of the $vl$ and $vu$ arguments when $range = 'V'$.

**Note:** This matrix is Example 4.1 in Reference [74 on page 1367].

Matrix $A$ is the same matrix used for DSPEVX in Example 1.

**Call Statement and Input:**

```
JOBZ RANGE UPLO N AP VL VU IL IU ABSTOL M W Z LDZ WORK IWORK IFAIL INFO
CALL DSPEVX ('V', 'V', 'U', 4, AP, 3.0, 11.0, 0, 0, -1.0, M, W, Z, 4, WORK, IWORK, IFAIL, INFO)
```

```
AP = (5.0, 4.0, 5.0, 1.0, 1.0, 4.0, 1.0, 1.0, 2.0, 4.0)
```

**Output:**

```
M = 2
```

```
AP = (1.000000, 0.000000, 6.000000, 0.414214, 2.828427, 7.000000, 0.224745, 0.224725, -2.449490, 4.000000)
```

```
W =

5.000000  
10.000000  

Z =

-0.316228 -0.632456  
-0.316228 -0.632456  
0.632456 -0.316228  
0.632456 -0.316228  

IFAIL = (0,0,...)  
INFO = 0
```

**Example 4**

This example shows how to find the eigenvalues only of a complex Hermitian matrix $A$ of order 3, stored in lower-packed storage mode.

**Note:** This matrix is Example 6.3 in referenced text [74 on page 1367].

Matrix $A$ is:

```
(2.0, 0.0) (0.0, 1.0) (0.0, 0.0)  
(0.0, -1.0) (2.0, 0.0) (0.0, 0.0)  
(0.0, 0.0) (0.0, 0.0) (3.0, 0.0) 
```

**Call Statement and Input:**

```
JOBZ RANGE UPLO N AP VL VU IL IU ABSTOL M W Z LDZ WORK RWORK IWORK IFAIL INFO
CALL ZHPEVX ('N', 'A', 'L', 3, AP, VL, VU, IL, IU, -1.0, M, W, Z, 3, WORK, RWORK, IWORK, IFAIL, INFO)
```

```
AP = ((2.0, . ), (0.0, -1.0), (0.0, 0.0), (2.0, . ), (0.0, 0.0), (3.0, . ))
```

**Output:**

```
M = 3
```

```
AP = (2.0, 0.0), (-1.0, 0.0), (0.0, 0.0), (2.0, 0.0), (0.0, 0.0), (3.0, 0.0))
```
\[ W = \begin{bmatrix} 1.000000 \\ 3.000000 \\ 3.000000 \end{bmatrix} \]

Z is not used when JOBZ = 'N'.
IFAIL is not used when JOBZ = 'N'.
INFO = 0

**Example 5**

This example shows how to find the eigenvalues and eigenvectors of a complex Hermitian matrix \( A \) of order 3, stored in upper-packed storage mode. This example also illustrates the use of the \( il \) and \( iu \) arguments when range = 'T'.

**Note:** This matrix is Example 6.3 in referenced text [74 on page 1367]. Matrix \( A \) is the same matrix used for ZHPEVX in Example 4.

**Call Statement and Input:**

```
CALL ZHPEVX ('T', 'T', 'U', 3, AP, VL, VU, IL, IU, ABSTOL, M, W, Z, LDZ, WORK, RWORK, IWORK, IFAIL, INFO)
```

**Output:**

\[ AP = ((2.0, .), (0.0, 1.0), (2.0, .), (0.0, 0.0), (0.0, 0.0), (3.0, .)) \]

\[ W = \begin{bmatrix} 1.000000 \\ 3.000000 . \end{bmatrix} \]

\[ Z = \begin{bmatrix} (0.0000, -0.7071), (0.0000, 0.1591) \\ (0.7071, 0.0000), (0.1591, 0.0000) \\ (0.0000, 0.0000), (0.9744, 0.0000) \end{bmatrix} \]

IFAIL = (0,0,)
INFO = 0

**Example 6**

This example shows how to find the eigenvalues and eigenvectors of a complex Hermitian matrix \( A \) of order 3, stored in upper-packed storage mode. This example also illustrates the use of the \( vl \) and \( vu \) arguments when range = 'V'.

**Note:** This matrix is Example 6.3 in referenced text [74 on page 1367]. Matrix \( A \) is the same matrix used for ZHPEVX in Example 4.

**Call Statement and Input:**

```
CALL ZHPEVX ('V', 'V', 'U', 3, AP, VL, VU, IL, IU, ABSTOL, M, W, Z, LDZ, WORK, RWORK, IWORK, IFAIL, INFO)
```

**Output:**

\[ AP = ((2.0, .), (0.0, 1.0), (2.0, .), (0.0, 0.0), (0.0, 0.0), (3.0, .)) \]
Example 7

This example shows how to find the eigenvalues only of a symmetric matrix $A$ of order 4.

Note:
1. This matrix is Example 4.1 in referenced text [74 on page 1367].
2. Because $lwork = 0$, the subroutine dynamically allocates $WORK$.

Matrix $A$ is the same matrix used for DSPEVX in Example 1

Call Statement and Input:

```
CALL DSYEV ( 'N', 'L', 4, A, 4, W, WORK, 0, INFO)
```

```
JOBZ UPLO N A LDA W WORK LWORK INFO
CALL DSYEV ( 'N', 'L', 4, A, 4, W, WORK, 0, INFO)
--or--

```

```
JOBZ RANGE UPLO N A LDA VL VU IL IU ABSTOL M W Z LDZ WORK LWORK IWORK IFAIL INFO
CALL DSYEVX ( 'N', 'L', 4, A, 4, VL, VU, IL, IU, -1.0, M, W, Z, 4, WORK, 0, IWORK, IFAIL, INFO)
```

```
A =
[ 5.0 , 4.0 , 1.0 , 1.0 ]
[ 4.0 , 5.0 , 1.0 , 2.0 ]
[ 1.0 , 1.0 , 4.0 , 4.0 ]
[ 1.0 , 2.0 , 4.0 ]
```

Output:

$M = 4$

For DSYEV, $A$ has been overwritten on output.

For DSYEVX:

```
A =
[ 5.000000 , -4.242641 , 0.121320 , 0.121320 ]
[ -4.242641 , 6.000000 , 1.414214 , 1.414214 ]
[ 0.121320 , 1.414214 , 5.000000 , 0.000000 ]
[ 0.121320 , 1.414214 , 0.000000 , 2.000000 ]
```

```
W =
[ 1.000000 ]
[ 2.000000 ]
[ 5.000000 ]
[10.000000 ]
```

```
M = 2
AP = ((2.0, 0.0), (-1.0, 0.0), (2.0, 0.0), (0.0, 0.0), (0.0, 0.0), (3.0, 0.0))

W =
[ 3.000000 ]
[ 3.000000 ]
[ . ]

Z =
[ ( 0.0000, -0.6634), ( 0.0000, -0.2447) ]
[ (-0.6634, 0.0000), (-0.2447, 0.0000) ]
[ (-0.3460, 0.0000), ( 0.9382, 0.0000) ]

IFAIL = (0,0,.)
INFO = 0
Z is not used when JOBZ = 'N'.

IFAIL is not used when JOBZ = 'N'.

INFO = 0

Example 8

This example shows how to find the eigenvalues and eigenvectors of a real symmetric matrix $A$ of order 4 stored in upper storage mode.

Note:
1. This matrix is Example 4.1 in referenced text [74 on page 1367].
2. Because lwork = 0, the subroutine dynamically allocates WORK.

Matrix $A$ is the same matrix used for DSPEVX in Example 1

Call Statement and Input:

```
CALL DSYEV ( 'V', 'U', 4, A, 4, W, WORK, 0, INFO)
```

Output:

```
A =
  5.0  4.0  1.0  1.0
  .  5.0  1.0  1.0
  .  .  4.0  2.0
  .  .  .  4.0

W =
  0.707107  0.000000  0.316228  0.632456
  -0.707107  0.000000  0.316228  0.632456
  0.000000  0.707107 -0.632456  0.316228
  0.000000 -0.707107 -0.632456  0.316228

INFO = 0
```

Example 9

This example shows how to find the eigenvalues and eigenvectors of a real symmetric matrix $A$ of order 4. This example also illustrates the use of the il and iu arguments when range = 'I'.

Note:
1. This matrix is Example 4.1 in referenced text [74 on page 1367].
2. Because lwork = 0, the subroutine dynamically allocates WORK.

Matrix $A$ is the same matrix used for DSPEVX in Example 1

Call Statement and Input:

```
CALL DSYEVX ( 'V', 'I', 'U', 4, A, 4, VL, VU, 1, 3, -1.0, M, W, Z, 4, WORK, 0, IWORK, IFAIL, INFO)
```

Output:
Example 10

This example shows how to find the eigenvalues and eigenvectors of a real symmetric matrix $A$ of order 4. This example also illustrates the use of the $vl$ and $vu$ arguments when $range = 'V'$.

Note:
1. This matrix is Example 4.1 in referenced text on page 1367.
2. Because $lwork = 0$, the subroutine dynamically allocates $WORK$.

Matrix $A$ is the same matrix used for DSPEVX in Example 1.

Call Statement and Input:

```
CALL DSYEVX ('V', 'V', 'U', 4, A, 4, 3.0, 11.0, IL, IU, -1.0, M, W, Z, 4, WORK, 0, IWORK, IFAIL, INFO)
```

Output:

```
M = 2
A =
  1.000000 0.000000 0.414214 0.224745
   . 6.000000 2.828427 0.224745
   . . 7.000000 -2.449490
   . . . 4.000000
W =
  1.000000
  2.000000
  5.000000
  .
Z =
  0.707107 0.000000 -0.316228
 -0.707107 0.000000 -0.316228
  0.000000 -0.707107 0.632456
  0.000000 0.707107 0.632456
IFAIL = (0, 0, .) INFO = 0
```
Example 11

This example shows how to find the eigenvalues only of a complex Hermitian matrix $A$ of order 3.

Note:
1. This matrix is Example 6.3 in referenced text [74 on page 1367].
2. Because $lwork = 0$, the subroutine dynamically allocates $WORK$.

Note:
For ZHEEVX, matrix $A$ is the same matrix used for ZHPEVX in Example 4.

Call Statement and Input:

```
CALL ZHEEV ('N', 'L', 3, A, 3, W, WORK, 0, RWORK, INFO)
```

-or-

```
CALL ZHEEVX ('N', 'A', 'L', 3, A, LDA, VL, VU, IL, IU, -1.0, M, W, Z, WORK, 0, RWORK, IWORK, IFAIL, INFO)
```

```
A = 
| 2.0, 0.0 |
| 0.0, -1.0 |
| 0.0, 0.0 |

W = 
| 1.000000 |
| 3.000000 |
| 3.000000 |
```

Output:
- For ZHEEV, $A$ has been overwritten on output.
- For ZHEEVX:

```
A = 
| 2.0, 0.0 |
| -1.0, 0.0 |
| 0.0, 0.0 |

W = 
| 1.000000 |
| 3.000000 |
| 3.000000 |
```

$Z$ is not used when $JOBZ = 'N'$.

IFAIL is not used when $JOBZ = 'N'$

INFO = 0

Example 12

This example shows how to find the eigenvalues and eigenvectors of a complex Hermitian matrix $A$ of order 3 stored in upper storage mode.

Note:
1. This matrix is Example 6.3 in referenced text [74 on page 1367].
2. Because $lwork = 0$, the subroutine dynamically allocates $WORK$.

Matrix $A$ is the same matrix used for ZHPEVX in Example 4.
Example 13

This example shows how to find the eigenvalues and eigenvectors of a complex Hermitian matrix \( A \) of order 3. This example also illustrates the use of the \( il \) and \( iu \) arguments when \( range = 'I' \).

Note:
1. This matrix is Example 6.3 in referenced text \([74\text{ on page }1367]\).
2. Because \( lwork = 0 \), the subroutine dynamically allocates \( WWORK \).

Matrix \( A \) is the same matrix used for \( ZHPEVX \) in **Example 4**.

---

Example 14

---
This example shows how to find the eigenvalues and eigenvectors of a complex Hermitian matrix $A$ of order 3. This example also illustrates the use of the $vl$ and $vu$ arguments when range = 'V'.

**Note:**
1. This matrix is Example 6.3 in referenced text [74 on page 1367](#).
2. Because $lwork = 0$, the subroutine dynamically allocates $WORK$.

Matrix $A$ is the same matrix used for ZHPEVX in Example 4

**Call Statement and Input:**

```
JOBZ RANGE UPL0 N A LDA VL VU IL IU ABSTOL M W Z LDZ WORK LWORK RWORK IWORK IFAIL INFO
CALL ZHEEVX ('V', 'V', 'U', 3, A, LDA, 2.0, 4.0, IL, IU, -1.0, M, W, Z, 3, WORK, 0, RWORK, IWORK, IFAIL, INFO)
```

**Output:**

```
 M = 2
 A =
  ( 2.0, 0.0) (1.0, 0.0) (0.0, 0.0)
  . (2.0, 0.0) (0.0, 0.0)
  . . (3.0, 0.0)

 W =
  [ 3.000000
    3.000000
    .
  ]

 Z =
  [ (0.0000, 0.6634), (0.0000, 0.2447)
    (-0.6634, 0.0000), (-0.2447, 0.0000)
    (-0.3460, 0.0000), (0.9382, 0.0000)
  ]

 IFAIL = (0,0,0)
 INFO = 0
```
SSPEVD, DSPEVD, CHPEVD, ZHPEVD, SSYEVD, DSYEVD, CHEEVD, and ZHEEVD (Eigenvalues and, Optionally the Eigenvectors, of a Real Symmetric or Complex Hermitian Matrix Using a Divide-and-Conquer Algorithm)

Purpose

These subroutines compute eigenvalues and, optionally, the eigenvectors of a real symmetric matrix or a complex Hermitian matrix.

If eigenvalues only are computed, these subroutines compute the eigenvalues using the Pal-Walker-Kahan variant of the QL or QR algorithm.

If eigenvectors are computed, the subroutine uses a divide-and-conquer method to compute them:

- SSPEVD and DSPEVD compute eigenvalues and, optionally, the eigenvectors of real symmetric matrix $A$, stored in lower- or upper-packed storage mode.
- CHPEVD and ZHPEVD compute eigenvalues and, optionally, the eigenvectors of complex Hermitian matrix $A$, stored in lower- or upper-packed storage mode.
- SSYEVD and DSYEVD compute eigenvalues and, optionally, the eigenvectors of real symmetric matrix $A$, stored in lower or upper storage mode.
- CHEEVD and ZHEEVD compute eigenvalues and, optionally, the eigenvectors of complex Hermitian matrix $A$, stored in lower or upper storage mode.

Eigenvalues are returned in vector $w$ and eigenvectors are returned in matrix $Z$ (for subroutines SSPEVD, DSPEVD, CHPEVD, ZHPEVD) or in matrix $A$ (for subroutines SSYEVD, DSYEVD, CHEEVD, ZHEEVD):

$$Az = wz$$

where $A = A^T$ or $A = A^H$.

Table 192. Data Types

<table>
<thead>
<tr>
<th>$w$, $rwork$</th>
<th>$A$, $Z$, $work$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SSPEVD$^A$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SSYEVD$^A$</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DSPEVD$^A$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DSYEVD$^A$</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>Short-precision complex</td>
<td>CHEEVD$^A$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZHEEVD$^A$</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision complex</td>
<td>ZHPEVD$^A$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZHEEVD$^A$</td>
</tr>
</tbody>
</table>

$^a$LAPACK
## Syntax

**Fortran**

<table>
<thead>
<tr>
<th>CALL SSPEVD</th>
<th>DSPEVD (jobz, uplo, n, ap, w, z, ldz, work, lwork, iwork, liwork, info)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL CHPEVD</td>
<td>ZHPEVD (jobz, uplo, n, ap, w, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)</td>
</tr>
<tr>
<td>CALL SSYEVD</td>
<td>DSYEVD (jobz, uplo, n, a, lda, w, work, lwork, iwork, liwork, info)</td>
</tr>
<tr>
<td>CALL CHEEVD</td>
<td>ZHEEVD (jobz, uplo, n, a, lda, w, work, lwork, rwork, lrwork, iwork, liwork, info)</td>
</tr>
</tbody>
</table>

**C and C++**

<table>
<thead>
<tr>
<th>sspevd</th>
<th>dspevd (jobz, uplo, n, ap, w, z, ldz, work, lwork, iwork, liwork, info);</th>
</tr>
</thead>
<tbody>
<tr>
<td>chpevd</td>
<td>zhpevd (jobz, uplo, n, ap, w, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info);</td>
</tr>
<tr>
<td>ssyevd</td>
<td>dsyevd (jobz, uplo, n, a, lda, w, work, lwork, iwork, liwork, info);</td>
</tr>
<tr>
<td>cheevd</td>
<td>zheevd (jobz, uplo, n, a, lda, w, work, lwork, rwork, lrwork, iwork, liwork, info);</td>
</tr>
</tbody>
</table>

**LAPACK**

<table>
<thead>
<tr>
<th>info = LAPACKSSPEVD</th>
<th>LAPACKDSPEVD (matrix_layout, jobz, uplo, n, ap, w, z, ldz);</th>
</tr>
</thead>
<tbody>
<tr>
<td>info = LAPACKCHPEVD</td>
<td>LAPACKZHPEVD (matrix_layout, jobz, uplo, n, ap, w, z, ldz);</td>
</tr>
<tr>
<td>info = LAPACKSYEVD</td>
<td>LAPACKDSYEV (matrix_layout, jobz, uplo, n, a, lda, w);</td>
</tr>
<tr>
<td>info = LAPACKCHEEV</td>
<td>LAPACKZHEEV (matrix_layout, jobz, uplo, n, a, lda, w);</td>
</tr>
</tbody>
</table>

### On Entry

**matrix_layout**

- Indicates whether the input and output matrices are stored in row major order or column major order, where:
  - If `matrix_layout` = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
  - If `matrix_layout` = LAPACK_COL_MAJOR, the matrices are stored in column major order.

  Specified as: an integer. It must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR

**jobz**

- Indicates the type of computation to be performed, where:
  - If `jobz` = 'N', eigenvalues only are computed.
  - If `jobz` = 'V', eigenvalues and eigenvectors are computed.

  Specified as: a single character; `jobz` = 'N' or 'V'.

**uplo**

- Indicates whether the upper or lower triangular part of the matrix $A$ is referenced, where:
  - If `uplo` = 'U', the upper triangular part is referenced.
  - If `uplo` = 'L', the lower triangular part is referenced.

  Specified as: a single character; `uplo` = 'U' or 'L'.

**n**

- Is the order of matrix $A$ used in the computation.

  Specified as: an integer; $n \geq 0$.

**ap**

- Is the real symmetric or complex Hermitian matrix $A$ of order $n$. It is stored in an array, referred to as $AP$, where:
If `uplo = 'U'`, it is stored in upper-packed storage.
If `uplo = 'L'`, it is stored in lower-packed storage mode.

Specified as: one-dimensional array of (at least) length \(n(n + 1)/2\), containing numbers of the data type indicated in Table 192 on page 978.

\(a\) is the real symmetric or complex Hermitian matrix \(A\) of order \(n\).

If `uplo = 'U'`, it is stored in upper storage mode.
If `uplo = 'L'`, 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 192 on page 978.

\(lda\) is the leading dimension of the array specified for \(A\).

Specified as: an integer; \(lda > 0\) and \(lda \geq n\).

\(w\) See On Return
\(z\) See On Return

\(ldz\) is the leading dimension of the array specified for \(Z\).

Specified as: an integer; \(ldz > 0\) and, if `jobz = 'V'`, \(ldz \geq n\).

\(work\)

is a work area used by these subroutines.

- If \(lwork = 0\) and \(liwork \neq -1\) and \(lrwork \neq -1\), \(work\) is ignored.
- If \(lwork \neq -1\), and \(liwork \neq -1\) and \(lrwork \neq -1\), \(work\) is (at least) of length \(lwork\).
- If \(lwork = -1\), or \(liwork = -1\), or \(lrwork = -1\), \(work\) is (at least) of length 1.

Specified as: an area of storage containing numbers of the data type indicated in Table 192 on page 978.

\(lwork\)

is used to determine the size of the \(WORK\) array.

Specified as: an integer, where:

- If \(lwork = 0\) and \(liwork \neq -1\) and \(lrwork \neq -1\), the subroutine dynamically allocates the workspace needed for use during this computation. The dynamically allocated workspace will be freed prior to returning control to the calling program.
- If \(lwork = -1\) or \(liwork = -1\) or \(lrwork = -1\), these subroutines perform a work area query for all work areas and return the optimal size of \(work\) in \(work_1\), and \(iwork\) in \(iwork_1\) and \(rwork\) in \(rwork_1\).

Otherwise:

- If \(n \leq 1\), \(lwork\) must be (at least) 1.
- If `jobz = 'N'` and \(n > 1\), \(lwork\) is as follows:

  For SSPEVD, DSPEVD
  \(lwork\) must be (at least) \(2n\)

  For SSYEVD, and DSYEVD
  \(lwork\) must be (at least) \(2n + 1\)

  For CHPEVD and ZHPEVD
  \(lwork\) must be (at least) \(n\)
For CHEEVD and ZHEEVD
lwork must be (at least) \( n + 1 \)

- If \( \text{jobz} = 'V' \) and \( n > 1 \), \( lwork \) is as follows:

  - For SSPEVD and DSPEVD
    \( lwork \) must be (at least) \( 1 + 6n + n^2 \)
  - For SSYEV and DSYEV
    \( lwork \) must be (at least) \( 1 + 6n + 2n^2 \)
  - For CHPEVD and ZHPEVD
    \( lwork \) must be (at least) \( 2n \)
  - For CHEEVD and ZHEEVD
    \( lwork \) must be (at least) \( 2n + n^2 \)

**Note:** These formulas represent the minimum workspace required. For best performance, specify either \( lwork = -1 \) (to obtain the optimal size to use) or \( lwork = 0 \) (to direct the subroutine to dynamically allocate the workspace).

**rwork**

has the following meaning:

- if \( lrwork = 0 \) and \( liwork \neq -1 \) and \( lwork \neq -1 \), \( rwork \) is ignored.
- If \( lrwork \neq -1 \), and \( liwork \neq -1 \) and \( lwork \neq -1 \), \( rwork \) is (at least) of length \( lrwork \).
- If \( lrwork = -1 \), or \( liwork = -1 \), or \( lwork = -1 \), \( rwork \) is (at least) of length 1.

Specified as: an area of storage containing real numbers of the data type indicated in Table 192 on page 978.

**lrwork**

is the number of elements in array \( rwork \).

Specified as: a fullword integer; where:

- If \( lrwork = 0 \) and \( liwork \neq -1 \) and \( lwork \neq -1 \), the subroutine dynamically allocates the workspace needed for use during this computation. The dynamically allocated workspace will be freed prior to returning control to the calling program.
- If \( lrwork = -1 \) or \( liwork = -1 \) or \( lwork = -1 \), these subroutines perform a work area query for all work areas and return the optimal size of \( work \) in \( work_1 \) and \( iwork \) in \( iwork_1 \) and \( rwork \) in \( rwork_1 \).
- Otherwise:
  - If \( n \leq 1 \), \( lrwork \) must be (at least) 1
  - If \( \text{jobz} = 'N' \) and \( n > 1 \), \( lrwork \) must be (at least) \( n \)
  - If \( \text{jobz} = 'V' \) and \( n > 1 \), \( lrwork \) must be (at least) \( 1 + 5n + 2n^2 \)

**iwork**

has the following meaning:

- if \( liwork = 0 \) and \( lrwork \neq -1 \) and \( lwork \neq -1 \), \( iwork \) is ignored.
- If \( liwork \neq -1 \), and \( lrwork \neq -1 \) and \( lwork \neq -1 \), \( iwork \) is (at least) of length \( liwork \).
- If \( liwork = -1 \), or \( lrwork = -1 \), or \( lwork = -1 \), \( iwork \) is (at least) of length 1.

Specified as: an area of storage containing fullword integers.

**liwork**

is the number of elements in array \( IWORK \).

Specified as: a fullword integer; where:
If \( \text{liwork} = 0 \) and \( \text{lrwork} \neq -1 \) and \( \text{lwork} \neq -1 \), the subroutine dynamically allocates the workspace needed for use during this computation. The dynamically allocated workspace will be freed prior to returning control to the calling program.

If \( \text{liwork} = -1 \) or \( \text{lrwork} = -1 \) or \( \text{lwork} = -1 \), these subroutines perform a work area query for all work areas and return the optimal size of \( \text{work} \) in \( \text{work}_1 \) and \( \text{iwork} \) in \( \text{iwork}_1 \) and \( \text{rwork} \) in \( \text{rwork}_1 \).

Otherwise:
- If \( n \leq 1 \), \( \text{liwork} \) must be (at least) 1
- If \( \text{jobz} = 'N' \) and \( n > 1 \), \( \text{liwork} \) must be (at least) 1
- If \( \text{jobz} = 'V' \) and \( n > 1 \), \( \text{lwork} \) must be (at least) \( 3 + 5n \)

On Return
- \( \text{ap} \) the matrix \( A \) is overwritten by values generated during the reduction to tridiagonal form.
- If \( \text{uplo} = 'U' \), the diagonal and first superdiagonal of the tridiagonal matrix \( T \) overwrite the corresponding elements of \( A \).
- If \( \text{uplo} = 'L' \), the diagonal and first subdiagonal of \( T \) overwrite the corresponding elements of \( A \).
  Returned as: a one-dimensional array of (at least) length \( n(n + 1)/2 \), containing numbers of the data type indicated in Table 192 on page 978.
- \( \text{a} \) If \( \text{jobz} = 'V' \) and \( \text{info} = 0 \), \( a \) contains the orthonormal eigenvectors corresponding to the computed eigenvalues, with the \( i \)-th column of \( A \) holding the eigenvector associated with \( w_i \).
  If \( \text{jobz} = 'V' \) and \( \text{info} \neq 0 \), no eigenvectors are valid.
  If \( \text{jobz} = 'N' \):
  - If \( \text{uplo} = 'U' \), the upper triangle of matrix \( A \) is overwritten.
  - If \( \text{uplo} = 'L' \), the lower triangle of matrix \( A \) is overwritten.
  Returned as: an array of dimension \( \text{lda} \) by (at least) \( n \), containing numbers of the data type indicated in Table 192 on page 978.
- \( \text{w} \) If \( \text{info} = 0 \), \( w \) is the vector \( w \), containing the computed eigenvalues in ascending order.
  If \( \text{info} \neq 0 \), no eigenvalues are valid.
  Returned as: a one-dimensional array of (at least) length \( n \), containing numbers of the data type indicated in Table 192 on page 978.
- \( \text{z} \) has the following meaning, where:
  If \( \text{jobz} = 'N' \), then \( z \) is ignored.
  If \( \text{jobz} = 'V' \) and \( \text{info} = 0 \), \( z \) contains the orthonormal eigenvectors corresponding to the computed eigenvalues, with the \( i \)-th column of \( z \) holding the eigenvector associated with \( w_i \).
  If \( \text{jobz} = 'V' \) and \( \text{info} \neq 0 \), no eigenvectors are valid.
  Returned as: an \( \text{ldz} \) by \( n \) array, containing numbers of the data type indicated in Table 192 on page 978.
- \( \text{work} \)
  is a work area used by these subroutines.
  Returned as: an area of storage where:
If \( lwork \geq 1 \) or \( lwork = -1 \) or \( liwork = -1 \) or \( lrwork = -1 \), then \( work_1 \) is set to the optimal \( lwork \) value and contains numbers of the data type indicated in Table 192 on page 978.

Except for \( work_1 \), the contents of \( work \) are overwritten on return.

**rwork**

is a work area used by these subroutines.

Returned as: an area of storage where:

If \( lrwork \geq 1 \) or \( lrwork = -1 \) or \( liwork = -1 \) or \( lwork = -1 \), then \( rwork_1 \) is set to the optimal \( lrwork \) value and contains numbers of the data type indicated in Table 192 on page 978.

Except for \( rwork_1 \), the contents of \( rwork \) are overwritten on return.

**iwork**

is a work area used by these subroutines.

Returned as: an area of storage where:

If \( liwork \geq 1 \) or \( liwork = -1 \) or \( lrwork = -1 \) or \( lwork = -1 \), then \( iwork_1 \) is set to the optimal \( liwork \) value and contains numbers of the data type indicated in Table 192 on page 978.

Except for \( iwork_1 \), the contents of \( iwork \) are overwritten on return.

**info**

has the following meaning:

If \( info = 0 \), then all eigenvalues converged. This indicates a normal exit.

If \( info = i \) and \( jobz = 'N' \), then the algorithm failed to converge. \( i \) indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

If \( info = i \) and \( jobz = 'V' \), then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns \( info / (n + 1) \) through \( \text{mod} (info, n + 1) \). No eigenvalues are valid.

Returned as:

- For SSPEVD, DSPEVD, CHPEVD, ZHPEVD, SSYEVD, DSYEVD, CHEEVD, and ZHEEVD returned as: an integer; \( info \geq 0 \).
- For LAPACKE sspevd, LAPACKE dspevd, LAPACKE chpevd, LAPACKE zhpevd, LAPACKE ssysevd, LAPACKE ssyevd, LAPACKE dsyevd, LAPACKE cheevd, and LAPACKE zheevd returned as an integer function value; \( info \geq 0 \).

**Notes**

1. This subroutine accepts lowercase letters for the \( jobz \) and \( uplo \) arguments.
2. In your C program, the argument \( info \) must be passed by reference.
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. \( A, Z, w, work, rwork, iwork \) must have no common elements; otherwise, results are unpredictable.
5. For a description of how real symmetric matrices are stored in lower- or upper-packed storage mode, see “Lower-Packed Storage Mode” on page 85 or “Upper-Packed Storage Mode” on page 87, respectively.
For a description of how complex Hermitian matrices are stored in lower- or upper-packed storage mode, see "Complex Hermitian Matrix" on page 90.

6. For a description of how real symmetric matrices are stored in lower or upper storage mode, see "Lower Storage Mode" on page 88 or "Upper Storage Mode" on page 89, respectively.

For a description of how complex Hermitian matrices are stored in lower or upper storage mode, see "Complex Hermitian Matrix" on page 90.

7. For best performance specify \textit{lw}ork = 0, \textit{li}work = 0, and \textit{lr}work = 0.

Function

These subroutines compute eigenvalues and, optionally, the eigenvectors of a real symmetric matrix or a complex Hermitian matrix.

If eigenvalues only are computed, these subroutines compute the eigenvalues using the Pal-Walker-Kahan variant of the QL or QR algorithm.

If eigenvectors are computed, the subroutine uses a divide-and-conquer method to compute them:

- SSPEVD and DSPEVD compute eigenvalues and, optionally, the eigenvectors of real symmetric matrix \(A\), stored in lower- or upper-packed storage mode.
- CHPEVD and ZHPEVD compute eigenvalues and, optionally, the eigenvectors of complex Hermitian matrix \(A\), stored in lower- or upper-packed storage mode.
- SSYEV and DSYEV compute eigenvalues and, optionally, the eigenvectors of real symmetric matrix \(A\), stored in lower or upper storage mode.
- CHEEVD and ZHEEVD compute eigenvalues and, optionally, the eigenvectors of complex Hermitian matrix \(A\), stored in lower or upper storage mode.

Eigenvalues are returned in vector \(w\) and eigenvectors are returned in matrix \(Z\) (for subroutines SSPEVD, DSPEVD, CHPEVD, ZHPEVD) or in matrix \(A\) (for subroutines SSYEV, DSYEV, CHEEVD, ZHEEVD):

\[
AZ = wZ
\]

where \(A = A^T\) or \(A = A^H\).

The computation involves the following steps:

1. If necessary, scale the matrix \(A\).
2. Reduce matrix \(A\) to tridiagonal form.
3. Compute the eigenvalues and, optionally, the eigenvectors of the symmetric tridiagonal matrix. The algorithm used depends on the value specified for \textit{jobz}:
   - If \textit{jobz} = 'N', compute all the eigenvalues using the Pal-Walker-Kahan variant of the QL or QR algorithms.
   - Otherwise, compute both the eigenvalues and eigenvectors using a divide-and-conquer algorithm, then apply Householder transformations to the eigenvector matrix.
4. Rescale eigenvalues appropriately if the matrix was scaled.

If \(n = 0\), the subroutine returns after completing parameter checking.

For more information on these methods, see references [8 on page 1363], [74 on page 1367], and [34 on page 1365].
Error conditions

Resource Errors
1. $lwork = 0$, and unable to allocate work area.
2. $lrwork = 0$, and unable to allocate work area.
3. $liwork = 0$, and unable to allocate work area.

Computational Errors
1. If $info = i$ and $jobz = 'N'$, then the algorithm failed to converge. $i$ indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.
2. If $info = i$ and $jobz = 'V'$, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns $info / (n + 1)$ through $\text{mod}(info, n + 1)$. No eigenvalues are valid.

Input-Argument Errors
1. $jobz \neq 'N'$ or 'V'
2. $uplo \neq 'U'$ or 'L'
3. $n < 0$
4. $lda \leq 0$
5. $n > lda$
6. $ldz \leq 0$
7. $n > ldz$ and $jobz = 'V'$
8. $lwork \neq 0$ and $lwork \neq -1$ and $liwork \neq -1$ and $lrwork \neq -1$, and $lwork <$ the minimum required value
9. $liwork \neq 0$ and $liwork \neq -1$ and $lwork \neq -1$ and $lrwork \neq -1$, and $liwork <$ the minimum required value
10. $lrwork \neq 0$ and $lrwork \neq -1$ and $liwork \neq -1$ and $lwork \neq -1$ and $lrwork <$ the minimum required value
11. The size of a work array is greater than 2147483647 when 32-bit integers are used.

Examples

Example 1

This example shows how to find the eigenvalues only of a real symmetric matrix of order 4, stored in lower-packed storage mode.

Notes:
1. Because $lwork = 0$, the subroutine dynamically allocates $WORK$.
2. Because $liwork = 0$, the subroutine dynamically allocates $IWORK$.
3. $Z$ is not used when $jobz = 'N'$.
4. On output, array $AP$ is overwritten.
5. This matrix is Example 4.1 in referenced text [74 on page 1367].

Matrix $A$ is:

\[
\begin{bmatrix}
5.0 & 4.0 & 1.0 & 1.0 \\
4.0 & 5.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 4.0 & 2.0 \\
1.0 & 1.0 & 2.0 & 4.0
\end{bmatrix}
\]

Call Statement and Input:
Example 2

This example shows how to find the eigenvalues and eigenvectors of a real symmetric matrix of order 4, stored in upper-packed storage mode.

Notes:
1. Because lwork = 0, the subroutine dynamically allocates WORK.
2. Because liwork = 0, the subroutine dynamically allocates IW0RK.
3. On output, array AP is overwritten.
4. This matrix is Example 4.1 in referenced text[74 on page 1367]

Matrix A is the same as in Example 1

Call Statement and Input:

\[
\text{JOBZ UPLD N AP W Z LDZ WORK LWORK IWORK LIWORK INFO} \\
\text{CALL DSPEVD ('N', 'U', 4, AP, W, Z, 4, WORK, 0, IW0RK, 0, INFO)} \\
\]

Output:

\[
W = \begin{bmatrix}
1.000000 \\
2.000000 \\
5.000000 \\
10.000000 \\
\end{bmatrix}
\]

INFO = 0

Example 3

This example shows how to find the eigenvalues only of a complex Hermitian matrix of order 3, stored in lower-packed storage mode.

Notes:
1. Because lwork = 0, the subroutine dynamically allocates WORK.
2. Because lrwork = 0, the subroutine dynamically allocates RWORK.
3. Because liwork = 0, the subroutine dynamically allocates IW0RK.
4. Z is not used when jobz = 'N'.
5. On output, array AP is overwritten.

\[
\text{JOBZ UPLO N AP W Z LDZ WORK LWORK IWORK LIWORK INFO} \\
\text{CALL DSPEVD ('V', 'U', 4, AP, W, Z, 4, WORK, 0, IW0RK, 0, INFO)} \\
\]

Output:

\[
W = \begin{bmatrix}
1.000000 \\
2.000000 \\
5.000000 \\
10.000000 \\
\end{bmatrix}
\]

\[
Z = \begin{bmatrix}
-0.707107 & 0.000000 & 0.316228 & -0.632456 \\
0.707107 & 0.000000 & 0.316228 & -0.632456 \\
0.000000 & -0.707107 & -0.632456 & -0.316228 \\
0.000000 & 0.707107 & -0.632456 & -0.316228 \\
\end{bmatrix}
\]

INFO = 0
6. This matrix is Example 4.1 in referenced text [74 on page 1367].

Matrix \( A \) is:

\[
\begin{pmatrix}
(2.0, 0.0) & (0.0, 1.0) & (0.0, 0.0) \\
(0.0, -1.0) & (2.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (0.0, 0.0) & (3.0, 0.0)
\end{pmatrix}
\]

Call Statement and Input:

```
JOBZ UPLO N AP W Z LDZ WORK LWORK
| | | | | | | |
CALL ZHPEVD ('N', 'L', 3, AP, W, Z, 3, WORK, 0,
RWORK LRWORK IWORK LIWORK INFO
| | | | | | | |
RWORK, 0, IWORK, 0, INFO)
```

\( AP = ((2.0, .), (0.0, -1.0), (0.0, 0.0), \)
(2.0, .), (0.0, 0.0), (3.0, .)) \)

Output:

\[
W = \begin{bmatrix}
1.000000 \\
3.000000 \\
3.000000
\end{bmatrix}
\]

\( INFO = 0 \)

**Example 4**

This example shows how to find the eigenvalues and eigenvectors of a complex Hermitian matrix of order 3, stored in upper-packed storage mode.

**Notes:**
1. Because \( lwork = 0 \), the subroutine dynamically allocates WORK.
2. Because \( lrwork = 0 \), the subroutine dynamically allocates RWORK.
3. Because \( liwork = 0 \), the subroutine dynamically allocates IWORK.
4. On output, array \( AP \) is overwritten.
5. This matrix is Example 4.1 in referenced text [74 on page 1367].

Matrix \( A \) is the same as in Example 3

Call Statement and Input:

```
JOBZ UPLO N AP W Z LDZ WORK LWORK
| | | | | | | |
CALL ZHPEVD ('Y', 'U', 3, AP, W, Z, 3, WORK, 0,
RWORK LRWORK IWORK LIWORK INFO
| | | | | | | |
RWORK, 0, IWORK, 0, INFO)
```

\( AP = ((2.0, .), (0.0, -1.0), (0.0, 0.0), (2.0, .), (0.0, 0.0), (3.0, .)) \)

Output:

\[
W = \begin{bmatrix}
1.000000 \\
3.000000 \\
3.000000
\end{bmatrix}
\]

\[
Z = \begin{bmatrix}
(-0.7071, 0.0) & (-0.7071, 0.0) & (0.0, 0.0) \\
(0.0, -0.7071) & (0.0, 0.7071) & (0.0, 0.0) \\
(0.0, 0.0) & (0.0, 0.0) & (1.0, 0.0)
\end{bmatrix}
\]

\( INFO = 0 \)
Example 5

This example shows how to find the eigenvalues only of a real symmetric matrix of order 4, stored in lower storage mode.

Notes:
1. Because lwork = 0, the subroutine dynamically allocates WORK.
2. Because liwork = 0, the subroutine dynamically allocates IWORK.
3. On output, array A is overwritten.
4. This matrix is Example 4.1 in referenced text [74 on page 1367]

Matrix A is:

\[
\begin{bmatrix}
5.0 & 4.0 & 1.0 & 1.0 \\
4.0 & 5.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 4.0 & 2.0 \\
1.0 & 1.0 & 2.0 & 4.0 \\
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
1.000000 \\
2.000000 \\
5.000000 \\
10.000000 \\
\end{bmatrix}
\]

INFO = 0

Example 6

This example shows how to find the eigenvalues and eigenvectors of a real symmetric matrix of order 4, stored in upper storage mode.

Notes:
1. Because lwork = 0, the subroutine dynamically allocates WORK.
2. Because liwork = 0, the subroutine dynamically allocates IWORK.
3. This matrix is Example 4.1 in referenced text [74 on page 1367]

Matrix A is the same as in Example 5

Call Statement and Input:

```fortran
CALL DSYEVD ('N', 'L', 4, A, 4, W, WORK, 0, IWORK, 0, INFO)
```

```fortran
A = [ 5.0  4.0  1.0  1.0 ]
   [ 5.0  1.0  1.0  1.0 ]
   [ 1.0  4.0  2.0  1.0 ]
   [ 1.0  2.0  4.0 ]
```

```fortran
CALL DSYEVD ('V', 'U', 4, A, 4, W, WORK, 0, IWORK, 0, INFO)
```

```fortran
A = [ 5.0  4.0  1.0  1.0 ]
   [ 5.0  1.0  1.0  1.0 ]
   [ 4.0  2.0  4.0 ]
```
Example 7

This example shows how to find the eigenvalues only of a complex Hermitian matrix of order 3, stored in lower storage mode.

Notes:
1. Because \( lwork = 0 \), the subroutine dynamically allocates \( WORK \).
2. Because \( lrwork = 0 \), the subroutine dynamically allocates \( RWORK \).
3. Because \( liwork = 0 \), the subroutine dynamically allocates \( IWORK \).
4. On output, array \( A \) is overwritten.
5. This matrix is Example 4.1 in referenced text \([74 \text{ on page 1367}]\).

Matrix \( A \) is:
\[
\begin{pmatrix}
(2.0, 0.0) & (0.0, 1.0) & (0.0, 0.0) \\
(0.0, -1.0) & (2.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (0.0, 0.0) & (3.0, 0.0)
\end{pmatrix}
\]

Call Statement and Input:

\[
\begin{array}{cccccccc}
\text{JOBZ} & \text{UPLO} & \text{N} & \text{A} & \text{LDA} & \text{W} & \text{WORK} & \text{LWORK} & \text{INFO} \\
\text{CALL} & \text{ZHHEVD} (\text{'N'}, \text{'L'}, 3, \text{A}, 3, \text{W}, \text{WORK}, 0, \text{RWORK}, 0, \text{IWORK}, 0, \text{INFO})
\end{array}
\]

Output:

\[
A = \begin{pmatrix}
(2.0, .) & . & . \\
(0.0,-1.0) & (2.0, .) & . \\
(0.0, 0.0) & (0.0, 0.0) & (3.0, .)
\end{pmatrix}
\]

Example 8

This example shows how to find the eigenvalues and eigenvectors of a complex Hermitian matrix \( A \) of order 3, stored in upper storage mode.

Notes:
1. Because \( lwork = 0 \), the subroutine dynamically allocates \( WORK \).
2. Because \( lrwork = 0 \), the subroutine dynamically allocates \( RWORK \).
3. Because \( liwork = 0 \), the subroutine dynamically allocates \( lwork \).

4. This matrix is Example 4.1 in referenced text \([74 on page 1367]\).

Matrix \( A \) is the same as in \( \text{Example 7} \).

Call Statement and Input:

\[
\begin{array}{cccccccc}
\text{JOBZ} & \text{UPLO} & \text{N} & \text{A} & \text{LDA} & \text{WORK} & \text{LWORK} & \text{INFO} \\
\hline
\text{CALL ZHEEVD ('V', 'U', 3, A, 3, WORK, 0, RWORK, 0, IWORK, 0, INFO)} \\
\end{array}
\]

\[
A = \begin{bmatrix}
  (2.0, .) & (0.0, 1.0) & (0.0, 0.0) \\
  (0.0, .) & (2.0, .) & (0.0, 0.0) \\
  (. , .) & (. , .) & (3.0, .)
\end{bmatrix}
\]

Output:

\[
A = \begin{bmatrix}
  (-0.7071, 0.0) & (-0.7071, 0.0) & (0.0, 0.0) \\
  (0.0, -0.7071) & (0.0, 0.7071) & (0.0, 0.0) \\
  (0.0, 0.0) & (0.0, 0.0) & (1.0, 0.0)
\end{bmatrix}
\]

\[
W = \begin{bmatrix}
  1.000000 \\
  3.000000 \\
  3.000000
\end{bmatrix}
\]

INFO = 0
SGGEV, DGGEV, CGGEV, ZGGEV, SGGEVX, DGGEVX, CGGEVX, and ZGGEVX (Eigenvalues and, Optionally, Right Eigenvectors, Left Eigenvectors, Reciprocal Condition Numbers for Eigenvalues, and Reciprocal Condition Numbers for Right Eigenvectors of a General Matrix Generalized Eigenproblem)

Purpose

- SGGEV, DGGEV, CGGEV, and ZGGEV compute the eigenvalues and, optionally, the left and/or right eigenvectors of a general matrix generalized eigenproblem.
- SGGEVX, DGGEVX, CGGEVX, and ZGGEVX compute the eigenvalues and, optionally, right eigenvectors, left eigenvectors, reciprocal condition numbers for eigenvalues, and reciprocal condition numbers for right eigenvectors of a general matrix generalized eigenproblem.

For the left eigenvectors:

\[ \mathbf{vl}^H \mathbf{A} = \lambda \mathbf{vl}^H \mathbf{B} \]

For the right eigenvectors:

\[ \mathbf{Avr} = \lambda \mathbf{Bvr} \]

The eigenvalues are returned in two parts, \( \alpha \) and \( \beta \), where:

For SGGEV, DGGEV, SGGEVX, and DGGEVX, \( \alpha \) and \( \beta \) are returned in vectors \( \mathbf{alphar} \), \( \mathbf{alphai} \), and \( \mathbf{beta} \), where \( \mathbf{alphar} \) contains the real part of \( \alpha \) and \( \mathbf{alphai} \) contains the imaginary part of \( \alpha \).
- If \( \mathbf{alphai} \) = 0, then the \( j \)-th eigenvalue is real.
- If \( \mathbf{alphai} > 0 \), then the \( j \)-th and \( (j+1) \)-th eigenvalues are a complex conjugate pair.

For CGGEV, ZGGEV, CGGEVX, and ZGGEVX, \( \alpha \) and \( \beta \) are returned in vectors \( \mathbf{alpha} \) and \( \mathbf{beta} \).

For SGGEV, DGGEV, SGGEVX, and DGGEVX:
- If \( \mathbf{alphai} = 0 \), then the \( j \)-th eigenvalue is real:
  - \( \alpha_j = \mathbf{alphar}_j \)
  - \( \beta_j = \mathbf{beta}_j \)
- If \( \mathbf{alphai} > 0 \), then the \( j \)-th and \( (j+1) \)-th eigenvalues \( \alpha_{j+1} \) are a complex conjugate pair:
  - \( \alpha_j = (\mathbf{alphar}_j, \mathbf{alphai}_j) \)
  - \( \beta_j = \mathbf{beta}_j \)
  - \( \alpha_{j+1} = (\mathbf{alphar}_{j+1}, -\mathbf{alphai}_j) \)
  - \( \beta_{j+1} = \mathbf{beta}_{j+1} \)

For CGGEV, ZGGEV, CGGEVX, and ZGGEVX:
- \( \alpha_j = \mathbf{alpha}_j \)
- \( \beta_j = \mathbf{beta}_j \)

Left eigenvectors are returned in matrix \( \mathbf{VL} \) and right eigenvectors are returned in matrix \( \mathbf{VR} \).
Table 193. Data Types

<table>
<thead>
<tr>
<th>A, B, VL, VR, bwork, work α, β</th>
<th>alphas, alphas, lscale, rscale, abnorm, rconde, rconde, rwork</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SGGEV^a</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DGGEV^a</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>CGGEV^a</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>ZGGEV^a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LAPACK</td>
</tr>
</tbody>
</table>

^aLAPACK

Syntax

Fortran

CALL SGGEV | DGGEV (jobvl, jobvr, n, a, lda, b, ldb, alphas, alphas, beta, vl, ldvl, vr, ldvr, work, lwork, info)

CALL CGGEV | ZGGEV (jobvl, jobvr, n, a, lda, b, ldb, alphas, alphas, beta, vl, ldvl, vr, ldvr, work, lwork, rwork, info)

CALL SGGEVX | DGGEVX (balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alphas, alphas, beta, vl, ldvl, vr, ldvr, info)

CALL CGGEVX | ZGGEVX (balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alphas, alphas, beta, vl, ldvl, vr, ldvr, info)

C and C++

sggev | dggev (jobvl, jobvr, n, a, lda, b, ldb, alphas, alphas, beta, vl, ldvl, vr, ldvr, work, lwork, info);

cggev | zggev (jobvl, jobvr, n, a, lda, b, ldb, alphas, alphas, beta, vl, ldvl, vr, ldvr, work, rwork, info);

sggevx | dggevx (balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alphas, alphas, beta, vl, ldvl, vr, ldvr, info);

cggevx | zggevx (balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alphas, alphas, beta, vl, ldvl, vr, ldvr, info);

LAPACK

info = LAPACKE_sggev | LAPACKE_dggev (matrix_layout, jobvl, jobvr, n, a, lda, b, ldb, alphas, alphas, beta, vl, ldvl, vr, ldvr);

info = LAPACKE_cggev | LAPACKE_zggev (matrix_layout, jobvl, jobvr, n, a, lda, b, ldb, alphas, alphas, beta, vl, ldvl, vr, ldvr);

info = LAPACKE_sggevx | LAPACKE_dggevx (matrix_layout, balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alphas, alphas, beta, vl, ldvl, vr, ldvr, info);

info = LAPACKE_cggevx | LAPACKE_zggevx (matrix_layout, balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alphas, alphas, beta, vl, ldvl, vr, ldvr, info);

On Entry

matrix_layout

indicates whether the input and output matrices are stored in row major order or column major order, where:
• If `matrix_layout` = LAPACK_ROW_MAJOR, the matrices are stored in row
  major order.

• If `matrix_layout` = LAPACK_COL_MAJOR, the matrices are stored in column
  major order.

Specified as: an integer. It must be LAPACK_ROW_MAJOR or
LAPACK_COL_MAJOR

`balance` indicates whether or not to scale **A** diagonally and whether or not to permute
its rows and columns to improve the conditioning of its eigenvalues, where
`balance` can have any of the following values:

- N  Neither diagonally scale nor permute **A**.
- P  Permute **A**, but do not diagonally scale it.
- S  Diagonally scale **A**, but do not permute it.
- B  Both diagonally scale and permute **A**.

When diagonal scaling is specified, the subroutine replaces **A** with $D^{-1}AD$
where **D** is a diagonal matrix chosen to make the rows and columns of **A** more
equal in norm and the condition numbers of its eigenvalues and eigenvectors
smaller.

When permuting is specified, the subroutine makes **A** more nearly upper
triangular.

The computed reciprocal condition numbers correspond to the balanced
matrix. In exact arithmetic, permuting rows and columns does not change the
condition numbers, but diagonal scaling does change the condition numbers.

Specified as: a single character. It must be 'N', 'P', 'S', or 'B'.

`jobvl` indicates the type of computation to be performed, where:

- If `jobvl` = 'N', the left eigenvectors are not computed.
- If `jobvl` = 'V', the left eigenvectors are computed.

Specified as: a single character. It must be 'N' or 'V'.

`jobvr` indicates the type of computation to be performed, where:

- If `jobvr` = 'N', the right eigenvectors are not computed.
- If `jobvr` = 'V', the right eigenvectors are computed.

Specified as: a single character. It must be 'N' or 'V'.

`sense` indicates which reciprocal numbers to compute (if any), where `sense` can have
any of the following values:

- N  Do not compute reciprocal condition numbers.
- E  Compute reciprocal condition numbers for eigenvalues only.
- V  Compute reciprocal condition numbers for right eigenvectors only.
- B  Compute reciprocal condition numbers for eigenvalues and right
eigenvectors.

Specified as: a single character. It must be 'N', 'E', 'V', or 'B'.

Chapter 11. Eigensystem Analysis 993
\( n \) is the order of the general matrices \( A \) and \( B \).

Specified as: an integer; \( n \geq 0 \).

\( a \) is the general matrix \( A \) of order \( n \).

Specified as: an \( lda \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 193 on page 992

\( lda \)

is the leading dimension of the array specified for matrix \( A \).

Specified as: an integer; \( lda > 0 \) and \( lda \geq n \).

\( b \) is the general matrix \( B \) of order \( n \).

Specified as: an \( ldb \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 193 on page 992

\( ldb \)

is the leading dimension of the array specified for matrix \( B \).

Specified as: an integer; \( ldb > 0 \) and \( ldb \geq n \).

\( \text{alphar} \)

See On Return

\( \text{alphai} \)

See On Return

\( \text{alpha} \)

See On Return

\( \text{beta} \)

See On Return

\( vl \) See On Return

\( ldvl \)

is the leading dimension of the array specified for \( vl \).

Specified as: an integer; \( ldvl > 0 \); if \( jobvl = 'V' \), \( ldvl \geq n \).

\( vr \) See On Return

\( ldvr \)

is the leading dimension of the array specified for \( vr \).

Specified as: an integer; \( ldvr > 0 \); if \( jobvr = 'V' \), \( ldvr \geq n \).

\( \text{ilo} \)

See On Return

\( \text{thi} \)

See On Return

\( lscale \)

See On Return

\( rscale \)

See On Return

\( abnrm \)

See On Return

\( bbnrm \)

See On Return
$rconde$

See On Return

$rcondv$

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 193 on page 992.

$lwork$

is the number of elements in array WORK.

Specified as an integer, where:

- If $lwork = 0$, the subroutine dynamically allocates the workspace needed for
  use during this computation. The dynamically allocated workspace will be
  freed prior to returning control to the calling program.

- If $lwork = -1$, a workspace query is assumed. The subroutine only calculates
  the optimal size of the WORK array and returns this value as the first entry of
  the WORK array.

Otherwise:

- For SGGEV and DGGEV:
  \[ lwork \geq \max(1, 8n) \]

- For CGGEV and ZGGEV:
  \[ lwork \geq \max(1, 2n) \]

- For SGGEVX and DGGEVX:
  \begin{itemize}
    \item If $sense = 'N'$:
      \begin{itemize}
        \item If $balanc = 'N'$ or 'P' and $jobvl = 'N'$ and $jobvr = 'N'$, $lwork \geq \max(1,2n)$;
        \item Otherwise, $lwork \geq \max(1,6n)$
      \end{itemize}
    \item If $sense = 'S'$ or 'B' or if $jobvl = 'V'$ or $jobvr = 'V'$, $lwork \geq \max(1, 6n)$
    \item If $sense = 'N'$, $lwork \geq \max(1, 2n)$
    \item If $sense = 'E'$, $lwork \geq \max(1, 10n)$
    \item If $sense = 'V'$ or 'B', $lwork \geq 2n^2+8n+16$
  \end{itemize}

- For CGGEVX and ZGGEVX:
  \begin{itemize}
    \item If $sense = 'N'$, $lwork \geq \max(1, 2n)$
    \item If $sense = 'E'$, $lwork \geq \max(1, 4n)$
    \item If $sense = 'V'$ or 'B', $lwork \geq \max(1,2n(n+1))$
  \end{itemize}

These formulas represent the minimum workspace required. For best
performance, specify either $lwork = -1$ (to obtain the optimal size to use) or
$lwork = 0$ (to direct the subroutine to dynamically allocate the workspace).

$rwork$

For CGGEV, and ZGGEV, $rwork$ is a storage work area of size $8n$.

For CGEEVX, and ZGGEVX, if $balanc = 'S'$ or 'B', $rwork$ must be at least
$max(1,6n)$. Otherwise, $rwork$ is of size $max(1,2n)$.

Specified as: an area of storage containing numbers of the data type indicated
in Table 193 on page 992.

$iwork$

For SGGEVX and DGGEVX, $iwork$ is a storage work area of $n+6$.

For CGEEVX and ZGGEVX, $iwork$ is a storage work area of size $2n-2$. 
If \( \text{sense} = 'N' \) or \( 'E' \), \( iwork \) is not referenced by the subroutine.

Specified as: an area of storage containing logical values of the data type indicated in [Table 193 on page 992](#).

\( bwork \)

is a storage work area of size \( n \). If \( \text{sense} = 'N' \), \( bwork \) is not referenced by the subroutine.

Specified as: an area of storage containing numbers of the data type indicated in [Table 193 on page 992](#).

On Return

\( a \)

is the updated general matrix \( A \) of order \( n \). On output, \( A \) is overwritten; that is, the original input is not preserved.

- For \( \text{SGGEVX, DGGEVX, CGGEVX, and ZGGEVX} \), if \( \text{jobvl = 'V' or jobvr = 'V'} \) and \( \text{balanc = 'N'} \), \( A \) contains the Schur form of the balanced matrices.

Returned as: an \( \text{lda by (at least)} n \) array, containing numbers of the data type indicated in [Table 193 on page 992](#).

\( b \)

is the updated general matrix \( B \) of order \( n \). On output, \( B \) is overwritten; that is, the original input is not preserved.

- For \( \text{SGGEVX, DGGEVX, CGGEVX, and ZGGEVX} \), if \( \text{jobvl = 'V' or jobvr = 'V'} \) and \( \text{balanc = 'N'} \), \( B \) contains the Schur form of the balanced matrices.

Returned as: an \( \text{lda by (at least)} n \) array, containing numbers of the data type indicated in [Table 193 on page 992](#).

\( \text{alphar} \)

is the vector of length \( n \), containing the real part of the numerators of the eigenvalues. For details, see “Function” on page 999.

Returned as: an array of (at least) length \( n \), containing numbers of the data type indicated in [Table 193 on page 992](#).

\( \text{alphai} \)

is the vector of length \( n \), containing the imaginary part of the numerators of the eigenvalues. For details, see “Function” on page 999.

Returned as: an array of (at least) length \( n \), containing numbers of the data type indicated in [Table 193 on page 992](#).

\( \text{alpha} \)

is the vector \( \alpha \) of length \( n \), containing the numerators of the eigenvalues. For details, see “Function” on page 999.

Returned as: an array of (at least) length \( n \), containing numbers of the data type indicated in [Table 193 on page 992](#).

\( \text{beta} \)

is the vector \( \beta \) of length \( n \), containing the denominators of the eigenvalues. For details, see “Function” on page 999.

Returned as: an array of (at least) length \( n \), containing numbers of the data type indicated in [Table 193 on page 992](#).

\( \text{vl} \)

contains the left eigenvectors.

- For \( \text{SGGEV, DGGEV, SGGEVX, and DGGEVX} \):
  - If \( \text{jobvl = 'V'} \), the left eigenvectors are stored in the columns of \( \text{vl} \), in the same order as their eigenvalues.
If the $j$-th eigenvalue is real or complex, then the $j$-th column of $vl$ contains its eigenvector.

- If the $j$-th and $(j+1)$-th eigenvalues form a complex conjugate pair, then the $j$-th and $(j+1)$-th columns of $vl$ contain the real and imaginary parts of the eigenvector corresponding to the $j$-th eigenvalue. The conjugate of this eigenvector is the eigenvector for the $(j+1)$-th eigenvalue.

- If $jobvl = 'N'$, $vl$ is not referenced.

- For CGGEV, ZGGEV, CGGEVX, and ZGGEVX:
  - If $jobvl = 'V'$, the left eigenvectors are stored in the columns of $vl$, in the same order as their eigenvalues.
  - If $jobvl = 'N'$, $vl$ is not referenced.

Returned as: an array of size $(ldvl, n)$ containing numbers of the data type indicated in Table 193 on page 992.

$vr$ contains the right eigenvectors.

- For SGGEV, DGGEV, SGGEVX, and DGGEVX:
  - If $jobvr = 'V'$, the right eigenvectors are stored in the columns of $vr$, in the same order as their eigenvalues.
  - If the $j$-th eigenvalue is real or complex, then the $j$-th column of $vr$ contains its eigenvector.
  - If the $j$-th and $(j+1)$-th eigenvalues form a complex conjugate pair, then the $j$-th and $(j+1)$-th columns of $vr$ contain the real and imaginary parts of the eigenvector corresponding to the $j$-th eigenvalue. The conjugate of this eigenvector is the eigenvector for the $(j+1)$-th eigenvalue.
  - If $jobvr = 'N'$, $vr$ is not referenced.

- For CGGEV, ZGGEV, CGGEVX, and ZGGEVX:
  - If $jobvr = 'V'$, the right eigenvectors are stored in the columns of $vr$, in the same order as their eigenvalues.
  - If $jobvr = 'N'$, $vr$ is not referenced.

Returned as: an array of size $(ldvr, n)$ containing numbers of the data type indicated in Table 193 on page 992.

$ilo$ has the following meaning:

If $balanc = 'N'$, $ilo = 1$.

Otherwise, the value of $ilo$ is determined when $A$ is balanced.

The balanced $a_{ij} = 0$ if $i > j$ and $j = 1, ..., (ilo-1)$ or $i = (ihi+1), ..., n$.

Returned as: an integer; $1 \leq ilo \leq n$.

$ihi$ has the following meaning:

If $balanc = 'N'$, $ihi = n$.

Otherwise, the value of $ihi$ is determined when $A$ is balanced.

The balanced $a_{ij} = 0$ if $i > j$ and $j = 1, ..., (ilo-1)$ or $i = (ihi+1), ..., n$.

Returned as: an integer; $1 \leq ihi \leq n$.

$lscale$ contains the details of the permutations and scaling factors applied to the left side when balancing $A$ and $B$. 
If $pl_j$ is the index of the row and column interchanged with row and column $j$, and $dl_j$ is the scaling factor applied to row and column $j$, then:

- $lscale_j = pl_j$ for $j = 1, ..., (ilo-1)$
- $lscale_j = dl_j$ for $j = ilo, ..., ihi$
- $lscale_j = pl_j$ for $j = (ihi+1), ..., n$

Returned as: a one-dimensional array of (at least) length $n$ containing numbers of the data type indicated in Table 193 on page 992.

$rscale$
contains the details of the permutations and scaling factors applied to the right side when balancing $A$ and $B$.

If $pr_j$ is the index of the row and column interchanged with row and column $j$, and $dr_j$ is the scaling factor applied to row and column $j$, then:

- $rscale_j = pr_j$ for $j = 1, ..., (ilo-1)$
- $rscale_j = dr_j$ for $j = ilo, ..., ihi$
- $rscale_j = pr_j$ for $j = (ihi+1), ..., n$

Returned as: a one-dimensional array of (at least) length $n$ containing numbers of the data type indicated in Table 193 on page 992.

$abnrm$
is the one-norm of the balanced matrix (the maximum of the sum of absolute values of elements of any column).

Returned as: a number of the data type indicated in Table 193 on page 992; $abnrm \geq 0$.

$bbnrm$
is the one-norm of the balanced matrix (the maximum of the sum of absolute values of elements of any column).

Returned as: a number of the data type indicated in Table 193 on page 992; $bbnrm \geq 0$.

$rconde$
contains the computed reciprocal condition numbers of the eigenvalues, where $rconde_j$ is the reciprocal condition number of the $j$th eigenvalue.

Returned as: an array of dimension $n$ containing numbers of the data type indicated in Table 193 on page 992.

$rcondv$
contains the computed reciprocal condition numbers of the eigenvectors, where $rcondv_j$ is the reciprocal condition number of the $j$th right eigenvector.

Returned as: an array of dimension $n$ containing numbers of the data type indicated in Table 193 on page 992.

$work$
is the work area used by this subroutine if $lwork \neq 0$, where:

If $lwork \neq 0$ and $lwork \neq -1$, its size is (at least) of length $lwork$.

If $lwork = -1$, its size is (at least) of length 1.

Returned as: an area of storage, where:

If $lwork \geq 1$ or $lwork = -1$, then $work_1$ is set to the optimal $lwork$ value and contains numbers of the data type indicated in Table 193 on page 992.

Except for $work_1$, the contents of $work$ are overwritten on return.
info
If info = 0, the subroutine completed successfully.

If 1 ≤ info ≤ n, the QZ algorithm failed to compute all the eigenvalues, and no
eigenvectors were computed. However:

- For SGGEV, DGGEV, SGGEVX, DGGEVX, LAPACKE_sggev,
  LAPACKE_dggev, LAPACKE_sggev, and LAPACKE_dggev,
  alphas and betas are valid for j = (info+1,...,n)
- For CGGEV, ZGGEV, CGGEVX, ZGGEVX, LAPACKE_cggev,
  LAPACKE_zggev, LAPACKE_cggev, and LAPACKE_zggev,
  alphas and betas are valid for j = (info+1,...,n)

If info = n + 1, the eigenvalues failed to converge in the computation of shifts.
If info = n + 2, the eigenvectors failed to converge because the 2-by-2 block did
not have a complex eigenvalue.

Retained as:

- For SGGEV, DGGEV, CGGEV, ZGGEV, SGGEVX, DGGEVX, and ZGGEVX,
  returned as: an integer; info ≥ 0.
- For LAPACKE_sggev, LAPACKE_dggev, LAPACKE_cggev, LAPACKE_zggev,
  LAPACKE_sggev, LAPACKE_dggev, LAPACKE_cggev, and
  LAPACKE_zggev returned as an integer function value; info ≥ 0.

Notes
1. The vectors and matrices used in the computation must have no common
elements; otherwise, results are unpredictable.
2. These subroutines accept lowercase letters for the balanc, jobvl, jobvr, and sense
   arguments.
3. When you specify jobvl = 'N', you must specify a dummy argument for vl.
4. When you specify jobvr = 'N', you must specify a dummy argument for vr.
5. These subroutines accept lowercase letters for the jobvl and jobvr arguments.
6. When you specify sense = 'N', you must specify a dummy argument for rconde.
7. When you specify sense = 'N' or 'E', you must specify dummy arguments for
   rconde and iwork.
8. In your C program, the ilo, ihi, abnrn, info arguments must be passed by reference.
9. In your C program, the info arguments must be passed by reference.
10. For best performance, specify lwork = 0.
11. The eigenvalue quotients might easily over- or underflow, and β might be
    zero. However, α is always less than and usually comparable with NORM(A)
    in magnitude, and β is always less than and usually comparable with
    NORM(B) in magnitude.

Function
- SGGEV, DGGEV, CGGEV, and ZGGEV compute the eigenvalues and, optionally,
  the left and/or right eigenvectors of a general matrix generalized eigenproblem.
- SGGEVX, DGGEVX, CGGEVX, and ZGGEVX compute the eigenvalues and,
  optionally, right eigenvectors, left eigenvectors, reciprocal condition numbers for
  eigenvalues, and reciprocal condition numbers for right eigenvectors of a general
  matrix generalized eigenproblem.

For the left eigenvectors:
$vlHA = \lambda vlHB$

For the right eigenvectors:

$Avr = \lambda Bvr$

The eigenvalues are returned in two parts, $\alpha$ and $\beta$, where:

For SGGEV, DGGEV, SGGEVX, and DGGEVX, $\alpha$ and $\beta$ are returned in vectors $\text{alphar}$, $\text{alpha}$, and $\text{beta}$, where $\text{alphar}$ contains the real part of $\alpha$ and $\text{alpha}$ contains the imaginary part of $\alpha$.

- If $\text{alphai}_j = 0$, then the $j$-th eigenvalue is real.
- If $\text{alphai}_j > 0$, then the $j$-th and $(j+1)$-th eigenvalues are a complex conjugate pair.

For CGGEV, ZGGEV, CGGEVX, and ZGGEVX, $\alpha$ and $\beta$ are returned in vectors $\text{alpha}$ and $\text{beta}$.

For SGGEV, DGGEV, SGGEVX, and DGGEVX:

- If $\text{alphai}_j = 0$, then the $j$-th eigenvalue is real:
  - $\alpha_j = \text{alphar}_j$
  - $\beta_j = \text{beta}_j$
- If $\text{alphai}_j > 0$, then the $j$-th and $(j+1)$-th eigenvalues $\alpha_{j+1}$ are a complex conjugate pair:
  - $\alpha_j = (\text{alphar}_j, \text{alpha}_j)$
  - $\beta_j = \text{beta}_j$
  - $\alpha_{j+1} = (\text{alphar}_{j+1}, -\text{alpha}_{j+1})$
  - $\beta_{j+1} = \text{beta}_{j+1}$

For CGGEV, ZGGEV, CGGEVX, and ZGGEVX:

- $\alpha_j = \text{alpha}_j$
- $\beta_j = \text{beta}_j$

Left eigenvectors are returned in matrix $VL$ and right eigenvectors are returned in matrix $VR$.

For SGGEV, DGGEV, CGGEV, and ZGGEV, the computation involves the following steps:

1. If necessary, scale the matrices $A$ and $B$.
2. Balance the matrices $A$ and $B$.
3. Reduce the balanced matrix $A$ to an upper Hessenberg matrix and reduce the balanced matrix $B$ to an upper triangular form.
4. Compute the eigenvalues of the Hessenberg-triangular pair, using the QZ algorithm.
5. If desired, compute the eigenvectors.
6. Undo balancing.
7. If necessary, undo scaling.

For SGGEVX, DGGEVX, CGGEVX, and ZGGEVX, the computation involves the following steps:

1. If necessary, scale the matrices $A$ and $B$.
2. Balance the matrices $A$ and $B$. 
3. Reduce the balanced matrix $A$ to triangular form.
4. Apply the unitary transformation to matrix $A$.
5. Reduce the matrix pair $(A,B)$ to generalized Hessenberg form.
6. Compute the eigenvalues and eigenvectors using the QZ algorithm and estimate condition numbers, if desired.
7. Undo balancing.
8. If necessary, undo scaling.

If $n = 0$, no computation is performed and the subroutine returns after doing some parameter checking.

For more information, see references [14 on page 1364], [15 on page 1364], [47 on page 1366], [51 on page 1366], [55 on page 1366], [65 on page 1367], [66 on page 1367], [69 on page 1367], [73 on page 1367], [74 on page 1367], [85 on page 1368], [86 on page 1368], [100 on page 1369], [106 on page 1369], [116 on page 1370], [118 on page 1370], and [119 on page 1370].

**Error conditions**

**Resource Errors**

$lwork = 0$, and unable to allocate work area.

**Computational Errors**

1. If $1 \leq info \leq n$, the QZ algorithm failed to compute all the eigenvalues, and no eigenvectors were computed.
2. If $info = n + 1$, the eigenvalues failed to converge in the computation of shifts.
3. If $info = n + 2$, the eigenvectors failed to converge because the 2-by-2 block did not have a complex eigenvalue.

**Input-Argument Errors**

1. $\text{balanc} \neq 'N', 'S', 'P$, or 'B'  
2. $\text{jobvl} \neq 'N'$, or 'V'  
3. $\text{jobvr} \neq 'N'$, or 'V'  
4. $\text{sense} \neq 'N', 'E$, 'V', or 'B'  
5. $n < 0$  
6. $\text{lda} \leq 0$  
7. $n > \text{lda}$  
8. $\text{ldb} \leq 0$  
9. $n > \text{ldb}$  
10. $\text{ldvl} \leq 0$  
11. $n > \text{ldvl}$ and $\text{jobvl} = 'V'$  
12. $\text{ldvr} \leq 0$  
13. $n > \text{ldvr}$ and $\text{jobvr} = 'V'$  
14. $\text{jobvl} = 'V'$ and $\text{ldvl} \leq 0$  
15. $\text{jobvr} = 'V'$ and $\text{ldvr} \leq 0$  
16. $\text{jobvl} = 'V'$ and $\text{ldvl} < n$  
17. $\text{jobvr} = 'V'$ and $\text{ldvr} < n$  
18. $lwork \neq 0$ and $lwork \neq -1$ and $lwork <$ the minimum required value.
19. The size of a work array is greater than 2147483647 when 32-bit integers are used.

**Examples**

**Example 1**
This example shows how to find the eigenvalues only of a long-precision real
generalized eigenproblem \((A, B)\).

**Note:**
1. Set \(ldvl\) and \(ldvr\) to 1 to avoid an error condition.
2. On output, matrices \(A\) and \(B\) are overwritten.
3. Because \(lwork = 0\), the subroutine dynamically allocates \(WORK\).

```
Call Statement and Input:
```
```
JOBVL JOBVR N A LDA B LDB ALPHAR ALPHAI BETA VL LDVL VR LDVR WORK LWORK INFO
CALL DGGEV( 'N', 'N', 3, A, 3, B, 3, ALPHAR, ALPHAI, BETA, VL, 1, VR, 1, WORK, 0, INFO)
```
```
A =
\[
\begin{bmatrix}
10.0 & 1.0 & 2.0 \\
1.0 & 2.0 & -1.0 \\
1.0 & 1.0 & 2.0
\end{bmatrix}
\]
```
```
B =
\[
\begin{bmatrix}
1.0 & 2.0 & 3.0 \\
4.0 & 5.0 & 6.0 \\
7.0 & 8.0 & 9.0
\end{bmatrix}
\]
```
```
Output:
```
```
ALPHAR =
\[
\begin{bmatrix}
2.092346 \\
-4.789188 \\
4.90731
\end{bmatrix}
\]
```
```
ALPHAI =
\[
\begin{bmatrix}
0.000000 \\
0.000000 \\
0.000000
\end{bmatrix}
\]
```
```
BETA =
\[
\begin{bmatrix}
12.711351 \\
0.998541 \\
0.000000
\end{bmatrix}
\]
```
```
INFO = 0
```
```
Example 2
```
This example shows how to find the eigenvalues and the left and right
eigenvectors of a long-precision real generalized eigenproblem \((A, B)\).

**Note:**
1. On output, matrices \(A\) and \(B\) are overwritten.
2. Because \(lwork = 0\), the subroutine dynamically allocates \(WORK\).
3. This matrix is used on page 263 in referenced text [5 on page 1363].

```
Call Statement and Input:
```
```
JOBVL JOBVR N A LDA B LDB ALPHAR ALPHAI BETA VL LDVL VR LDVR WORK LWORK INFO
CALL DGGEV( 'V', 'V', 5, A, 5, B, 5, ALPHAR, ALPHAI, BETA, VL, 1, VR, 1, WORK, 0, INFO)
```
```
A =
\[
\begin{bmatrix}
2.0 & 3.0 & 4.0 & 5.0 & 6.0 \\
4.0 & 4.0 & 5.0 & 6.0 & 7.0 \\
0.0 & 3.0 & 6.0 & 7.0 & 8.0
\end{bmatrix}
\]
Example 3

This example shows how to find the eigenvalues only of a long-precision complex generalized eigenproblem \((A, B)\).

Note:
1. \(ldvl\) and \(ldvr\) have been set to 1 to avoid an error condition.
2. On output, matrices \(A\) and \(B\) are overwritten.
3. Because \(lwork = 0\), the subroutine dynamically allocates \(WORK\).

Call Statement and Input:

\[
\begin{align*}
0.0 & 0.0 & 2.0 & 8.0 & 9.0 \\
0.0 & 0.0 & 0.0 & 1.0 & 10.0 \\
\end{align*}
\]

\[
B = 
\begin{bmatrix}
1.0 & -1.0 & -1.0 & -1.0 & -1.0 \\
0.0 & 1.0 & -1.0 & -1.0 & -1.0 \\
0.0 & 0.0 & 1.0 & -1.0 & -1.0 \\
0.0 & 0.0 & 0.0 & 1.0 & -1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 1.0 \\
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
7.950050 \\
-0.277338 \\
10.987556 \\
\end{bmatrix}
\]

\[
ALPHAR = 
\begin{bmatrix}
2.149669 \\
6.720718 \\
10.987556 \\
\end{bmatrix}
\]

\[
ALPHAI = 
\begin{bmatrix}
0.000000 \\
0.000000 \\
0.000000 \\
\end{bmatrix}
\]

\[
BETA = 
\begin{bmatrix}
1.636872 \\
1.213574 \\
0.908837 \\
\end{bmatrix}
\]

\[
VL = 
\begin{bmatrix}
0.374183 \\
1.480299 \\
0.000000 \\
\end{bmatrix}
\]

\[
VR = 
\begin{bmatrix}
0.000000 \\
0.000000 \\
0.000000 \\
\end{bmatrix}
\]

\[
INFO = 0
\]
Example 4

This example shows how to find the eigenvalues and the left and right eigenvectors of a long-precision complex eigenproblem \((A, B)\).

**Note:**
1. On output, matrices \(A\) and \(B\) are overwritten.
2. Because \(lwork = 0\), the subroutine dynamically allocates \(WORK\).
3. This matrix is used on page 263 in referenced text \[5\] on page 1363.

Call Statement and Input:

\[
\begin{align*}
\text{CALL } & \text{ ZGGEV( 'V', 'V', 3, A, 3, B, 3, ALPHAR, ALPHAI, BETAI, VL, 3, VR, 3, WORK, 0, RWORK, INFO)} \\
A & = \\
\begin{bmatrix}
1.0, & 2.0, & 3.0, & 4.0, & 21.0, & 22.0, \\
43.0, & 44.0, & 13.0, & 14.0, & 15.0, & 16.0, \\
5.0, & 6.0, & 7.0, & 8.0, & 25.0, & 26.0
\end{bmatrix} \\
B & = \\
\begin{bmatrix}
2.0, & 0.0, & 0.0, & -1.0, & 0.0, & 0.0, \\
0.0, & 1.0, & 2.0, & 0.0, & 0.0, & 0.0, \\
0.0, & 0.0, & 0.0, & 3.0, & 0.0, & 0.0
\end{bmatrix}
\end{align*}
\]

Output:

\[
\begin{align*}
\text{ALPHA} & = \\
\begin{bmatrix}
15.863783, & 41.115283 \\
-12.917205, & 19.973815 \\
3.215518, & -4.912439
\end{bmatrix}
\end{align*}
\]
BETA = 
\[
\begin{bmatrix}
1.668461, & 0.0 \\
2.024212, & 0.0 \\
2.664836, & 0.0 \\
\end{bmatrix}
\]

VL = 
\[
\begin{bmatrix}
0.0634, & -0.8686 \\
-0.3652, & -0.3826 \\
0.3605, & 0.6395 \\
\end{bmatrix}
\]

VR = 
\[
\begin{bmatrix}
0.3799, & -0.1986 \\
-0.3652, & -0.9868 \\
0.3799, & 0.2400 \\
\end{bmatrix}
\]

INFO = 0

Example 5

This example shows how to find the eigenvalues, the left and right eigenvectors, and the reciprocal condition numbers of the eigenvalues and right eigenvectors of a long-precision real generalized eigenproblem \((A, B)\).

Note:
1. On output, matrices \(A\) and \(B\) are overwritten.
2. Because \(lwork = 0\), the subroutine dynamically allocates \(WORK\).

Call Statement and Input:

```
CALL DGGEVX( 'P', 'V', 'V', 'B', 6, A, 6, B, 6, ALPHAR, ALPHAI, BETA, VL, 6, VR, 6, ILO, IHI, LSAME, RSCALE, 
              ABNRM, BBNRM, RCONDE, RCONDV, WORK, LWORK, IWORK, BWORK, INFO)
```

\[
A = 
\begin{bmatrix}
9.0 & 98.0 & 1.0 & 96.0 & 30.0 & 94.0 \\
1.0 & 92.0 & 91.0 & 90.0 & 8.0 & 88.0 \\
66.0 & 1.0 & 68.0 & 0.0 & 40.0 & 1.0 \\
0.0 & 1.0 & 64.0 & 50.0 & 66.0 & 1.0 \\
10.0 & 0.0 & 10.0 & 88.0 & 8.0 & 10.0 \\
8.0 & 1.0 & 20.0 & 12.0 & 20.0 & 15.0 \\
\end{bmatrix}
\]

\[
B = 
\begin{bmatrix}
21.0 & 24.0 & 9.0 & 22.0 & 25.0 & 18.0 \\
19.0 & 9.0 & 20.0 & 25.0 & 15.0 & 6.0 \\
5.0 & 4.0 & 9.0 & 14.0 & 2.0 & 21.0 \\
18.0 & 7.0 & 8.0 & 24.0 & 9.0 & 1.0 \\
15.0 & 20.0 & 9.0 & 1.0 & 14.0 & 24.0 \\
21.0 & 23.0 & 9.0 & 14.0 & 88.0 & 88.0 \\
\end{bmatrix}
\]

Output:

```
ALPHAR =
\begin{bmatrix}
64.6956 \\
55.0897 \\
-28.2595 \\
-0.0472 \\
-0.0437 \\
13.8550 \\
\end{bmatrix}
```

```
\begin{bmatrix}
70.4260 \\
-59.9693 \\
\end{bmatrix}
```
Example 6

This example shows how to find the eigenvalues, the left and right eigenvectors, and the reciprocal condition numbers of the eigenvalues and right eigenvectors of a long-precision complex generalized eigenproblem \((A,B)\).

Note:
1. On output, matrices \(A\) and \(B\) are overwritten.
2. Because \(lwork = 0\), the subroutine dynamically allocates \(WORK\).

Call Statement and Input:

```
CALL ZGGEVX('S', 'V', 'V', 'B', 3, A, 3, B, 3, ALPHA, BETA, VL, 3, VR, 3, ILO, IHI, LSSCALE, RSCALE,
          BALANC = JOBVL, JOBVR, SENSE = N, A, LDA, B, LDB, ALPHA, BETA, VL, LDVL, VR, LDVR, ILO, IHI, LSSCALE, RSCALE,
          INFO = 0
```

\[
\begin{bmatrix}
0.0000 \\
28.9059 \\
-26.7715 \\
0.0000 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
15.2997 \\
13.0280 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
\begin{bmatrix}
-0.8391 & 0.3207 & -0.5181 & 0.5129 & 0.1237 & 0.1230 \\
-0.1545 & 0.6066 & 0.0489 & -0.4870 & 0.0269 & -0.0401 \\
-0.2966 & 0.2303 & -0.5078 & 0.4050 & -0.2545 & 0.1577 \\
-0.4076 & 0.5085 & -0.4921 & -0.1974 & -0.0241 & 0.0910 \\
-0.7701 & 0.8937 & -0.6723 & -0.1299 & -0.2564 & 0.0892 \\
0.8391 & -0.6723 & -0.1299 & -0.2564 & -0.3189 & 0.3349 \\
0.1062 & 0.0314 & 0.5842 & 0.0377 & -0.0345 & -0.0360 \\
-0.9222 & 0.0642 & -0.0423 & 0.1502 & 0.1791 & -0.0818 \\
-0.0277 & -0.1385 & -0.0507 & -0.1421 & -0.1836 & 0.2469 \\
0.0654 & -0.8803 & 0.0497 & -0.1390 & 0.1957 & 1.0000 \\
\end{bmatrix}
\end{bmatrix}
\]

\[
\begin{bmatrix}
27.2117 \\
11.8821 \\
6.6118 \\
6.6118 \\
26.3358 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.1657 \\
0.1657 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
5.2466 \\
1.8139 \\
1.8139 \\
6.5921 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.0000 \\
28.9059 \\
-26.7715 \\
0.0000 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
15.2997 \\
13.0280 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
-0.8391 & 0.3207 & -0.5181 & 0.5129 & 0.1237 & 0.1230 \\
-0.1545 & 0.6066 & 0.0489 & -0.4870 & 0.0269 & -0.0401 \\
-0.2966 & 0.2303 & -0.5078 & 0.4050 & -0.2545 & 0.1577 \\
-0.4076 & 0.5085 & -0.4921 & -0.1974 & -0.0241 & 0.0910 \\
-0.7701 & 0.8937 & -0.6723 & -0.1299 & -0.2564 & 0.0892 \\
0.8391 & -0.6723 & -0.1299 & -0.2564 & -0.3189 & 0.3349 \\
0.1062 & 0.0314 & 0.5842 & 0.0377 & -0.0345 & -0.0360 \\
-0.9222 & 0.0642 & -0.0423 & 0.1502 & 0.1791 & -0.0818 \\
-0.0277 & -0.1385 & -0.0507 & -0.1421 & -0.1836 & 0.2469 \\
0.0654 & -0.8803 & 0.0497 & -0.1390 & 0.1957 & 1.0000 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
27.2117 \\
11.8821 \\
6.6118 \\
6.6118 \\
26.3358 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.1657 \\
0.1657 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
5.2466 \\
1.8139 \\
1.8139 \\
6.5921 \\
\end{bmatrix}
\]

INFO = 0
\begin{verbatim}
ABNRM  BBNRM  RCONDE  RCONDV  WORK  LWORK  RWORK  IWORK  BWORK  INFO
ABNRM, BBNRM, RCONDE, RCONDV, WORK, 0, RWORK, IWORK, BWORK, INFO)

A = \begin{bmatrix}
(10.0, 1.0) & (1.0, 0.0) & (2.0, -1.0) \\
(1.0, 2.5) & (2.0, -1.0) & (-1.0, 0.0) \\
(1.0, 0.0) & (1.0, 0.0) & (2.0, 1.5) \\
\end{bmatrix}

B = \begin{bmatrix}
(1.0, 0.0) & (2.0, 0.0) & (3.0, 0.0) \\
(4.0, 1.0) & (5.0, -5.0) & (6.0, -2.0) \\
(7.0, 5.0) & (8.0, -8.0) & (9.0, 0.5) \\
\end{bmatrix}

Output:

\begin{align*}
\text{ALPHA} &= \begin{bmatrix}
-6.6742, & -0.5632 \\
2.1073, & 1.6591 \\
1.2922, & -3.5963 \\
\end{bmatrix} \\
\text{BETA} &= \begin{bmatrix}
0.4561, & 0.0000 \\
17.7898, & 0.0000 \\
1.9190, & 0.0000 \\
\end{bmatrix} \\
\text{VL} &= \begin{bmatrix}
(0.7337, 0.0000) & (0.1686, 0.0647) & (0.9185, 0.0814) \\
(0.0000, 0.0000) & (0.0000, 0.0000) & (1.0000, 0.0000) \\
(0.0000, 0.0000) & (0.0000, 0.0000) & (0.0000, 0.0000) \\
\end{bmatrix} \\
\text{VR} &= \begin{bmatrix}
(1.0000, 0.0000) & (0.0000, 0.0000) & (0.0000, 0.0000) \\
(-0.1260, -0.8739) & (0.1299, -0.2557) & (0.6423, 0.0000) \\
(0.0000, 0.0000) & (0.0000, 0.0000) & (0.0000, 0.0000) \\
\end{bmatrix} \\
\text{RCONDE} &= \begin{bmatrix}
4.1319 \\
15.2697 \\
2.4329 \\
\end{bmatrix} \\
\text{RCONDV} &= \begin{bmatrix}
3.0489 \\
4.1998 \\
2.3870 \\
\end{bmatrix} \\
\text{INFO} &= 0
\end{align*}
\end{verbatim}
SSPGVX, DSPGVX, CHPGVX, ZHPGVX, SSYGVX, DSYGVX, CHEGVX, and ZHEGVX (Eigenvalues and, Optionally, the Eigenvectors of a Positive Definite Real Symmetric or Complex Hermitian Generalized Eigenproblem)

Purpose

These subroutines compute eigenvalues and, optionally, the eigenvectors of a positive definite real symmetric or complex Hermitian generalized eigenproblem:

- If \( \text{itype} = 1 \), the problem is \( Ax = \lambda Bx \)
- If \( \text{itype} = 2 \), the problem is \( ABx = \lambda x \)
- If \( \text{itype} = 3 \), the problem is \( BAx = \lambda x \)

In the formulas above:
- \( A \) represents the real symmetric or complex Hermitian matrix \( A \)
- \( B \) represents the positive definite real symmetric or complex Hermitian matrix \( B \)

Eigenvalues and eigenvectors can be selected by specifying a range of values or a range of indices for the desired eigenvalues.

Table 194. Data Types

<table>
<thead>
<tr>
<th>( A, B, Z, \text{work} )</th>
<th>( vl, vu, \text{abstol, w, rwork} )</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SSPGVX(^a)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SSYGVX(^a)</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DSPGVX(^a)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DSYGVX(^a)</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>CHPGVX(^a)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CHEGVX(^a)</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>ZHPGVX(^a)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZHEGVX(^a)</td>
</tr>
</tbody>
</table>

\(^a\)LAPACK

Syntax

Fortran

CALL SSPGVX | DSPGVX (itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info)

CALL CHPGVX | ZHPGVX (itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info)

CALL SSYGVX | DSYGVX (itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork, ifail, info)

CALL CHEGVX | ZHEGVX (itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork, rwork, iwork, ifail, info)
On Entry

**matrix_layout**

indicates whether the input and output matrices are stored in row major order or column major order, where:

- If `matrix_layout` = LAPACK_ROW_MAJOR, the matrices are stored in row major order.
- If `matrix_layout` = LAPACK_COL_MAJOR, the matrices are stored in column major order.

Specified as: an integer. It must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR

**itype**

specifies the problem type, where:

If `itype` = 1, the problem is $Ax = \lambda Bx$.
If `itype` = 2, the problem is $ABx = \lambda x$.
If `itype` = 3, the problem is $BAX = \lambda x$.

Specified as: an integer; `itype` = 1, 2, or 3.

**jobz**

indicates the type of computation to be performed, where:

If `jobz` = 'N', eigenvalues only are computed.
If `jobz` = 'V', eigenvalues and eigenvectors are computed.

Specified as: a single character; `jobz` = 'N' or 'V'.

**range**

indicates which eigenvalues to compute, where:

If `range` = 'A', all eigenvalues are to be found.
If `range` = 'V', all eigenvalues in the interval $[vl, vu]$ are to be found.
If `range` = 'I', the il-th through iu-th eigenvalues are to be found.
Specified as: a single character; \textit{range} = 'A', 'V', or 'I'.

\textit{uplo}
indicates whether the upper or lower triangular part of the matrices \(A\) and \(B\) are referenced, where:

If \textit{uplo} = 'U', the upper triangular part is referenced.

If \textit{uplo} = 'L', the lower triangular part is referenced.

Specified as: a single character; \textit{uplo} = 'U' or 'L'.

\textit{n}
is the order of matrices \(A\) and \(B\) used in the computation.

Specified as: an integer; \(n \geq 0\).

\textit{ap}
is the real symmetric or complex Hermitian matrix \(A\) of order \(n\). It is stored in an array, referred to as \(AP\), where:

If \textit{uplo} = 'U', it is stored in upper-packed storage mode.

If \textit{uplo} = 'L', it is stored in lower-packed storage mode.

Specified as: one-dimensional array of (at least) length \(n(n + 1)/2\), containing numbers of the data type indicated in Table 194 on page 1008.

\textit{bp}
is the positive definite real symmetric or complex Hermitian matrix \(B\) of order \(n\). It is stored in an array, referred to as \(BP\), where:

If \textit{uplo} = 'U', it is stored in upper-packed storage mode.

If \textit{uplo} = 'L', it is stored in lower-packed storage mode.

Specified as: one-dimensional array of (at least) length \(n(n + 1)/2\), containing numbers of the data type indicated in Table 194 on page 1008.

\textit{a}
is the real symmetric or complex Hermitian matrix \(A\) of order \(n\).

If \textit{uplo} = 'U', it is stored in upper storage mode.

If \textit{uplo} = 'L', 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 194 on page 1008.

\textit{lda}
is the leading dimension of the array specified for \(A\).

Specified as: an integer; \(lda > 0\) and \(lda \geq n\).

\textit{b}
is the positive definite real symmetric or complex Hermitian matrix \(B\) of order \(n\).

If \textit{uplo} = 'U', it is stored in upper storage mode.

If \textit{uplo} = 'L', 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 194 on page 1008.

\textit{ldb}
is the leading dimension of the array specified for \(B\).

Specified as: an integer; \(ldb > 0\) and \(ldb \geq n\).

\textit{vl}
has the following meaning:

If \textit{range} = 'V', it is the lower bound of the interval to be searched for eigenvalues.
If \( \text{range} \neq 'V' \), this argument is ignored.

Specified as: a number of the data type indicated in [Table 194 on page 1008]

If \( \text{range} = 'V', vl < vu \).

\( vu \) has the following meaning:

If \( \text{range} = 'V' \), it is the upper bound of the interval to be searched for eigenvalues.

If \( \text{range} \neq 'V' \), type indicated in [Table 194 on page 1008]. If \( \text{range} = 'V', vl < vu \).

\( il \) has the following meaning:

If \( \text{range} = 'I', it is the index (from smallest to largest) of the smallest eigenvalue to be returned.

If \( \text{range} \neq 'I', this argument is ignored.

Specified as: an integer; \( il \geq 1 \).

\( iu \) has the following meaning:

If \( \text{range} = 'I', it is the index (from smallest to largest) of the largest eigenvalue to be returned.

If \( \text{range} \neq 'I', this argument is ignored.

Specified as: an integer; \( \min(il, n) \leq iu \leq n \).

\( \text{abstol} \)

is the absolute tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval \([a, b]\) of width less than or equal to:

- \( \text{abstol} + \epsilon(\max(|a|, |b|)) \)

where \( \epsilon \) is the machine precision. If \( \text{abstol} \leq \text{zero} \), then \( \epsilon(\text{norm}(T)) \) is used in its place, where \( \text{norm}(T) \) is the 1-norm of the tridiagonal matrix obtained by reducing the standard form of the generalized problem to tridiagonal form. For most problems, this is the appropriate level of accuracy to request.

For certain strongly graded matrices, greater accuracy can be obtained in very small eigenvalues by setting \( \text{abstol} \) to a very small positive number. However, if \( \text{abstol} \) is less than:

\[ \sqrt{\text{unfl}} \]

where \( \text{unfl} \) is the underflow threshold, then:

\[ \sqrt{\text{unfl}} \]

is used in its place.

Eigenvalues are computed most accurately when \( \text{abstol} \) is set to twice the underflow threshold—that is, \( 2(\text{unfl}) \).

If \( \text{jobz} = 'V' \), then setting \( \text{abstol} \) to \( \text{unfl} \), the underflow threshold, yields the most orthogonal eigenvectors.

**Note:**

1. The approximate values of the constants used for \( \text{abstol} \) are listed below:
For SSPGVX, CHPGVX, SSYGVX, and CHEGVX

\[ \epsilon = 0.119209289550781250E-06 \]
\[ \text{unfl} = 0.1175494351E-37 \]
\[ \sqrt{\text{unfl}} = 0.1084202172E-18 \]

For DSPGVX, ZHPGVX, DSYGVX, and ZHEGVX

\[ \epsilon = 0.222044604925031308E-15 \]
\[ \text{unfl} = 0.222507385850720138E-307 \]
\[ \sqrt{\text{unfl}} = 0.149166814624004135E-153 \]

2. The value of \( \text{abstol} \) can affect which algorithm is used to compute the eigenvalues and eigenvectors. See [Function](#).

Specified as: a number of the data type indicated in [Table 194 on page 1008](#).

- \( m \) See "On Return" on page 1013.
- \( w \) See "On Return" on page 1013.
- \( z \) See "On Return" on page 1013.

\( \text{ldz} \)

is the leading dimension of the array specified for \( Z \).

Specified as: an integer; \( \text{ldz} > 0 \) and \( \text{ldz} \geq n \).

\( \text{work} \)

is a work area used by these subroutines, where:

- For SSPGVX and DSPGVX
  - Its size is \( 8n \).

- For CHPGVX and ZHPGVX
  - Its size is \( 2n \).

- For SSYGVX, DSYGVX, CHEGVX, and ZHEGVX
  - If \( lwork = 0 \), \( \text{work} \) is ignored.
    - If \( lwork \neq 0 \), the size of \( \text{work} \) is determined as follows:
      - If \( lwork \neq -1 \), \( \text{work} \) is (at least) of length \( lwork \).
      - If \( lwork = -1 \), \( \text{work} \) is (at least) of length 1.

Specified as: an area of storage containing numbers of the data type indicated in [Table 194 on page 1008](#).

\( lwork \)

is the number of elements in array \( \text{WORK} \).

Specified as: an integer, where:

- If \( lwork = 0 \), the subroutine dynamically allocates the workspace needed for use during this computation. The work area is deallocated before control is returned to the calling program.
- If \( lwork = -1 \), subroutine performs a workspace query and returns the optimal required size of \( \text{work} \) in \( \text{work}_i \). No computation is performed and the subroutine returns after error checking is complete.
• Otherwise:

\textbf{For SSYGVX and DSYGVX}
\[ lwork \geq \max(1, 8n). \]

\textbf{For CHEGVX and ZHEGVX}
\[ lwork \geq \max(1, 2n). \]

\textbf{rwork}

is a work area of size 7n.

Specified as: an area of storage containing numbers of the data type indicated in \textit{Table 194 on page 1008}

\textbf{iwork}

is a work area of size 5n.

Specified as: an area of storage containing integers.

\textbf{ifail}

See “On Return.”

\textbf{info}

See “On Return.”

\textbf{On Return}

\textbf{ap}

is overwritten.

Returned as: a one-dimensional array of (at least) length \( n(n+1)/2 \), containing numbers of the data type indicated in \textit{Table 194 on page 1008}

\textbf{bp}

contains the results of the Cholesky factorization.

\textbf{For SSPGVX and DSPGVX}

If \( \text{uplo} = 'U' \), if \( \text{info} \leq n \), the triangular factor \( U \) from the Cholesky factorization \( B = \mathbf{U}^T \mathbf{U} \) stored in upper-packed storage format.

If \( \text{uplo} = 'L' \), if \( \text{info} \leq n \), the triangular factor \( L \) from the Cholesky factorization \( B = \mathbf{L} \mathbf{L}^T \) stored in lower-packed storage mode.

\textbf{For CHPGVX and ZHPGVX}

If \( \text{uplo} = 'U' \), if \( \text{info} \leq n \), the triangular factor \( U \) from the Cholesky factorization \( B = \mathbf{U} \mathbf{U}^H \) stored in upper-packed storage format.

If \( \text{uplo} = 'L' \), if \( \text{info} \leq n \), the triangular factor \( L \) from the Cholesky factorization \( B = \mathbf{L} \mathbf{L}^H \) stored in lower-packed storage mode.

Returned as: one-dimensional array of (at least) length \( n(n+1)/2 \), containing numbers of the data type indicated in \textit{Table 194 on page 1008}

\textbf{a}

is overwritten as follows:

• If \( \text{uplo} = 'U' \), the leading \( n \) by \( n \) upper triangular part of \( A \) is overwritten.

• If \( \text{uplo} = 'L' \), the leading \( n \) by \( n \) lower triangular part of \( A \) is overwritten.

Returned as: an array of dimension \( \text{lda} \) by (at least) \( n \), containing numbers of the data type indicated in \textit{Table 194 on page 1008}

\textbf{b}

contains the results of the Cholesky factorization.

\textbf{For SSYGVX and DSYGVX}

If \( \text{uplo} = 'U' \), if \( \text{info} \leq n \), the leading \( n \) by \( n \) upper triangular part of \( B \) contains the triangular factor \( \mathbf{U} \) from the Cholesky factorization \( B = \mathbf{U} \mathbf{U}^T \).
If \( uplo = 'L' \), if \( info \leq n \), the leading \( n \) by \( n \) lower triangular part of \( B \) contains the triangular factor \( L \) from the Cholesky factorization \( B = LL^T \).

For CHEGVX and ZHEGVX

If \( uplo = 'U' \), if \( info \leq n \), the leading \( n \) by \( n \) upper triangular part of \( B \) contains the triangular factor \( U \) from the Cholesky factorization \( B = U^H U \).

If \( uplo = 'L' \), if \( info \leq n \), the leading \( n \) by \( n \) lower triangular part of \( B \) contains the triangular factor \( L \) from the Cholesky factorization \( B = LL^H \).

\( m \) is the number of eigenvalues found.

\( w \) is the vector \( w \), containing the computed eigenvalues in ascending order in the first \( m \) elements of \( w \).

\( z \) has the following meaning, where:

If \( jobz = 'N' \), then \( z \) is ignored.

If \( jobz = 'V' \) and \( info = 0 \), the first \( m \) columns of \( Z \) contain the eigenvectors corresponding to the selected eigenvalues, with the \( i \)-th column of \( Z \) holding the eigenvector associated with \( w(i) \).

The eigenvectors are normalized as follows:

For SSYGVX, DSYGVX, SSPGVX, and DSPGVX

If \( itype = 1 \) or \( 2 \), \( Z^T B Z = I \).

If \( itype = 3 \), \( Z^T B^{-1} Z = I \).

For CHPGVX, ZHPGVX, CHEGVX, and ZHEGVX

If \( itype = 1 \) or \( 2 \), \( Z^H B Z = I \).

If \( itype = 3 \), \( Z^H B^{-1} Z = I \).

where \( I \) is the identity matrix.

If an eigenvector fails to converge, then that column of \( Z \) contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in \( ifail \).

**Note:** You must ensure that at least \( \max(1, m) \) columns are supplied in the array \( Z \); if \( range = 'V' \), the exact value of \( m \) is not known in advance and an upper bound must be used.

Returned as: an \( ldz \) by (at least) \( \max(1, m) \) array, containing numbers of the data type indicated in Table 194 on page 1008.

\( work \)

is a work area used by this subroutine if \( lwork \neq 0 \), where:

If \( lwork \neq 0 \) and \( lwork \neq -1 \), its size is (at least) of length \( lwork \).

If \( lwork = -1 \), its size is (at least) of length 1.

Returned as: an area of storage, where:
If `lwork ≥ 1` or `lwork = -1`, then `work_1` is set to the optimal `lwork` value and all other elements of `work` are overwritten.

**ifail**

has the following meaning:

If `jobz = 'N'`, `ifail` is ignored.

If `jobz = 'V'`:

- If `info = 0`, the first `m` elements of `ifail` are zero.
- If `info > 0`, `ifail` contains the indices of the eigenvectors that failed to converge.

Returned as: an array of length `n`, containing integers.

**info**

has the following meaning:

If `info = 0`, the subroutine completed successfully.

If `info = i`, then `i` eigenvectors failed to converge. Their indices are saved in array `ifail`.

If `info = n + i` for `1 ≤ i ≤ n`, then the leading minor of order `i` of `B` is not positive definite. The factorization of `B` could not be completed, and no eigenvalues or eigenvectors were computed.

Returned as:

- For `SSPGVX`, `DSPGVX`, `CHPGVX`, `ZHPGVX`, `SSYGVX`, `DSYGVX`, `CHEGVX`, and `ZHEGVX` returned as: an integer; `info ≥ 0`.
- For `LAPACKE_sspgsvx`, `LAPACKE_dspgsvx`, `LAPACKE_chpgvx`, `LAPACKE_zhpgvx`, `LAPACKE_ssygsvx`, `LAPACKE_dsygsvx`, `LAPACKE_chegvx`, and `LAPACKE_zhegsvx` returned as an integer function value; `info ≥ 0`.

**Notes**

1. These subroutines accept lowercase letters for the `jobz`, `range`, and `uplo` arguments.

2. In your C program, arguments `m` and `info` must be passed by reference.

3. `ap`, `bp`, `a`, `b`, `w`, `z`, `work`, `rwork`, `iwork` and `ifail` must have no common elements; otherwise, results are unpredictable.

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 85](#) or [“Upper-Packed Storage Mode” on page 87](#), respectively.

For a description of how complex Hermitian matrices are stored in lower- or upper-packed storage mode, see [“Complex Hermitian Matrix” on page 90](#).

5. For a description of how real symmetric matrices are stored in lower or upper storage mode, see [“Lower Storage Mode” on page 88](#) or [“Upper Storage Mode” on page 89](#), respectively.

For a description of how complex Hermitian matrices are stored in lower or upper storage mode, see [“Complex Hermitian Matrix” on page 90](#).

6. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrices `A` and `B` are assumed to be zero, so you do not have to set these values. On output, for matrix `B` they are set to zero.

7. For best performance, specify `lwork = 0`.  

---

Chapter 11. Eigensystem Analysis
Function

These subroutines compute eigenvalues and, optionally, the eigenvectors of a positive definite real symmetric or complex Hermitian generalized eigenproblem:

- If \( \text{itype} = 1 \), the problem is \( Ax = \lambda Bx \)
- If \( \text{itype} = 2 \), the problem is \( ABx = \lambda x \)
- If \( \text{itype} = 3 \), the problem is \(BAx = \lambda x \)

In the formulas above:

- \( A \) represents the real symmetric or complex Hermitian matrix \( A \)
- \( B \) represents the positive definite real symmetric or complex Hermitian matrix \( B \)

Eigenvalues and eigenvectors can be selected by specifying a range of values or a range of indices for the desired eigenvalues.

The computation involves the following steps:

1. Compute the Cholesky factorization of \( B \).
2. Reduce the positive definite real symmetric or complex Hermitian generalized eigenproblem to standard form.
3. Compute the requested eigenvalues and, optionally, the eigenvectors of the standard form.
4. Backtransform the eigenvectors to obtain the eigenvectors of the original problem.

If \( n = 0 \), no computation is performed and the subroutine returns after doing some parameter checking.

See reference [8 on page 1363].

Error conditions

Resource Errors

1. \( \text{lwork} = 0 \) and unable to allocate work area

Computational Errors

1. The matrix \( B \) is not positive definite. See output argument \( \text{info} \) for more details.
2. Bisection failed to converge for some eigenvalues. The eigenvalues may not be as accurate as the absolute and relative tolerances.
3. The number of eigenvalues computed does not match the number of eigenvalues requested.
4. No eigenvalues were computed because the Gershgorin interval initially used was incorrect.
5. Some eigenvectors failed to converge. The indices are stored in \( \text{ifail} \).

Informational Errors

1. ESSL computed the eigenvalues using multiple algorithms. Performance may be degraded.

Input-Argument Errors

1. \( \text{itype} < 1 \) or \( \text{itype} > 3 \)
2. \( \text{jobz} \neq \text{'N'} \) or \( \text{'V'} \)
3. \( \text{range} \neq \text{'A'}, \text{'V'}, \) or \( \text{'I'} \)
4. \( \text{uplo} \neq \text{'U'} \) or \( \text{'L'} \)
5. \( n < 0 \)
6. \( lda \leq 0 \)
7. \( lda < n \)
8. \( ldb \leq 0 \)
9. \( ldb < n \)
10. \( range = 'V' \), \( n > 0 \), and \( vu \leq vl \)
11. \( range = 'T' \) and \((il < 1 \text{ or } il > \max(1, n))\)
12. \( range = 'T' \) and \((iu < \min(n, il) \text{ or } iu > n)\)
13. \( ldz \leq 0 \)
14. \( jobz = 'V' \) and \( ldz < n \)
15. \( lwork \neq 0 \) and \( lwork \neq -1 \) and \( lwork < \) the minimum required value

**Examples**

**Example 1**

This example shows how to find the eigenvalues of a real symmetric positive
generalized eigenproblem of the form: \( Ax = \lambda B x \). Matrices \( A \) and \( B \) are stored
in lower-packed storage mode.

Matrix \( A \) is:

\[
\begin{bmatrix}
6.0 & 4.0 & 4.0 & 1.0 \\
4.0 & 6.0 & 1.0 & 4.0 \\
4.0 & 1.0 & 6.0 & 4.0 \\
1.0 & 4.0 & 4.0 & 6.0 \\
\end{bmatrix}
\]

Matrix \( B \) is:

\[
\begin{bmatrix}
1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 1.0 \\
\end{bmatrix}
\]

**Notes:**

1. Because \( jobz = 'N' \), \( Z \) and \( ifail \) are not referenced.
2. Because \( range = 'A' \), arguments \( vl, vu, il, \) and \( iu \) are not referenced.

**Call Statement and Input:**

```
CALL DSPGVX ( 1, 'N', 'A', 'L', 4, AP, BP, VL, VU, IL, IU, -1.0, M, W, Z, 4, WORK, IWORK, IFAIL, INFO)
```

**Output:**

Matrix \( AP \) is overwritten.

\( AP = (6.0, 4.0, 4.0, 1.0, 6.0, 1.0, 4.0, 6.0, 4.0, 6.0) \)

\( BP = (1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0) \)

**Example 2**
This example shows how to find the eigenvalues and eigenvectors of a real symmetric positive generalized eigenproblem of the form: \( ABx = \lambda x \). Matrices \( A \) and \( B \) are stored in upper-packed storage mode.

This example illustrates the use of the \( il \) and \( iu \) parameters when range = 'T'. Matrices \( A \) and \( B \) are the same as in Example 1.

Notes:
1. Because range = 'I', arguments \( vl \) and \( vu \) are not referenced.
2. Because \( range = 'A' \), arguments \( vl \), \( vu \), \( il \), and \( iu \) are not referenced.

Output:

Matrix \( AP \) is overwritten.

\[
\begin{pmatrix}
6.0 & 4.0 & 6.0 & 4.0 & 1.0 & 6.0 & 1.0 & 4.0 & 4.0 & 6.0
\end{pmatrix}
\]

\[BP = (1.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0)\]

\[M = 2\]

\[W = (-1.0, 5.0, \ldots, \ldots)\]

\[
Z = \begin{bmatrix}
0.500000 & 0.544042 & \cdot & \cdot \\
-0.500000 & 0.451683 & \cdot & \cdot \\
-0.500000 & -0.451683 & \cdot & \cdot \\
0.500000 & -0.544042 & \cdot & \cdot \\
\end{bmatrix}
\]

\[IFAIL = (0, 0, \ldots, \ldots)\]

\[INFO = 0\]

Example 3

This example shows how to find all eigenvalues only of a positive definite complex Hermitian generalized eigenproblem of the form: \( Ax = \lambda Bx \). Matrices \( A \) and \( B \) are stored in lower-packed storage mode.

Matrix \( A \) is:

\[
\begin{pmatrix}
6.0 & 0.0 & 4.0 & 0.0 & 4.0 & 0.0 & 1.0 & 0.0 \\
4.0 & 0.0 & 6.0 & 0.0 & 1.0 & 0.0 & 4.0 & 0.0 \\
4.0 & 0.0 & 1.0 & 0.0 & 6.0 & 0.0 & 4.0 & 0.0 \\
1.0 & 0.0 & 4.0 & 0.0 & 4.0 & 0.0 & 6.0 & 0.0 \\
\end{pmatrix}
\]

Matrix \( B \) is:

\[
\begin{pmatrix}
1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \\
\end{pmatrix}
\]

Notes:
1. Because \( jobz = 'N' \), \( Z \) and \( ifail \) are not referenced.
2. Because \( range = 'A' \), arguments \( vl \), \( vu \), \( il \), and \( iu \) are not referenced.
3. On input, the imaginary parts of the Hermitian matrix $A$ are assumed to be zero, values. On output, they are set to zero.

Call Statement and Input:

\[
\text{CALL ZHPEVX} \begin{array}{l}
\text{ITYPE} = 1, \quad \text{JOBZ} = 'N' , \quad \text{UPLO} = 'L', \quad \text{N} = 4, \quad \text{AP} = ((1.0 , 0.0), (4.0 , 0.0), (4.0 , 0.0), (1.0 , 0.0), (6.0 , 0.0), (1.0 , 0.0), (4.0 , 0.0), (6.0 , 0.0), (4.0 , 0.0), (6.0 , 0.0)) \\
\text{BP} = ((1.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (1.0, 0.0), (0.0, 0.0), (0.0, 0.0), (1.0, 0.0), (0.0, 0.0), (1.0, 0.0)) \\
\end{array}
\]

Output:

Matrix AP is overwritten.

\[
\begin{array}{c}
\text{M} = 4 \\
\text{BP} = ((1.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (1.0, 0.0), (0.0, 0.0), (0.0, 0.0), (1.0, 0.0), (0.0, 0.0), (1.0, 0.0)) \\
\text{W} = (-1.0, 5.0, 5.0, 15.0) \\
\text{INFO} = 0 \\
\end{array}
\]

Example 4

This example shows how to find all eigenvalues and eigenvectors of a positive definite complex Hermitian generalized eigenproblem of the form: $AB\mathbf{x} = \lambda \mathbf{x}$. Matrices $A$ and $B$ are stored in upper-packed storage mode.

This example illustrates the use of the $il$ and $iu$ parameters when range = 'T'. Matrices $A$ and $B$ are the same as in Example 3.

Notes:

1. Because range = 'T', arguments vl and vu are not referenced.
2. On output, the imaginary parts of the Hermitian matrix $A$ are assumed to be zero, values. On output, they are set to zero.

Call Statement and Input:

\[
\text{CALL ZHPEVX} \begin{array}{l}
\text{ITYPE} = 2, \quad \text{JOBZ} = 'V', \quad \text{UPLO} = 'L', \quad \text{N} = 4, \quad \text{AP} = ((1.0 , 0.0), (4.0 , 0.0), (4.0 , 0.0), (1.0 , 0.0), (6.0 , 0.0), (1.0 , 0.0), (4.0 , 0.0), (6.0 , 0.0), (4.0 , 0.0), (6.0 , 0.0)) \\
\text{BP} = ((1.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (1.0, 0.0), (0.0, 0.0), (0.0, 0.0), (1.0, 0.0), (0.0, 0.0), (1.0, 0.0)) \\
\end{array}
\]

Output:

Matrix AP is overwritten.

\[
\begin{array}{c}
\text{BP} = ((1.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (1.0, 0.0), (0.0, 0.0), (0.0, 0.0), (1.0, 0.0), (0.0, 0.0), (1.0, 0.0)) \\
\text{M} = 2 \\
\text{W} = (-1.0, 5.0, \ldots) \\
\text{Z} = \begin{bmatrix}
0.500000 & 0.0 \\
0.544042 & 0.0 \\
-0.500000 & 0.451683 \\
-0.544042 & 0.0 \\
0.500000 & -0.451683 \\
-0.500000 & 0.0 \\
\end{bmatrix} \\
\text{IFAIL} = \{0, 0, \ldots\} \\
\text{INFO} = 0 \\
\end{array}
\]

Example 5
This example shows how to find the eigenvalues only of a positive definite real symmetric generalized eigenproblem of the form: \( Ax = \lambda Bx \). Matrices \( A \) and \( B \) are stored in lower storage mode.

Matrices \( A \) and \( B \) are the same as in Example 1

Notes:
1. Because \( \text{jobz} = 'N' \), \( Z \) and \( \text{ifail} \) are not referenced.
2. Because \( \text{range} = 'A' \), arguments \( vl \), \( vu \), \( il \), and \( iu \) are not referenced.
3. Because \( lwork = 0 \), the subroutine dynamically allocates \( \text{WORK} \).

Call Statement and Input:

\[
\begin{bmatrix}
6.0 & . & . \\
4.0 & 6.0 & . \\
1.0 & 4.0 & 4.0 & 6.0
\end{bmatrix}
\]

\[
\begin{bmatrix}
1.0 & . & . \\
0.0 & 1.0 & . \\
0.0 & 0.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 1.0
\end{bmatrix}
\]

Output:
Matrix \( A \) is overwritten.

\[
M = 4
\]

\[
\begin{bmatrix}
1.000000 & . & . & . \\
0.000000 & 1.000000 & . & . \\
0.000000 & 0.000000 & 1.000000 & . \\
0.000000 & 0.000000 & 0.000000 & 1.000000
\end{bmatrix}
\]

\[
\begin{bmatrix}
-1.000000 \\
5.000000 \\
5.000000 \\
15.000000
\end{bmatrix}
\]

INFO = 0

Example 6

This example shows how to find the eigenvalues and eigenvectors of a positive definite real symmetric generalized eigenproblem of the form: \( ABx = \lambda x \). Matrices \( A \) and \( B \) are stored in upper storage mode.

This example illustrates the use of the \( il \) and \( iu \) parameters when \( \text{range} = 'I' \).

Matrices \( A \) and \( B \) are the same as in Example 1

Notes:
1. Because \( \text{range} = 'I' \), arguments \( vl \) and \( vu \) are not referenced.
2. Because \( lwork = 0 \), the subroutine dynamically allocates \( \text{WORK} \).
Example 7

This example shows how to find the eigenvalues and eigenvectors of a positive definite real symmetric generalized eigenproblem of the form: \(BAx = \lambda x\).

Matrices \(A\) and \(B\) are stored in upper storage mode.

This example illustrates the use of the \(vl\) and \(vu\) parameters when \(range = 'V'\).

Matrices \(A\) and \(B\) are the same as in Example 1.

Notes:

1. Because \(range = 'V'\), arguments \(il\) and \(iu\) are not referenced.
2. On output, array \(A\) is overwritten.
3. Because \(lwork = 0\), the subroutine dynamically allocates \(WORK\).
Matrix $A$ is overwritten.

\[
M = 2
\]

\[
B = \begin{bmatrix}
1.000000 & 0.000000 & 0.000000 & 0.000000 \\
0.000000 & 1.000000 & 0.000000 & 0.000000 \\
0.000000 & 0.000000 & 1.000000 & 0.000000 \\
0.000000 & 0.000000 & 0.000000 & 1.000000 \\
\end{bmatrix}
\]

\[
W = \begin{bmatrix}
5.000000 \\
5.000000 \\
\end{bmatrix}
\]

\[
Z = \begin{bmatrix}
-0.123202 & -0.696291 & . & . \\
0.696291 & -0.123202 & . & . \\
-0.696291 & 0.123202 & . & . \\
0.123202 & 0.696291 & . & . \\
\end{bmatrix}
\]

\[
IFAIL = (0, 0, ., .) \\
INFO = 0
\]

**Example 8**

This example shows how to find the eigenvalues only of a positive definite complex Hermitian generalized eigenproblem of the form: $Ax = \lambda Bx$. Matrices $A$ and $B$ are stored in lower-packed storage mode.

Matrices $A$ and $B$ are the same as in Example 3.

**Notes:**

1. Because $jobz = 'N'$, $Z$ and $ifail$ are not referenced.
2. Because $range = 'A'$, arguments $vl$, $vu$, $il$, and $iu$ are not referenced.
3. Because $lwork = 0$, the subroutine dynamically allocates WORK.
4. On input, the imaginary parts of the Hermitian matrix $A$ are assumed to be zero, values. On output, they are set to zero.

Call Statement and Input:

```
CALL ZHEGVX ( ITYPE, JOBZ, RANGE, UPLO, N, A, LDA, B, LDB, VL, VI, IL, IU, ABSTOL, M, W, Z, LDZ, WORK, LWORK, RWORK, IWORK, IFAIL, INFO )
```

```
A = \begin{bmatrix}
(6.0, .) & . & . & . \\
(4.0, 0.0) & (6.0, .) & (6.0, .) & . \\
(4.0, 0.0) & (1.0, 0.0) & (6.0, .) & . \\
(1.0, 0.0) & (4.0, 0.0) & (4.0, 0.0) & (6.0, .) \\
\end{bmatrix}
```

```
B = \begin{bmatrix}
(1.0, .) & (1.0, .) & . & . \\
(0.0, 0.0) & (0.0, 0.0) & (1.0, .) & . \\
(0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (1.0, .) \\
\end{bmatrix}
```

Output:

Matrix $A$ is overwritten.

\[
M = 4
\]

\[
B = \begin{bmatrix}
(1.0, 0.0) & . & . & . \\
(0.0, 0.0) & (1.0, 0.0) & . & . \\
(0.0, 0.0) & (0.0, 0.0) & (1.0, 0.0) & . \\
(0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (1.0, 0.0) \\
\end{bmatrix}
\]
Example 9

This example shows how to find the eigenvalues and eigenvectors of a positive definite complex Hermitian generalized eigenproblem of the form: $ABx = \lambda x$. Matrices $A$ and $B$ are stored in upper-packed storage mode.

This example illustrates the use of the $il$ and $iu$ parameters when range = 'T'.

Matrices $A$ and $B$ are the same as in Example 3

Notes:
1. Because range = 'T', arguments $vl$ and $vu$ are not referenced.
2. Because $lwork = 0$, the subroutine dynamically allocates $w0rk$.
3. On input, the imaginary parts of the Hermitian matrix $A$ are assumed to be zero, values. On output, they are set to zero.

Call Statement and Input:

```
ITYPE JOBZ RANGE UPLE N A LDA B LDB VU VL IL IUBSTOL M W Z LDZ WORK LWORK RWORK IWORK IFAIL INFO
```

```
CALL ZHEGVX ( 2, 'V', 'I', 4, A, 4, B, 4, VL, VU, 1, 2, -1.0, M, W, Z, 4, WORK, 0, RWORK IWORK, IFAIL, INFO)
```

```
A =
\begin{bmatrix}
(6.0, .) & . & . & . \\
(4.0, 0.0) & (6.0, .) & . & . \\
(1.0, 0.0) & (4.0, 0.0) & (4.0, 0.0) & (6.0, .) \\
\end{bmatrix}
```

```
B =
\begin{bmatrix}
(1.0, .) & . & . & . \\
(0.0, 0.0) & (1.0, .) & . & . \\
(0.0, 0.0) & (0.0, 0.0) & (1.0, .) & . \\
(0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (1.0, .) \\
\end{bmatrix}
```

Output:

Matrix $A$ is overwritten.

```
M = 2
```

```
B =
\begin{bmatrix}
(1.0, 0.0) & . & . & . \\
(0.0, 0.0) & (1.0, 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) & (1.0, 0.0) \\
\end{bmatrix}
```

```
W =
\begin{bmatrix}
-1.000000 \\
5.000000 \\
\end{bmatrix}
```

```
Z =
\begin{bmatrix}
0.500000, 0.0 & -0.154815, 0.0 & . & . \\
-0.500000, 0.0 & -0.689950, 0.0 & . & . \\
-0.500000, 0.0 & 0.689950, 0.0 & . & . \\
0.500000, 0.0 & 0.154815, 0.0 & . & . \\
\end{bmatrix}
```

```
IFAIL = (\{0, 0, \ldots\}
```

```
INFO = 0
```

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

The signal processing subroutines, provided in three areas, are described here.

Overview of the Signal Processing Subroutines

This describes the subroutines in each of the three signal processing areas:
- Fourier transform subroutines
- Convolution and correlation subroutines
- Related-computation subroutines

Fourier Transforms Subroutines

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

Table 195. List of Fourier Transform Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCFTD</td>
<td>DCFTD</td>
<td>“SCFTD and DCFTD (Multidimensional Complex Fourier Transform)” on page 1036</td>
</tr>
<tr>
<td>SRCFTD</td>
<td>DRCFTD</td>
<td>“SRCFTD and DRCFTD (Multidimensional Real-to-Complex Fourier Transform)” on page 1044</td>
</tr>
<tr>
<td>SCRFTD</td>
<td>DCRFTD</td>
<td>“SCRFTD and DCRFTD (Multidimensional Complex-to-Real Fourier Transform)” on page 1052</td>
</tr>
<tr>
<td>SCF\textsuperscript{§}</td>
<td>DCFT\textsuperscript{§}</td>
<td>“SCFT and DCFT (Complex Fourier Transform)” on page 1060</td>
</tr>
<tr>
<td>SRCF\textsuperscript{§}</td>
<td>DRCF\textsuperscript{§}</td>
<td>“SRCFT and DRCFT (Real-to-Complex Fourier Transform)” on page 1069</td>
</tr>
<tr>
<td>SCR\textsuperscript{§}</td>
<td>DCR\textsuperscript{§}</td>
<td>“SCRFT and DCRFT (Complex-to-Real Fourier Transform)” on page 1077</td>
</tr>
<tr>
<td>SCOSF SCOSFT\textsuperscript{§}, ND</td>
<td>DCOSF</td>
<td>“SCOSF and DCOSF (Cosine Transform)” on page 1085</td>
</tr>
<tr>
<td>SSINF</td>
<td>DSINF</td>
<td>“SSINF and DSINF (Sine Transform)” on page 1093</td>
</tr>
<tr>
<td>SCFT2\textsuperscript{§}, SCFT2P\textsuperscript{§}, ND</td>
<td>DCFT2\textsuperscript{§}</td>
<td>“SCFT2 and DCFT2 (Complex Fourier Transform in Two Dimensions)” on page 1101</td>
</tr>
<tr>
<td>SRCFT2\textsuperscript{§}</td>
<td>DRCFT2\textsuperscript{§}</td>
<td>“SRCFT2 and DRCFT2 (Real-to-Complex Fourier Transform in Two Dimensions)” on page 1108</td>
</tr>
<tr>
<td>SCRFT2\textsuperscript{§}</td>
<td>DCRFT2\textsuperscript{§}</td>
<td>“SCRFT2 and DCRFT2 (Complex-to-Real Fourier Transform in Two Dimensions)” on page 1115</td>
</tr>
<tr>
<td>SCFT3\textsuperscript{§}, SCFT3P\textsuperscript{§}, ND</td>
<td>DCFT3\textsuperscript{§}</td>
<td>“SCFT3 and DCFT3 (Complex Fourier Transform in Three Dimensions)” on page 1123</td>
</tr>
<tr>
<td>SRCFT3\textsuperscript{§}</td>
<td>DRCFT3\textsuperscript{§}</td>
<td>“SRCFT3 and DRCFT3 (Real-to-Complex Fourier Transform in Three Dimensions)” on page 1130</td>
</tr>
</tbody>
</table>
Table 195. List of Fourier Transform Subroutines (continued)

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCRFT3§</td>
<td>DCRFT3§</td>
<td>“SCRFT3 and DCRFT3 (Complex-to-Real Fourier Transform in Three Dimensions)” on page 1137</td>
</tr>
</tbody>
</table>

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

ND Documentation for this subroutine is no longer provided.

### 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 196. List of Convolution and Correlation Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCON§</td>
<td>SCOR§</td>
<td>“SCON and SCOR (Convolution or Correlation of One Sequence with One or More Sequences)” on page 1145</td>
</tr>
<tr>
<td>SCOND</td>
<td>SCORD</td>
<td>“SCOND and SCORD (Convolution or Correlation of One Sequence with Another Sequence Using a Direct Method)” on page 1151</td>
</tr>
<tr>
<td>SCONF</td>
<td>SCORF</td>
<td>“SCONF and SCORF (Convolution or Correlation of One Sequence with One or More Sequences Using the Mixed-Radix Fourier Method)” on page 1157</td>
</tr>
<tr>
<td>SDCON</td>
<td>SDCOR</td>
<td>“SDCON, DDCON, SDCOR, and DDCOR (Convolution or Correlation with Decimated Output Using a Direct Method)” on page 1167</td>
</tr>
<tr>
<td>SACOR§</td>
<td>SACORF</td>
<td>“SACOR (Autocorrelation of One or More Sequences)” on page 1172</td>
</tr>
</tbody>
</table>

§ 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 197. List of Related-Computation Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPOL</td>
<td>DPOLY</td>
<td>“SPOLY and DPOLY (Polynomial Evaluation)” on page 1183</td>
</tr>
<tr>
<td>SIZC</td>
<td>DIZC</td>
<td>“SIZC and DIZC (I-th Zero Crossing)” on page 1186</td>
</tr>
<tr>
<td>STREC</td>
<td>DTREC</td>
<td>“STREC and DTREC (Time-Varying Recursive Filter)” on page 1189</td>
</tr>
<tr>
<td>SQINT</td>
<td>DQINT</td>
<td>“SQINT and DQINT (Quadratic Interpolation)” on page 1192</td>
</tr>
</tbody>
</table>
Table 197. List of Related-Computation Subroutines (continued)

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWLEV</td>
<td>DWLEV</td>
<td>“SWLEV, DWLEV, CWLEV, and ZWLEV (Wiener-Levinson Filter Coefficients)” on page 1196</td>
</tr>
<tr>
<td>CWLEV</td>
<td>ZWLEV</td>
<td></td>
</tr>
</tbody>
</table>

Fourier Transforms, Convolutions, and Correlations Considerations

This describes some global information applying to the Fourier transform, convolution, and correlation subroutines.

Use Considerations

This 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.

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 \( j = 0, 1, \ldots, n-1 \) rather than \( j = 1, 2, \ldots, n \). When using the formulas provided 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 different choices depending on which subroutine you are using:

- For subroutines in Table 198 all transform lengths between 0 and 1073479680 are acceptable.
- For subroutines in Table 199 on page 1028 you have two choices:
  - You can use the formula or table of values in “Acceptable Lengths for the Transforms” on page 1028 to choose a value.
  - Alternatively, 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 58.

Table 198. Fourier Transform subroutines allowing all lengths between 0 and 1073479680

<table>
<thead>
<tr>
<th>Subroutine Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCFTD, DCFTD</td>
</tr>
<tr>
<td>SRCFTD, DRCFTD</td>
</tr>
<tr>
<td>SCRFTD, DCRFTD</td>
</tr>
</tbody>
</table>
Table 199. Fourier Transform subroutines whose lengths are limited to those in Figure 15 on page 1029

<table>
<thead>
<tr>
<th>Subroutine Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCFT, DCFT</td>
</tr>
<tr>
<td>SCFTP</td>
</tr>
<tr>
<td>SRCFT, DRCFT</td>
</tr>
<tr>
<td>SCRFT, DCRFT</td>
</tr>
<tr>
<td>SCOSF, DCOSF</td>
</tr>
<tr>
<td>SCOSFT</td>
</tr>
<tr>
<td>SSINF, DSINF</td>
</tr>
<tr>
<td>SCFT2, DCFT2</td>
</tr>
<tr>
<td>SCFT2P</td>
</tr>
<tr>
<td>SRCFT2, DRCFT2</td>
</tr>
<tr>
<td>SCRFT2, DCRFT2</td>
</tr>
<tr>
<td>SCFT3, DCFT3</td>
</tr>
<tr>
<td>SCFT3P</td>
</tr>
<tr>
<td>SRCFT3, DRCFT3</td>
</tr>
<tr>
<td>SCRFT3, DCRFT3</td>
</tr>
</tbody>
</table>

Acceptable Lengths for the Transforms

Use the following formula to determine acceptable transform lengths:

\[ n = (2^h) (3^i) (5^j) (7^k) (11^m) \quad \text{for} \quad n \leq 37748736 \]

where:

- \( h = 1, 2, ..., 25 \)
- \( i = 0, 1, 2 \)
- \( j, k, m = 0, 1 \)

Figure 15 on page 1029 lists all the acceptable values for transform lengths in the Fourier transform subroutines.
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 aux, aux1, and aux2, respectively.

**AUX1:**

Chapter 12. Fourier Transforms, Convolutions and Correlations, and Related Computations
The \textit{aux1} array is used for storing tables and other parameters when you call a Fourier transform, convolution, or correlation subroutine for initialization with \textit{init} = 1. The initialized \textit{aux1} array is then used on succeeding calls with \textit{init} = 0, when the computation is actually done. You should not use this array between the initialization and the computation.

\textbf{AUX and AUX2:}
The \textit{aux} and \textit{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.

\textbf{AUX3:}
The \textit{aux3} argument is provided for migration purposes only and is ignored.

\textbf{Initializing Auxiliary Working Storage}
In many of those subroutines requiring \textit{aux1} auxiliary working storage, two invocations of the subroutines are necessary. The first invocation initializes the working storage in \textit{aux1} for the subroutine, and the second performs the computations. (For an explanation of auxiliary working storage, see \textit{“Understanding Auxiliary Working Storage Requirements” on page 1029.}) As a result, the working storage in \textit{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 \textit{aux1} array; that is, the initialized \textit{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 \textit{aux1} arrays for use with the different computations, saving and reusing them until they are not needed any more.

\textbf{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 \textit{“Using Auxiliary Storage in ESSL” on page 51.} Third, you can have ESSL dynamically allocate \textit{aux} and \textit{aux2}. For details, see \textit{“Dynamic Allocation of Auxiliary Storage” on page 52.}

\textbf{Performance and Accuracy Considerations}
The following explain the performance and accuracy considerations for the Fourier transforms, convolution, and correlation subroutines. For further details about performance and accuracy, see \textit{Chapter 2, “Planning Your Program,” on page 31.}
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 64.

Defining Arrays

The stride arguments, inc1h, inc1x, inc1y, inc2x, inc2y, incx, incy, incmx, incmy, 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 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.

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 1307.

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 xijk 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 xijk occupy the same array element, X(l), the subroutines have the following restriction: inc2 \( \equiv \) n1 and inc3 \( \equiv \) (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 implementations by factoring the transform length as follows:
  \[ N = N_1 N_2 N_3 \cdots N_p \]
  where each \( N_i \) is a power of 2; the power of 2 used depends on the machine model.

  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 \[1 on page 1363\], \[5 on page 1363\], and \[7 on page 1363\]).

- 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 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 \( 2^{36} - 1 \) for long-precision computations), the results are exact.

However, when short-precision subroutines use the Altivec or VSX unit to improve performance, they do not accumulate intermediate results in long precision.

**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 \([115 \text{ on page 1370}] \) and \([103 \text{ on page 1369}] \) 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 transform subroutines:

\[
nt = \min(nh+nx-1, \, ny) \text{ for convolution and correlation} \\
nt = \min(nx+nx-1, \, ny) \text{ for autocorrelation}
\]

which is also acceptable to the Fourier subroutines.

**Related Computation Considerations**
This 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 64.
Fourier Transform Subroutines

This contains the Fourier transform subroutine descriptions.
SCFTD and DCFTD (Multidimensional Complex Fourier Transform)

Purpose

These subroutines compute a set of \( m \) \( d \)-dimensional discrete Fourier transforms of complex data.

Table 200. Data Types

<table>
<thead>
<tr>
<th>( x, y )</th>
<th>( \text{scale} )</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>SCFTD</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>DCFTD</td>
</tr>
</tbody>
</table>

Notes:

1. Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SCFTD</th>
<th>DCFTD (init, d, x, incx, incmx, y, incy, incmy, n, m, isign, scale, aux1, naux1, aux2, naux2);</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>scftd</td>
<td>dcfld (init, d, x, incx, incmx, y, incy, incmy, n, m, isign, scale, aux1, naux1, aux2, naux2);</td>
</tr>
</tbody>
</table>

On Entry

\( \text{init} \)

is a flag, where:

If \( \text{init} = 1 \), trigonometric functions and other parameters, depending on arguments other than \( x \), are computed and saved in \( \text{aux1} \). The contents of \( x \) and \( y \) are not used or changed.

If \( \text{init} = 2 \), trigonometric functions and other parameters, depending on arguments other than \( x \), are computed and saved in \( \text{aux1} \), and no SIMD algorithms are used (see “What ESSL Library Do You Want to Use?” on page 31). The contents of \( x \) and \( y \) are not used or changed.

If \( \text{init} = 0 \), the discrete Fourier transforms of the given array is computed. The only arguments that may change after initialization are \( x, y, \) and \( \text{aux2} \). The arguments \( d, \text{incx}, \text{incmx}, \text{incy}, \text{incmy}, n, m, \text{isign}, \text{scale}, \text{aux1}, \text{naux1}, \text{aux2}, \text{naux2} \) must be the same as when the subroutine was called for initialization with \( \text{init} = 1 \) or \( \text{init} = 2 \).

Specified as: an integer; \( 0 \leq \text{init} \leq 2 \).

\( d \)

is the dimension of the transform.

Specified as: an integer; \( 1 \leq d \leq 3 \).

\( x \)

is the array \( x \), consisting of \( m \) sequences of \( d \)-dimensional complex arrays to be transformed. Using zero-based indexing, \( x_{j1,j2,...,jm} \) is stored in location \( j1(\text{incx}_1) + j2(\text{incx}_2) + ... + jd(\text{incx}_d) + mn(\text{incmx}) \) of the array \( x \).

Specified as: an array of (at least) length \( 1 + \text{incx}(n-1) + ... + \text{incx}(n-1) + \text{incmx}(m-1) \), containing numbers of the data type indicated in [Table 200](#)
\textbf{incx}

is an array containing the strides between the elements in array \( X \) for each of the \( d \) dimensions.

Specified as: an array of length \( d \) containing integers; \( incx_{1:d} > 0 \).

\textbf{incmx}

is the stride between the first elements of the \( d \)-dimensional sequences in array \( X \). (If \( m = 1 \), this argument is ignored.)

Specified as: an integer; \( incmx > 0 \).

\textbf{incy}

is an array containing the strides between the elements in array \( Y \) for each of the \( d \) dimensions.

Specified as: an array of length \( d \) containing integers; \( incy_{1:d} > 0 \).

\textbf{incmy}

is the stride between the first elements of the \( d \)-dimensional sequences in array \( Y \). (If \( m = 1 \), this argument is ignored.)

Specified as: an integer; \( incmy > 0 \).

\textbf{n}

is an array containing the lengths of the dimensions of the array to be transformed.

Specified as: an array of length \( d \) containing integers; \( 0 \leq n_{1:d} \leq 1073479680 \).

\textbf{m}

is the number of sequences to be transformed.

Specified as: an integer; \( m > 0 \).

\textbf{isign}

is an array that controls the direction of the transform (from time to frequency or from frequency to time). The sign of \( isign_i \) determines the signs in the exponents of \( W_{n_1}, W_{n_2}, \ldots, W_{n_d} \) where:

If \( isign_i > 0 \), \( isign_i = + \) (transforming time to frequency).

If \( isign_i < 0 \), \( isign_i = - \) (transforming frequency to time).

Specified as: an array of length \( d \) containing integers; \( isign_{1:d} \neq 0 \).

\textbf{scale}

is the scaling constant by which the transforms are multiplied. See \textbf{“Function” on page 1040} for its usage.

Specified as: a number of the data type indicated in \textbf{Table 200 on page 1036}, where \( scale \neq 0.0 \).

\textbf{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.

\textbf{naux1}

is the number of doublewords in the working storage specified in \( aux1 \).

Specified as: an integer; \( naux1 > 7(d+1)+1 \) and \( naux1 \geq (\text{minimum value required for successful processing}) \). To determine a sufficient value, use the
Processor-independent formulas (see Processor-Independent Formulas for SCFTD for NAUX1 and NAUX2 and Processor-Independent Formulas for DCFTD for NAUX1 and NAUX2). For values between $7(d+1)+1$ and the minimum value, you have the option of having the minimum value returned in this argument; for details, see On Return and “Using Auxiliary Storage in ESSL” on page 51.

**aux2**

has the following meaning:

If $\text{aux2} = 0$ 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: an integer, where:

If $\text{naux2} = 0$ and error 2015 is unrecoverable, the subroutine dynamically allocates the work area. The work area is deallocated before control is returned to the calling program.

Otherwise, $\text{naux2} \geq$ (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 On Return and “Using Auxiliary Storage in ESSL” on page 51.

On Return

**y** has the following meaning, where:

If $\text{init} > 0$, this argument is not used, and its contents remain unchanged.

If $\text{init} = 0$, this is array $Y$, consisting of the results of the $m d$-dimensional discrete Fourier transforms. Using zero-based indexing, $y_{k_1,k_2,...,k_d,mm}$ is stored in location $k_1(\text{incy}_1) + k_2(\text{incy}_2) + ... + k_d(\text{incy}_d) + mm(\text{incmy})$ of the array $Y$.

Returned as: an array of (at least) length $1 + \text{incy}_1(n_1-1) + ... + \text{incy}_d(n_d-1) + \text{incmy}(m-1)$, containing numbers of the data type indicated in Table 200 on page 1036.

**aux1**

is the working storage for this subroutine, where:

If $\text{init} > 0$, it contains information ready to be passed in a subsequent invocation of this subroutine.

If $\text{init} = 0$, its contents are unchanged.

Returned as: the contents are not relevant.

**naux1**

contains the minimum value required for successful processing (as returned by the subroutine), provided that the following are true:

- You specified that error 2015 is recoverable.
- You specified an input value for naux1 that is at least $7(d+1)+1$ (but insufficient for the problem).
- There were no other errors.
Otherwise, it remains unchanged.

Returned as: an integer.

\textit{naux2}

contains the minimum value required for successful processing (as returned by the subroutine), provided that the following are true:

\begin{itemize}
  \item You specified that error 2015 is recoverable.
  \item You specified an input value for \textit{naux2} that is greater than or equal to zero (but insufficient for the problem).
  \item There were no other errors.
\end{itemize}

Otherwise, it remains unchanged.

Returned as: an integer.

\section*{Notes}

1. \textit{aux1} should not be used by the calling program between calls to this subroutine with \textit{init} > 0 and \textit{init} > 0. However, it can be reused after intervening calls to this subroutine with different arguments.

2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for \textit{init} > 0 and \textit{init} = 0.

3. For optimal performance, the preferred value for \textit{incx} and \textit{incy} is 1.

   If you specify the same array for \textit{X} and \textit{Y}, then \textit{incx} and \textit{incy}, for \(i = 1, \ldots, d\) must be equal, and \textit{incmx} and \textit{incmy} must be equal. In this case, output overwrites input. If you specify different arrays for \textit{X} and \textit{Y}, they must have no common elements; otherwise, results are unpredictable. See “Concepts” on page 75.

4. You have the option of having the minimum required value for \textit{naux1} and \textit{naux2} dynamically returned to your program. For details, see “Using Auxiliary Storage in ESSL” on page 51.

\section*{Formulas}

\subsection*{Processor-Independent Formulas for SCFTD for N\textit{AUX1} and N\textit{AUX2}:

\textbf{NAUX1 Formulas}}

If \(\max(n_1, n_2, \ldots, n_d) \leq 2048\), \textit{naux1} = 30000d.

If \(\max(n_1, n_2, \ldots, n_d) > 2048\), \textit{naux1} = 60000d + 14.12(n_1 + \ldots + n_d).

\textbf{NAUX2 Formulas}

If \(\max(n_1, n_2, \ldots, n_d) < 252\), \textit{naux2} = 20000.

If \(\max(n_1, n_2, \ldots, n_d) \geq 252\), \textit{naux2} = 20000 + (r + 256)(s + 8.56).

where:

\(r = \max(n_1, n_2, \ldots, n_d)\) and

\(s = \min(64, r)\)

\subsection*{Processor-Independent Formulas for DCFTD for N\textit{AUX1} and N\textit{AUX2}:

\textbf{NAUX1 Formulas}}

If \(\max(n_1, n_2, \ldots, n_d) \leq 1024\), \textit{naux1} = 30000d.

If \(\max(n_1, n_2, \ldots, n_d) > 1024\), \textit{naux1} = 60000d + 28.24(n_1 + \ldots + n_d).

\textbf{NAUX2 Formulas}
If \( \max(n_1, n_2, \ldots, n_d) < 252 \), \( naux2 = 20000 \).

If \( \max(n_1, n_2, \ldots, n_d) \geq 252 \), \( naux2 = 20000 + (2r + 256)(s + 17.12) \).

where:

\[ r = \max(n_1, n_2, \ldots, n_d) \]  
\[ s = \min(64, r) \]

**Function**

The set of \( m \) \( d \)-dimensional discrete Fourier transforms of complex data in array \( X \), with results going into array \( Y \), is expressed as follows:

\[
Y_{k_1,k_2,\ldots,k_d, i} = scale \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_d=0}^{n_d-1} x_{j_1,j_2,\ldots,j_d, i} W_{n_1}^{(isign_1)j_1} W_{n_2}^{(isign_2)j_2} \cdots W_{n_d}^{(isign_d)j_d}
\]

for:

\[ k_1 = 0, \ldots, n_1 - 1 \]
\[ k_2 = 0, \ldots, n_2 - 1 \]
\[ \ldots \]
\[ k_d = 0, \ldots, n_d - 1 \]
\[ i = 0, \ldots, m - 1 \]

where:

\[
W_{n_l} = e^{-2\pi(i/(n_l - 1))}
\]

for:

\[ l = 1, \ldots, d \]

and where:

\( x_{j_1,j_2,\ldots,j_d, i} \) are elements of the \( d \)-dimensional sequences in array \( X \). 
\( y_{k_1,k_2,\ldots,k_d, i} \) are elements of the \( d \)-dimensional sequences in array \( Y \).

For \( scale = 1.0 \) and \( isign_1 = isign_2 = \ldots = isign_d = 1 \), you obtain the discrete Fourier transform (DFT), a function of frequency. The inverse Fourier transform is obtained with \( scale = 1.0/n_1 \ n_2 \ldots n_d \) and \( isign_1 = isign_2 = \ldots = isign_d = -1 \). See references [5 on page 1363], [7 on page 1363], [12 on page 1364], and [31 on page 1365].

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.
If \( n_i = 0 \) for any \( i \) from 1 to \( d \) or if \( m = 0 \); no initialization or computation is performed.

**Error conditions**

**Resource Errors**

Error 2015 is unrecoverable, \( naux2 = 0 \), and unable to allocate work area.

**Computational Errors**

None

**Input-Argument Errors**

1. \( init < 0 \) or \( init > 2 \)
2. \( d < 1 \) or \( d > 3 \)
3. \( incx_i \leq 0 \) (\( i = 1,...,d \))
4. \( incm \leq 0 \)
5. \( incy_i \leq 0 \) (\( i = 1,...,d \))
6. \( incmy \leq 0 \)
7. \( n_i < 0 \) or \( n_i > 1073479680 \) (\( i = 1,...,d \))
8. \( m < 0 \)
9. \( isign_i = 0 \) (\( i = 1,...,d \))
10. \( scale = 0.0 \)
11. \( naux1 \leq 7(d+1)+1. \)
12. \( naux1 \) is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
13. 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.
14. The subroutine has not been initialized with the present arguments.

**Examples**

**Example 1**

This example shows an input array \( X \) with a set of four long-precision complex sequences:

\[
e^{2\pi jk/n}
\]

for \( j = 0, 1, ..., n-1 \) with \( n = 8 \), and the single frequencies \( k = 0, 1, 2, \) and 3.

**Note:** \( X \) is the same input array used in Example 1

The arrays are declared as follows:

```plaintext
COMPLEX*16 X(0:31),Y(0:31)
REAL*8 AUX1(10000),AUX2(1)
```

First, initialize \( AUX1 \) using the calling sequence shown below with \( INIT \neq 0 \). Then use the same calling sequence with \( INIT = 0 \) to do the calculation.

**Note:** Because \( NAUX2 = 0 \), this subroutine dynamically allocates the \( AUX2 \) working storage.

Call Statement and Input:
INIT = 1 (for initialization)
INIT = 0 (for computation)
INCY is an array of length d.
INCY(1) = 1
INCY is an array of length d.
INCY(1) = 1
N is an array of length d.
N(1) = 8
ISIGN is an array of length d.
ISIGN(1) = 1
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) (-1.0000, 0.0000) ( 1.0000, 0.0000) (-0.0000, 0.0000)
(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) ( 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, 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, 8.0000)
(0.0000, 0.0000) ( 0.0000, 0.0000) ( 0.0000, 0.0000) ( 0.0000, 8.0000)

Example 2
This example shows how to compute a three-dimensional transform. In this example, INCX ≥ INCY, so the same array can be used for both input and output.

Note: X is the same input array used in Example 1

The STRIDE subroutine is called to select good values for the INCY strides. (As explained below, STRIDE is not called for INCX.) Using the transform lengths (N(1) = 32, N(2) = 64, and N(3) = 40) along with the output data type (short-precision complex: 'C'), STRIDE is called once for each stride needed. First, it is called for INCY(2):

CALL STRIDE (N(2),N(1),INCY(2),'C',0)

The output value returned for INCY(2) is 32. Then STRIDE is called again for INCY(3):

CALL STRIDE (N(3),N(2)*INCY(2),INCY(3),'C',0)

The output value returned for INCY(3) is 2056. Because INCY(3) is not a multiple of INCY(2), Y is not declared as a three-dimensional array; it is declared as a two-dimensional array, Y(INCY(3),N(3)).
For equivalence, it is required that \( \text{INCX}(2) \geq \text{INCY}(2) \) and \( \text{INCX}(3) \geq \text{INCY}(3) \). Therefore, \( \text{INCX}(2) \) and \( \text{INCY}(2) \) are set as follows: \( \text{INCX}(2) = \text{INCY}(2) = 32 \).

To enable the \( X \) array to be declared as a three-dimensional array, \( \text{INCX}(3) \) must be a multiple of \( \text{INCX}(2) \). Therefore, its value is set as \( \text{INCX}(3) = 65(\text{INCX}(2)) = 2080 \).

The arrays are declared as follows:

\[
\text{COMPLEX*8} \quad X(32,65,40), Y(2056,40) \\
\text{REAL*8} \quad \text{AUX1}(90000), \text{AUX2}(1), \text{SCALE}
\]

Arrays \( X \) and \( Y \) are made equivalent by the following statement, making them occupy the same storage:

\[
\text{EQUIVALENCE} \ (X,Y)
\]

**Note:** Because \( \text{NAUX2} = 0 \), this subroutine dynamically allocates the \( \text{AUX2} \) working storage.

**Call Statement and Input:**

\[
\text{CALL SCFTD(INIT, 3, X, INCX, 0, Y, INCMX, 0, N, 1, ISIGN, 1.0, AUX1, NAUX1, AUX2, NAUX2)}
\]

\[
\begin{align*}
\text{INIT} & = 1 \quad \text{(for initialization)} \\
\text{INIT} & = 0 \quad \text{(for computation)} \\
\text{INCX} & \text{ is an array of length } d. \\
\text{INCX}(1) & = 1 \\
\text{INCX}(2) & = 32 \\
\text{INCX}(3) & = 2080 \\
\text{INCMX} & \text{ is an array of length } d. \\
\text{INCMX}(1) & = 1 \\
\text{INCMX}(2) & = 32 \\
\text{INCMX}(3) & = 2056 \\
\text{N} & \text{ is an array of length } d. \\
\text{N}(1) & = 32 \\
\text{N}(2) & = 64 \\
\text{N}(3) & = 40 \\
\text{ISIGN} & \text{ is an array of length } d. \\
\text{ISIGN}(1) & = 1 \\
\text{ISIGN}(2) & = 1 \\
\text{ISIGN}(3) & = 1 \\
\text{SCALE} & = 1.0
\end{align*}
\]

\( X \) 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 all locations.
SRCFTD and DRCFTD (Multidimensional Real-to-Complex Fourier Transform)

**Purpose**

These subroutines compute a set of $m$ $d$-dimensional complex discrete Fourier transforms of real data.

**Table 201. Data Types**

<table>
<thead>
<tr>
<th>$X$, $scale$</th>
<th>$Y$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision complex</td>
<td>SRCFTD</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision complex</td>
<td>DRCFTD</td>
</tr>
</tbody>
</table>

**Notes:**

1. Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.

2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

**Syntax**

**Fortran**

CALL SRCFTD | DRCFTD (init, d, x, incx, incmx, y, incy, incmy, n, m, isign, scale, aux1, naux1, aux2, naux2)

**C and C++**

srcftd | drcftd (init, d, x, incx, incmx, y, incy, incmy, n, m, isign, scale, aux1, naux1, aux2, naux2);

**On Entry**

`init`

is a flag, where:

If $init = 1$, 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 $init = 2$, trigonometric functions and other parameters, depending on arguments other than $x$, are computed and saved in $aux1$, and no SIMD algorithms are used (see “What ESSL Library Do You Want to Use?” on page 31). The contents of $x$ and $y$ are not used or changed.

If $init = 0$, the discrete Fourier transforms of the given array are computed. The only arguments that may change after initialization are $x$, $y$, and $aux2$. The arguments $d$, $incx$, $incmx$, $incy$, $incmy$, $n$, $m$, $isign$, $scale$, $aux1$, $naux1$, and $naux2$ must be the same as when the subroutine was called for initialization with $init = 1$ or $init = 2$.

Specified as: an integer; $0 \leq init \leq 2$.

$d$

is the dimension of the transform.

Specified as: an integer; $1 \leq d \leq 3$.

$x$

is the array $X$, consisting of $m$ sequences of $d$-dimensional complex arrays to be transformed. Using zero-based indexing, $X_{1,2,...,d,mm}$ is stored in location $j1(incx_1) + j2(incx_2) + ... + jd(incx_d) + mm(incmx)$ of the array $X$. 

1044    ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
Specified as: an array of (at least) length 1 + incx1(n-1) + ... + incx_d(n-1) + incmx(m-1), containing numbers of the data type indicated in Table 201 on page 1044.

*incx*

is an array containing the strides between the elements in array X for each of the d dimensions.

Specified as: an array of length d containing integers; incx_1,d > 0.

*incmx*

is the stride between the first elements of the d-dimensional sequences in array X. (If m = 1, this argument is ignored.)

Specified as: an integer; incmx > 0.

*y* See "On Return"

*incy*

is an array containing the strides between the elements in array Y for each of the d dimensions.

Specified as: an array of length d containing integers; incy_1,d > 0.

*incmy*

is the stride between the first elements of the d-dimensional sequences in array Y. (If m = 1, this argument is ignored.)

Specified as: an integer; incmy > 0.

*n*

is an array containing the lengths of the dimensions of the array to be transformed.

Specified as: an array of length d containing integers; 0 ≤ n_1,d ≤ 1073479680.

*m*

is the number of sequences to be transformed.

Specified as: an integer; m > 0.

*isign*

is an array that controls the direction of the transform (from time to frequency or from frequency to time). The sign of isign_i determines the signs in the exponents of W_n1, W_n2, ..., W_n_d, where:

If isign_i > 0, isign_i = + (transforming time to frequency).

If isign_i < 0, isign_i = - (transforming frequency to time).

Specified as: an array of length d containing integers; isign_1,d ≠ 0.

*scale*

is the scaling constant by which the transforms are multiplied. See “Function” on page 1048 for its usage.

Specified as: a number of the data type indicated in Table 201 on page 1044, where 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: an integer; *naux1* > \((4d + 11)\) and *naux1* \(\geq\) (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas (see “Formulas” on page 1047). For values between \((4d + 11)\) and the minimum value, you have the option of having the minimum value returned in this argument; for details, see "On Return" and “Using Auxiliary Storage in ESSL” on page 51.

**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: an integer, where:

If *naux2* = 0 and error 2015 is unrecoverable, the subroutine dynamically allocates the work area. The work area is deallocated before control is returned to the calling program.

Otherwise, *naux2* \(\geq\) (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 "On Return" and “Using Auxiliary Storage in ESSL” on page 51.

**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* *d*-dimensional complex discrete Fourier transforms. Using zero-based indexing, *y*\(_{k1,k2,...,kd,mm}\) is stored in location *k1*\(incy_1\) + *k2*\(incy_2\) + ... + *kd*\(incy_d\) + *mm*\(incmy\) of the array *Y*. Due to complex conjugate symmetry, the output consists of only the first \(n_1/2+1\) values along the first dimension of the array, for \(k1 = 0, 1, ..., n_1/2\).

Returned as: an array of (at least) length \(1 + incy_1(n_1-1) + ... + incy_d(n_d-1) + incmy(m-1)\), containing numbers of the data type indicated in Table 201 on page 1044.

**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.

**naux1**

contains the minimum value required for successful processing (as returned by the subroutine), provided that the following are true:
• You specified that error 2015 is recoverable.
• You specified an input value for \( naux1 \) that is at least \((4d+11)\) (but insufficient for the problem).
• There were no other errors.
Otherwise, it remains unchanged.
Returned as: an integer.

\( naux2 \)
contains the minimum value required for successful processing (as returned by the subroutine), provided that the following are true:
• You specified that error 2015 is recoverable.
• You specified an input value for \( naux2 \) that is greater than or equal to zero (but insufficient for the problem).
• There were no other errors.
Otherwise, it remains unchanged.
Returned as: an integer.

**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. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for \( init > 0 \) and \( init = 0 \).
3. For optimal performance, the preferred value for \( incx_1 \) and \( incy_1 \) is 1.
   If you specify the same array for \( X \) and \( Y \), then:
   • \( incx_i \) must equal \( 2(incy_i) \), for \( i = 2,\ldots,d \)
   • \( incmx \) must be equal to \( 2(incmy) \) if \( m > 1 \)
   In this case, output overwrites input. If you specify different arrays for \( X \) and \( Y \), they must have no common elements; otherwise, results are unpredictable. See “Concepts” on page 75.
4. You have the option of having the minimum required value for \( naux1 \) and \( naux2 \) dynamically returned to your program. For details, see “Using Auxiliary Storage in ESSL” on page 51.

**Formulas**

**Processor-Independent Formulas for SRCFTD for NAUX1 and NAUX2:**

**NAUX1 Formulas**

If \( \max(n_1,n_2,\ldots,n_d) \leq 2048 \), \( naux1 = 60000d \).
If \( \max(n_1,n_2,\ldots,n_d) > 2048 \), \( naux1 = 60000d+14.12(n_1+\ldots+n_d) \).

**NAUX2 Formulas**

If \( \max(n_1,n_2,\ldots,n_d) < 252 \), \( naux2 = 20000 \).
If \( \max(n_1,n_2,\ldots,n_d) \geq 252 \), \( naux2 = 20000+(r+256)(s+8.56) \).

where:

\( r = \max(n_1,n_2,\ldots,n_d) \) and
\( s = \min(64,r) \)
Processor-Independent Formulas for DRCFTD for NAUX1 and NAUX2:

NAUX1 Formulas

If $\max(n_1, n_2, \ldots, n_d) \leq 1024$, $\text{naux1} = 60000d$.
If $\max(n_1, n_2, \ldots, n_d) > 1024$, $\text{naux1} = 60000d + 28.24(n_1 + \ldots + n_d)$.

NAUX2 Formulas

If $\max(n_1, n_2, \ldots, n_d) < 252$, $\text{naux2} = 20000$.
If $\max(n_1, n_2, \ldots, n_d) \geq 252$, $\text{naux2} = 20000 + (2r + 256)(s + 17.12)$.

where:

$r = \max(n_1, n_2, \ldots, n_d)$ and
$s = \min(64, r)$

Function

The set of $m$ $d$-dimensional complex conjugate even discrete Fourier transforms of real data in array $x$ with results going into array $y$ is expressed as follows:

$$y_{k_1, k_2, \ldots, k_d} = \text{scale} \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \ldots \sum_{j_d=0}^{n_d-1} x_{j_1, j_2, \ldots, j_d} W_{n_1}^{(\text{isign}_1)(j_1)(k_1)} W_{n_2}^{(\text{isign}_2)(j_2)(k_2)} \ldots W_{n_d}^{(\text{isign}_d)(j_d)(k_d)}$$

for:

$k_1 = 0, \ldots, n_1 - 1$
$k_2 = 0, \ldots, n_2 - 1$
. .
. .
kd = 0, ..., nd - 1
i = 0, ..., m - 1

where:

$$W_{n_l} = e^{-2\pi i (j_l - 1)/n_l}$$

for:

l = 1, ..., d

and where:

$x_{j_1, j_2, \ldots, j_d, m}$ are elements of the $d$-dimensional sequences in array $X$.
$y_{k_1, k_2, \ldots, k_d, m}$ are elements of the $d$-dimensional sequences in array $Y$.

For $\text{scale} = 1.0$ and $\text{isign}_1 = \text{isign}_2 = \ldots = \text{isign}_d = 1$, you obtain the discrete Fourier transform (DFT), a function of frequency. The inverse Fourier transform is obtained with $\text{scale} = 1.0/n_1 n_2 \ldots n_d$ and $\text{isign}_1 = \text{isign}_2 = \ldots = \text{isign}_d = -1$. See references [5 on page 1363], [7 on page 1363], [12 on page 1364], and [31 on page 1365].
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.

If \( n_i = 0 \) for any \( i \) from 1 to \( d \) or if \( m = 0 \); no initialization or computation is performed.

**Error conditions**

**Resource Errors**

Error 2015 is unrecoverable, unable to allocate work area, and internal deallocation error.

**Computational Errors**

None

**Input-Argument Errors**

1. \( init < 0 \) or \( init > 2 \)
2. \( d < 1 \) or \( d > 3 \)
3. \( incx_i \leq 0 \) (\( i = 1,\ldots,d \))
4. \( incmx \leq 0 \)
5. \( incy_i \leq 0 \) (\( i = 1,\ldots,d \))
6. \( incmy \leq 0 \)
7. \( n_i < 0 \) or \( n_i > 1073479680 \) (\( i = 1,\ldots,d \))
8. \( m < 0 \)
9. \( isign_i = 0 \) (\( i = 1,\ldots,d \))
10. \( scale = 0.0 \)
11. \( naux1 \leq 4d+11 \).
12. \( naux1 \) is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
13. 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.
14. The subroutine has not been initialized with the present arguments.

**Examples**

**Example 1**

This example shows an input array \( X \) with a set of \( m \) cosine sequences \( \cos(2\pi jk/n) \), \( j = 0, 1, \ldots, 15 \) 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:

\[
\begin{align*}
\text{REAL*4} & \quad (X(0:100)) \\
\text{COMPLEX*8} & \quad (Y(0:50)) \\
\text{REAL*8} & \quad \text{AUX1(1000)}, \text{AUX2}
\end{align*}
\]

First, initialize \( AUX1 \) using the calling sequence shown below with \( INIT \neq 0 \).

Then use the same calling sequence with \( INIT = 0 \) to do the calculation.
Note: Because NAUX2 = 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

<table>
<thead>
<tr>
<th>INIT</th>
<th>D</th>
<th>X</th>
<th>INCR</th>
<th>INCM</th>
<th>Y</th>
<th>INCR</th>
<th>INCM</th>
<th>N</th>
<th>M</th>
<th>ISIGN</th>
<th>SCALE</th>
<th>AUX1</th>
<th>NAUX1</th>
<th>AUX2</th>
<th>NAUX2</th>
</tr>
</thead>
</table>
| CALL SRCFTD(INIT, 1, X, INCR, 16, Y, INCR, 9, N, 4, ISIGN, SCALE, AUX1, 1000, AUX2, 0)

INIT = 1 (for initialization)
INIT = 0 (for computation)
INCY is an array of length d.
INCY(1) = 1
INCR is an array of length d.
INCR(1) = 1
N is an array of length d.
N(1) = 16
ISIGN is an array of length d.
ISIGN(1) = 1
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

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)

Example 2

This example shows how to compute a three-dimensional transform.

The STRIDE subroutine is called to select good values for the INCY strides (as the following explains, STRIDE is not called for INCR.) Using the transform lengths (N(1) = 33, N(2) = 64, and N(3) = 40) along with the output data type (short-precision complex: 'C'), STRIDE is called once for each stride needed. First, it is called for INCY(2):
CALL STRIDE (N(2),N(1)/2+1,INCY(2), 'C', 0)
The output value returned for INCY(2) is 18. Then STRIDE is called again for INCY(3):

```call
    call stride (n(3), n(2)*inCY(2), incY(3), 'c', 0)
```

The output value returned for INCY(3) is 1160. Because INCY(3) is not a multiple of INCY(2), Y is not declared as a three-dimensional array; it is declared as a two-dimensional array, Y(INCY(3),N(3)).

For equivalence, it is required that INCX(2) = 2(INCY(2)) and INCX(3) = 2(INCY(3)). Therefore, INCX(2), INCX(2), INCX(3) and INCY(3) are set as follows:

| INCY(2) | 18 |
| INCX(2) | 36 |
| INCY(3) | 1160 |
| INCX(3) | 2320 |

The arrays are declared as follows:

```c
REAL*4 X(2320,40), SCALE
COMPLEX*8 Y(1160,40)
REAL*8 AUX1(5000), AUX2
```

Arrays X and Y are made equivalent by the following statement, making them occupy the same storage:

```c
    equivalence (x,y)
```

**Note:** Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```c
    call srcftd(init, 3 x, incx, 0, y, incy, 0 n, 1, isign, 1.0, aux1, 5000, aux2, 0)
```

**INIT** = 1 (for initialization)
**INIT** = 0 (for computation)
**INCY** is an array of length \(d\).
**INCY(1)** = 1
**INCY(2)** = 18
**INCY(3)** = 1160

**INCMX** is an array of length \(d\).
**INCMX(1)** = 1
**INCMX(2)** = 36
**INCMX(3)** = 2320

**INCMY** is an array of length \(d\).
**INCMY(1)** = 1
**INCMY(2)** = 18
**INCMY(3)** = 1160

**N** is an array of length \(d\).
**N(1)** = 33
**N(2)** = 64
**N(3)** = 40

**ISIGN** is an array of length \(d\).
**ISIGN(1)** = 1
**ISIGN(2)** = 1
**ISIGN(3)** = 1

**SCALE** = 1.0

\(X\) has 1.0 in location \(X(1,1)\) and 0.0 in all other locations.

Output:

\(Y\) has \((1.0,0.0,0.0)\) in all locations.
**SCRFTD and DCRFTD (Multidimensional Complex-to-Real Fourier Transform)**

**Purpose**

These subroutines compute a set of $m$ $d$-dimensional real discrete Fourier transforms of complex conjugate even data.

<table>
<thead>
<tr>
<th>Table 202. Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$, scale</td>
</tr>
<tr>
<td>Y</td>
</tr>
<tr>
<td>Subroutine</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
</tbody>
</table>

**Notes:**

1. Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

**Syntax**

<table>
<thead>
<tr>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL SCRFTD $\mid$ DCRFTD (init, d, x, incx, incmx, y, incy, incmy, n, m, isign, scale, aux1, auxx1, auxx2, aux2)</td>
</tr>
<tr>
<td>C and C++</td>
</tr>
<tr>
<td>scrftd $\mid$ dcrftd (init, d, x, incx, incmx, y, incy, incmy, n, m, isign, scale, aux1, auxx1, auxx2, aux2);</td>
</tr>
</tbody>
</table>

**On Entry**

- **init**
  - is a flag, where:
    - If $init = 1$, 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 $init = 2$, trigonometric functions and other parameters, depending on arguments other than $x$, are computed and saved in $aux1$, and no SIMD algorithms are used (see “What ESSL Library Do You Want to Use?” on page 31). The contents of $x$ and $y$ are not used or changed.
    - If $init = 0$, the discrete Fourier transforms of the given array are computed. The only arguments that may change after initialization are $x$, $y$, and $aux2$. The arguments $d$, incx, incmx, incy, incmy, n, m, isign, scale, aux1, auxx1, and auxx2 must be the same as when the subroutine was called for initialization with $init = 1$ or $init = 2$.
  - Specified as: an integer; $0 \leq init \leq 2$.

- **d**
  - is the dimension of the transform.
  - Specified as: an integer; $1 \leq d \leq 3$.

- **x**
  - is the array $X$, consisting of $m$ sequences of $d$-dimensional complex arrays to be transformed. Using zero-based indexing, $x_{1,2,...,d,mm}$ is stored in location $j1(incx_1) + j2(incx_2) + ... + jd(incx_d) + mm(incmx)$ of the array $X$. Due to complex
conjugate symmetry, the output consists of only the first \(n_1/2+1\) values along the first dimension of the array, for \(j_1 = 0, 1, \ldots, n_1/2\).

Specified as: an array of (at least) length \(1 + \text{incx}_1(n_1-1) + \ldots + \text{incx}_d(n_d-1) + \text{incmx}(m-1)\), containing numbers of the data type indicated in Table 202 on page 1052.

\textit{incx}

is an array containing the strides between the elements in array \(X\) for each of the \(d\) dimensions.

Specified as: an array of length \(d\) containing integers; \(\text{incx}_{1:d} > 0\).

\textit{incmx}

is the stride between the first elements of the \(d\)-dimensional sequences in array \(X\). (If \(m = 1\), this argument is ignored.)

Specified as: an integer; \(\text{incmx} > 0\).

\(y\) See On Return

\textit{incy}

is an array containing the strides between the elements in array \(Y\) for each of the \(d\) dimensions.

Specified as: an array of length \(d\) containing integers; \(\text{incy}_{1:d} > 0\).

\textit{incmy}

is the stride between the first elements of the \(d\)-dimensional sequences in array \(Y\). (If \(m = 1\), this argument is ignored.)

Specified as: an integer; \(\text{incmy} > 0\).

\(n\)

is an array containing the lengths of the dimensions of the array to be transformed.

Specified as: an array of length \(d\) containing integers; 0 \(\leq n_{1:d} \leq 1073479680\).

\(m\)

is the number of sequences to be transformed.

Specified as: an integer; \(m > 0\).

\textit{isign}

is an array that controls the direction of the transform (from time to frequency or from frequency to time). The sign of \(\text{isign}_i\) determines the signs in the exponents of \(W_{n_1}, W_{n_2}, \ldots, W_{n_d}\) where:

If \(\text{isign}_i > 0\), \(\text{isign}_i = +\) (transforming time to frequency).

If \(\text{isign}_i < 0\), \(\text{isign}_i = -\) (transforming frequency to time).

Specified as: an array of length \(d\) containing integers; \(\text{isign}_{1:d} \neq 0\).

\textit{scale}

is the scaling constant by which the transforms are multiplied. See “Function” on page 1056 for its usage.

Specified as: a number of the data type indicated in Table 202 on page 1052, where \(\text{scale} \neq 0.0\).

\textit{aux1}

is the working storage for this subroutine, where:

If \(\text{init} > 0\), the working storage is computed.

If \(\text{init} = 0\), the working storage is used in the computation of the Fourier transforms.
Specified as: an area of storage, containing \textit{naux1} long-precision real numbers.

\textit{naux1}

is the number of doublewords in the working storage specified in \textit{aux1}.

Specified as: an integer; \textit{naux1} > \((4d + 11)\) and \textit{naux1} \(\geq\) (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas (see “Formulas” on page 1056). For values between \((4d + 11)\) and the minimum value, you have the option of having the minimum value returned in this argument; for details, see \textit{On Return} and “Using Auxiliary Storage in ESSL” on page 51.

\textit{aux2}

has the following meaning:

If \textit{naux2} = 0 and error 2015 is unrecoverable, \textit{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 \textit{naux2} long-precision real numbers.

On output, the contents are overwritten.

\textit{naux2}

is the number of doublewords in the working storage specified in \textit{aux2}.

Specified as: an integer, where:

If \textit{naux2} = 0 and error 2015 is unrecoverable, the subroutine dynamically allocates the work area. The work area is deallocated before control is returned to the calling program.

Otherwise, \textit{naux2} \(\geq\) (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 \textit{On Return} and “Using Auxiliary Storage in ESSL” on page 51.

\textbf{On Return}

\textit{x}

has the following meaning, where:

If \textit{init} > 0, this argument is not used, and its contents remain unchanged.

If \textit{init} = 0, this argument is not used, and its contents remain unchanged if one of the following is true:

\begin{itemize}
\item \(d = 1\)
\item \(\text{incx}_i = 1\) and \(\text{incy}_i = 1\)
\end{itemize}

Otherwise, \textit{x} is overwritten; that is, the original input is not preserved.

\textit{y}

has the following meaning, where:

If \textit{init} > 0, this argument is not used, and its contents remain unchanged.

If \textit{init} = 0, this is array \textit{Y}, consisting of the results of the \(m\) \(d\)-dimensional discrete Fourier transforms of complex conjugate even data. Using zero-based indexing, \(y_{k1,k2,...,kd,mm}\) is stored in location \(k1(\text{incy}_1) + k2(\text{incy}_2) + ... + kd(\text{incy}_d) + mm(\text{incmy})\) of the array \textit{Y}.

Returned as: an array of (at least) length \(1 + \text{incy}_1(n_1-1) + ... + \text{incy}_d(n_d-1) + \text{incmy}(m-1)\), containing numbers of the data type indicated in Table 202 on page 1052.
**aux1**

is the working storage for this subroutine, where:

If \( \text{init} > 0 \), it contains information ready to be passed in a subsequent invocation of this subroutine.

If \( \text{init} = 0 \), its contents are unchanged.

Returned as: the contents are not relevant.

**naux1**

contains the minimum value required for successful processing (as returned by the subroutine), provided that the following are true:

- You specified that error 2015 is recoverable.
- You specified an input value for \( \text{naux1} \) that is at least \((4d+11)\) (but insufficient for the problem).
- There were no other errors.

Otherwise, it remains unchanged.

Returned as: an integer.

**naux2**

contains the minimum value required for successful processing (as returned by the subroutine), provided that the following are true:

- You specified that error 2015 is recoverable.
- You specified an input value for \( \text{naux2} \) that is greater than or equal to zero (but insufficient for the problem).
- There were no other errors.

Otherwise, it remains unchanged.

Returned as: an integer.

**Notes**

1. \( \text{aux1} \) should **not** be used by the calling program between calls to this subroutine with \( \text{init} > 0 \) and \( \text{init} = 0 \). However, it can be reused after intervening calls to this subroutine with different arguments.

2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for \( \text{init} > 0 \) and \( \text{init} = 0 \).

3. If \( \text{incx}_1 = 1 \) and \( \text{incy}_1 = 1 \), then:
   - \( \text{incy}_i \) must be even for \( i = 2,\ldots,d \)
   - \( \min(\text{incm}, \text{incy}_2,\ldots,\text{incy}_d) \geq 2(n_1/2+1) \)

4. For optimal performance, the preferred value for \( \text{incx}_1 \) and \( \text{incy}_1 \) is 1.

   If you specify the same array for \( X \) and \( Y \), then:
   - \( \text{incy}_i \) must equal \( 2(\text{incx}_i) \), for \( i = 2,\ldots,d \)
   - \( \text{incm} \) must equal to \( 2(\text{incm}_i) \) if \( m > 1 \)

   In this case, output overwrites input. If you specify different arrays for \( X \) and \( Y \), they must have no common elements; otherwise, results are unpredictable. See “Concepts” on page 75.

5. You have the option of having the minimum required value for \( \text{naux1} \) and \( \text{naux2} \) dynamically returned to your program. For details, see “Using Auxiliary Storage in ESSL” on page 51.
Formulas

Processor-Independent Formulas for SCRFTD for NAUX1 and NAUX2:

NAUX1 Formulas

If \( \max(n_1,n_2,...,n_d) \leq 2048 \), \( naux1 = 60000 \).
If \( \max(n_1,n_2,...,n_d) > 2048 \), \( naux1 = 60000 + 14.12(n_1+...+n_d) \).

NAUX2 Formulas

If \( \max(n_1,n_2,...,n_d) < 252 \), \( naux2 = 20000 \).
If \( \max(n_1,n_2,...,n_d) \geq 252 \), \( naux2 = 20000 + (2r + 256)(s + 8.56) \).

where:

\( r = \max(n_1,n_2,...,n_d) \) and
\( s = \min(64,r) \)

Processor-Independent Formulas for DCRFTD for NAUX1 and NAUX2:

NAUX1 Formulas

If \( \max(n_1,n_2,...,n_d) \leq 1024 \), \( naux1 = 60000 \).
If \( \max(n_1,n_2,...,n_d) > 1024 \), \( naux1 = 60000 + 28.24(n_1+...+n_d) \).

NAUX2 Formulas

If \( \max(n_1,n_2,...,n_d) < 252 \), \( naux2 = 20000 \).
If \( \max(n_1,n_2,...,n_d) \geq 252 \), \( naux2 = 20000 + (2r + 256)(s + 17.12) \).

where:

\( r = \max(n_1,n_2,...,n_d) \) and
\( s = \min(64,r) \)

Function

The set of \( m \) \( d \)-dimensional real discrete Fourier transforms of complex conjugate even data in array \( x \) with results going into array \( y \) is expressed as follows:

\[
Y_{k_1,k_2,...,kd} = scale \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_d=0}^{n_d-1} x_{j_1,j_2,...,j_d} W_{n_1}^{(j_1n_1+1)(k_1)} W_{n_2}^{(j_2n_2+1)(k_2)} \cdots W_{n_d}^{(j_dn_d+1)(kd)}
\]

for:

\( k_1 = 0,...,n_1-1 \)
\( k_2 = 0,...,n_2-1 \)
..
..
\( kd = 0,...,n_d-1 \)
\( i = 0,...,m-1 \)

where:
\[ W_{nl} = e^{-2\pi (\sqrt{-1})/n_l} \]

for:

\[ l = 1, \ldots, d \]

and where:

\[ x_{j_1, j_2, \ldots, j_d} \]

are elements of the \( d \)-dimensional sequences in array \( X \).

\[ y_{k_1, k_2, \ldots, k_d} \]

are elements of the \( d \)-dimensional sequences in array \( Y \).

For \( scale = 1.0 \) and \( isign_1 = isign_2 = \ldots = isign_d = 1 \), you obtain the discrete Fourier transform (DFT), a function of frequency. The inverse Fourier transform is obtained with \( scale = 1.0/n_1, n_2, \ldots, n_d \) and \( isign_1 = isign_2 = \ldots = isign_d = -1 \). See references [5 on page 1363], [7 on page 1363], [12 on page 1364], and [31 on page 1365].

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.

If \( n_i = 0 \) for any \( i \) from 1 to \( d \) or if \( m = 0 \); no initialization or computation is performed.

**Error conditions**

**Resource Errors**

Error 2015 is unrecoverable, unable to allocate work area, and internal deallocation error.

**Computational Errors**

None

**Input-Argument Errors**

1. \( init < 0 \) or \( init > 2 \)
2. \( d < 1 \) or \( d > 3 \)
3. \( incx_i \leq 0 \) (\( i = 1, \ldots, d \))
4. \( incnx \leq 0 \)
5. \( incy_i \leq 0 \) (\( i = 1, \ldots, d \))
6. \( incmy \leq 0 \)
7. \( n_i < 0 \) or \( n_i > 1073479680 \) (\( i = 1, \ldots, d \))
8. \( m \leq 0 \)
9. \( isign_i = 0 \) (\( i = 1, \ldots, d \))
10. \( scale = 0.0 \)
11. \( naux1 \leq 4d+11 \)
12. \( naux1 \) is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
13. 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.

14. The subroutine has not been initialized with the present arguments.

**Examples**

**Example 1**

This example shows how to compute a single one-dimensional transform.

The arrays are declared as follows:

```plaintext
COMPLEX*8 X(0:6)
REAL*8 AUX1(100), AUX2
REAL*4 Y(0:11)
```

First, initialize \( AUX1 \) using the calling sequence shown below with \( INIT \neq 0 \). Then use the same calling sequence with \( INIT = 0 \) to do the calculation.

**Note:** Because \( NAUX2 = 0 \), this subroutine dynamically allocates the \( AUX2 \) working storage.

**Call Statement and Input:**

```plaintext
CALL SCRFTD( INIT, 1 X, INCX, 7, Y, INCY, 12, N, 1, ISIGN, 1.0, AUX1, 100, AUX2, 0 )
```

**Output:**

\( X \) contains the following sequence:

- \((1.0, 0.0)\)
- \((0.0, 0.0)\)
- \((0.0, 0.0)\)
- \((0.0, 0.0)\)
- \((0.0, 0.0)\)
- \((0.0, 0.0)\)
- \((0.0, 0.0)\)

\( Y \) contains the following sequence:

- 1.0000
- 1.0000
- 1.0000
- 1.0000
- 1.0000
- 1.0000
- 1.0000
- 1.0000
- 1.0000
- 1.0000
- 1.0000
Example 2

This example shows how to compute a 3-dimensional Fourier transform. This example requires additional storage for array Y.

The arrays are declared as follows:

```plaintext
COMPLEX*8    X(4,3,2)
REAL*4       Y(9,3,2)
REAL*8       AUX1(5000, AUX2)
```

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:** Because NAUX2 = 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```plaintext
CALL SCRFTD( INIT, 3 X, 0, INCMX, Y, INCY, 0, N, 1, ISIGN, 1.0, AUX1, 5000, AUX2, 0)
```

- INIT = 1 (for initialization)
- INIT = 0 (for computation)
- INCMX is an array of length d.
- INCMX(1) = 1
- INCMX(2) = 4
- INCMX(3) = 12
- INCY is an array of length d
- INCY(1) = 1
- INCY(2) = 9
- INCY(3) = 27
- N is an array of length d
- N(1) = 7
- N(2) = 3
- N(3) = 2
- ISIGN is an array of length d
- ISIGN(1) = 1
- ISIGN(2) = 1
- ISIGN(3) = 1

X has (1.0,0.0) in location X(1,1,1) and (0.0,0.0) in all other locations.

Output:

- Y(i,j,k) = 1.0 for i = 1,...,7; j = 1,...,3; k = 1,2
- Y(i,j,k) is unchanged for i = 8,9; j = 1,...,3; k = 1,2
SCFT and DCFT (Complex Fourier Transform)

Purpose

These subroutines compute a set of *m* complex discrete *n*-point Fourier transforms of complex data.

**Table 203. Data Types**

<table>
<thead>
<tr>
<th>X, Y</th>
<th>scale</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>SCFT</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>DCFT</td>
</tr>
</tbody>
</table>

**Note:**

1. Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

**Syntax**

**Fortran**

```call SCFT | DCFT (init, x, inc1x, inc2x, y, inc1y, inc2y, n, m, isign, scale, aux1, aux1, aux2, aux2)```

**C and C++**

```scft | dcft (init, x, inc1x, inc2x, y, inc1y, inc2y, n, m, isign, scale, aux1, aux1, aux2, aux2);```

**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: an 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 203](#).

*inc1x* is the stride between the elements within each sequence in array *X*.

*Specified as: an 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: an integer; inc2x > 0.

*y* See [On Return](#)

*inc1y* is the stride between the elements within each sequence in array *Y*.
Specified as: an integer; inc1y > 0.

\textit{inc2y} is the stride between the first elements of each sequence in array \textit{Y}. (If \(m = 1\), this argument is ignored.) Specified as: an integer; inc2y > 0.

\textit{n} is the length of each sequence to be transformed.

Specified as: an integer; \(n \leq 37748736\) and must be one of the values listed in “Acceptable Lengths for the Transforms” on page 1028. 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 58.

\textit{m} is the number of sequences to be transformed.

Specified as: an integer; \(m > 0\).

\textit{isign} controls the direction of the transform, determining the sign \(I_{sign}\) of the exponent of \(W_n\), where:

If \(isign = \) positive value, \(I_{sign} = +\) (transforming time to frequency).

If \(isign = \) negative value, \(I_{sign} = -\) (transforming frequency to time).

Specified as: an integer; \(isign > 0\) or \(isign < 0\).

\textit{scale} is the scaling constant \(scale\). See “Function” on page 1063 for its usage.

Specified as: a number of the data type indicated in Table 203 on page 1060 where \(scale > 0.0\) or \(scale < 0.0\)

\textit{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.

\textit{naux1} is the number of doublewords in the working storage specified in \(aux1\).

Specified as: an integer; \(naux1 > 7\) (32-bit integer arguments) or 13 (64-bit integer arguments) and \(naux1 \geq (\) minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For values between 7 (32-bit integer arguments) or 13 (64-bit integer arguments) 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 51.

\textit{naux2} is the number of doublewords in the working storage specified in \(aux2\).

Specified as: an 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.

Chapter 12. Fourier Transforms, Convolutions and Correlations, and Related Computations 1061
Otherwise, \( n_{aux2} \geq \) (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 51.

**On Return**

- \( y \) has the following meaning, where:
  - If \( \text{init} \neq 0 \), this argument is not used, and its contents remain unchanged.
  - If \( \text{init} = 0 \), 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 \( 1+(n-1)\text{inc1y}+(m-1)\text{inc2y} \), containing numbers of the data type indicated in Table 203 on page 1060.

- \( \text{aux1} \) is the working storage for this subroutine, where:
  - If \( \text{init} \neq 0 \), it contains information ready to be passed in a subsequent invocation of this subroutine.
  - If \( \text{init} = 0 \), its contents are unchanged.

  Returned as: the contents are not relevant.

**Notes**

1. \( \text{aux1} \) should not be used by the calling program between calls to this subroutine with \( \text{init} \neq 0 \) and \( \text{init} = 0 \). However, it can be reused after intervening calls to this subroutine with different arguments.

2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for \( \text{init} \neq 0 \) and \( \text{init} = 0 \).

3. For optimal performance, the preferred value for \( \text{inc1x} \) and \( \text{inc1y} \) is 1. This implies that the sequences are stored with stride 1. The preferred value for \( \text{inc2x} \) and \( \text{inc2y} \) 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, \( \text{inc2x} \) (or \( \text{inc2y} \)) = 1 and \( \text{inc1x} \) (or \( \text{inc1y} \)) 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 \( \text{inc1x} \) and \( \text{inc1y} \) for _CFT, you should use “STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)” on page 1307. 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 \( \text{inc1x} \) and \( \text{inc1y} \) must be equal, and \( \text{inc2x} \) and \( \text{inc2y} \) must be equal. In this case, output overwrites input. If \( m = 1 \), the \( \text{inc2x} \) and \( \text{inc2y} \) 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 75.

**Formulas**

Processor-Independent Formulas for SCFT for NAUX1 and NAUX2:

**NAUX1 Formulas**

For 32-bit integer arguments:

...
If \( n \leq 8192 \), use \( \text{naux1} = 20000 \).
If \( n > 8192 \), use \( \text{naux1} = 20000 + 1.14n \).

**For 64-bit integer arguments:**

If \( n \leq 8192 \), use \( \text{naux1} = 30000 \).
If \( n > 8192 \), use \( \text{naux1} = 30000 + 1.14n \).

**NAUX2 Formulas**

If \( n \leq 8192 \), use \( \text{naux2} = 20000 \).
If \( n > 8192 \), use \( \text{naux2} = 20000 + 1.14n \).

For the transposed case, where \( \text{inc2x} = 1 \) or \( \text{inc2y} = 1 \), 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**

**For 32-bit integer arguments:**

If \( n \leq 2048 \), use \( \text{naux1} = 20000 \).
If \( n > 2048 \), use \( \text{naux1} = 20000 + 2.28n \).

**For 64-bit integer arguments:**

If \( n \leq 2048 \), use \( \text{naux1} = 30000 \).
If \( n > 2048 \), use \( \text{naux1} = 30000 + 2.28n \).

**NAUX2 Formulas**

If \( n \leq 2048 \), use \( \text{naux2} = 20000 \).
If \( n > 2048 \), use \( \text{naux2} = 20000 + 2.28n \).

For the transposed case, where \( \text{inc2x} = 1 \) or \( \text{inc2y} = 1 \), and where \( n \geq 252 \), add the following to the above storage requirements:

\[(2n+256)(\min(64, m))\]

**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:

\[y_{ki} = scale \sum_{j=0}^{n-1} x_{ji} W_n^{(\text{sign})jk}\]

for:

\[k = 0, 1, ..., n-1\]
\[i = 1, 2, ..., m\]

where:
\[ W_n = e^{-2\pi i / n} \]

and where:

- \( x_{ji} \) are elements of the sequences in array \( X \).
- \( y_{ki} \) are elements of the sequences in array \( Y \).
- \( \text{isign} \) is + or - (determined by argument \( \text{isign} \)).
- \( \text{scale} \) is a scalar value.

For \( \text{scale} = 1.0 \) and \( \text{isign} \) being positive, you obtain the discrete Fourier transform, a function of frequency. The inverse Fourier transform is obtained with \( \text{scale} = 1.0/n \) and \( \text{isign} \) being negative. See references [1 on page 1363], [3 on page 1363], [4 on page 1363], [26 on page 1364], and [27 on page 1364].

Two invocations of this subroutine are necessary:

1. With \( \text{init} \neq 0 \), the subroutine tests and initializes arguments of the program, setting up the aux1 working storage.
2. With \( \text{init} = 0 \), 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. \( \text{inc1x, inc2x, inc1y, or inc2y} \leq 0 \)
3. \( m \leq 0 \)
4. \( \text{isign} = 0 \)
5. \( \text{scale} = 0.0 \)
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. \( naux1 \leq 7 \)
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 \( naux2 \neq 0 \), and \( naux2 \) is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

**Examples**

**Example 1**

This example shows an input array \( X \) with a set of four short-precision complex sequences:
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:

\[
\begin{align*}
\text{COMPLEX*8} & \quad X(0:1023), Y(0:1023) \\
\text{REAL*8} & \quad \text{AUX1(1693), AUX2(1)}
\end{align*}
\]

First, initialize \( \text{AUX1} \) using the calling sequence shown below with \( \text{INIT} \neq 0 \). Then use the same calling sequence with \( \text{INIT} = 0 \) to do the calculation.

**Note:** Because \( \text{NAUX2} = 0 \), this subroutine dynamically allocates the \( \text{AUX2} \) working storage.

**Call Statement and Input:**

\[
\begin{align*}
\text{INIT} & \quad \text{X INC1X INC2X Y INC1Y INC2Y N M ISIGN SCALE AUX1 NAUX1 AUX2 NAUX2} \\
\text{CALL SCFT(INIT, X, 1, 8, Y, 1, 8, 8, 4, 1, SCALE, AUX1, 1693, AUX2, 0)}
\end{align*}
\]

\[
\begin{align*}
\text{INIT} & = 1 \text{ (for initialization)} \\
\text{INIT} & = 0 \text{ (for computation)} \\
\text{SCALE} & = 1.0 \\
\end{align*}
\]

\( X \) contains the following four sequences:

\[
\begin{align*}
&(1.0000, 0.0000) \quad (1.0000, 0.0000) \quad (1.0000, 0.0000) \quad (1.0000, 0.0000) \\
&(1.0000, 0.0000) \quad (0.7071, 0.7071) \quad (0.0000, 1.0000) \quad (-0.7071, 0.7071) \\
&(1.0000, 0.0000) \quad (0.0000, 1.0000) \quad (-1.0000, 0.0000) \quad (0.0000, -1.0000) \\
&(1.0000, 0.0000) \quad (-0.7071, 0.7071) \quad (0.0000, -1.0000) \quad (0.7071, 0.7071) \\
&(1.0000, 0.0000) \quad (-1.0000, 0.0000) \quad (1.0000, 0.0000) \quad (-1.0000, 0.0000) \\
&(1.0000, 0.0000) \quad (-0.7071, -0.7071) \quad (0.0000, 1.0000) \quad (0.7071, -0.7071) \\
&(1.0000, 0.0000) \quad (0.0000, -1.0000) \quad (-1.0000, 0.0000) \quad (0.0000, 1.0000) \\
&(1.0000, 0.0000) \quad (0.7071, -0.7071) \quad (0.0000, -1.0000) \quad (-0.7071, -0.7071)
\end{align*}
\]

Output:

\( Y \) contains the following four sequences:

\[
\begin{align*}
&(8.0000, 0.0000) \quad (0.0000, 0.0000) \quad (0.0000, 0.0000) \quad (0.0000, 0.0000) \\
&(0.0000, 0.0000) \quad (8.0000, 0.0000) \quad (0.0000, 0.0000) \quad (0.0000, 0.0000) \\
&(0.0000, 0.0000) \quad (0.0000, 0.0000) \quad (8.0000, 0.0000) \quad (0.0000, 0.0000) \\
&(0.0000, 0.0000) \quad (0.0000, 0.0000) \quad (0.0000, 0.0000) \quad (8.0000, 0.0000) \\
&(0.0000, 0.0000) \quad (0.0000, 0.0000) \quad (0.0000, 0.0000) \quad (0.0000, 0.0000) \\
&(0.0000, 0.0000) \quad (0.0000, 0.0000) \quad (0.0000, 0.0000) \quad (0.0000, 0.0000) \\
&(0.0000, 0.0000) \quad (0.0000, 0.0000) \quad (0.0000, 0.0000) \quad (0.0000, 0.0000)
\end{align*}
\]

**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 \( \text{isign} \), giving as output the four sequences listed in the input for Example 1. First, initialize \( \text{AUX1} \) using the calling sequence shown below with \( \text{INIT} \neq 0 \). Then use the same calling sequence with \( \text{INIT} = 0 \) to do the calculation.

**Note:** Because \( \text{NAUX2} = 0 \), this subroutine dynamically allocates the \( \text{AUX2} \) working storage.

**Call Statement and Input:**
Example 3

This example shows an input array X with a set of four short-precision complex sequences 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:

```cpp
COMPLEX*8 X (4,0:11), Y(4,0:11)
REAL*8 AUX1(10000), AUX2(1)
```

First, initialize AUX1 using the calling sequence shown below with \( INIT \neq 0 \).
Then use the same calling sequence with \( INIT = 0 \) to do the calculation.

**Note:** Because \( NAUX2 = 0 \), this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```cpp
CALL SCFT(INIT, X, 4, 1, Y, 4, 12, 4, 1, SCALE, AUX1, 10000, AUX2, 0)
```

\( INIT = 1 \) (for initialization)
\( INIT = 0 \) (for computation)
\( SCALE = 1.0 \)

\( X \) contains the following four sequences:

\[
(1.0000, 0.0000) \quad (1.0000, 0.0000) \quad (1.0000, 0.0000) \quad (1.0000, 0.0000)
(1.0000, 0.0000) \quad (0.8660, 0.5000) \quad (0.5000, 0.8660) \quad (0.0000, 1.0000)
(1.0000, 0.0000) \quad (0.5000, 0.8660) \quad (-0.5000, 0.8660) \quad (-1.0000, 0.0000)
(1.0000, 0.0000) \quad (0.0000, 1.0000) \quad (-1.0000, 0.0000) \quad (0.0000, -1.0000)
(1.0000, 0.0000) \quad (-0.5000, 0.8660) \quad (-0.5000, -0.8660) \quad (1.0000, 0.0000)
(1.0000, 0.0000) \quad (-1.0000, 0.0000) \quad (1.0000, 0.0000) \quad (-1.0000, 0.0000)
(1.0000, 0.0000) \quad (-0.8660, -0.5000) \quad (0.5000, 0.8660) \quad (0.0000, -1.0000)
(1.0000, 0.0000) \quad (-1.0000, 0.0000) \quad (1.0000, 0.0000) \quad (-1.0000, 0.0000)
(1.0000, 0.0000) \quad (0.5000, -0.8660) \quad (-0.5000, -0.8660) \quad (-1.0000, 0.0000)
(1.0000, 0.0000) \quad (0.8660, -0.5000) \quad (0.5000, -0.8660) \quad (0.0000, -1.0000)
\]

Output:

\( Y \) contains the following four sequences:
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 $\text{isign}$, giving as output the four sequences listed in the input for Example 3. First, initialize $\text{AUX1}$ using the calling sequence shown below with $\text{INIT} \neq 0$. Then use the same calling sequence with $\text{INIT} = 0$ to do the calculation.

**Note:** Because $\text{NAUX2} = 0$, this subroutine dynamically allocates the $\text{AUX2}$ working storage.

**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, 0)
```

- $\text{INIT} = 1$ (for initialization)
- $\text{INIT} = 0$ (for computation)
- $\text{SCALE} = 1.0/12.0$
- $X$ = (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 $\text{isign} = 1$ and $\text{scale} = 1.0$. The arrays are declared as follows:

```
COMPLEX*16 X(0:7),Y(0:7)
REAL*8 AUX1(26),AUX2(1)
```

The input in $X$ is an impulse at zero, and the output in $Y$ is constant for all frequencies. First, initialize $\text{AUX1}$ using the calling sequence shown below with $\text{INIT} \neq 0$. Then use the same calling sequence with $\text{INIT} = 0$ to do the calculation.

**Note:** Because $\text{NAUX2} = 0$, this subroutine dynamically allocates the $\text{AUX2}$ working storage.

**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, 0)
```
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)

Purpose

These subroutines compute a set of \( m \) complex discrete \( n \)-point Fourier transforms of real data.

<table>
<thead>
<tr>
<th>Table 204. Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X, \text{scale} )</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
</tbody>
</table>

Note:
1. Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL SRCFT (init, x, inc2x, y, inc2y, n, m, isign, scale, aux1, aux2, aux3, aux4)</td>
</tr>
<tr>
<td>CALL DRCFT (init, x, inc2x, y, inc2y, n, m, isign, scale, aux1, aux2, aux3, aux4)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C and C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcft (init, x, inc2x, y, inc2y, n, m, isign, scale, aux1, aux2, aux3, aux4);</td>
</tr>
<tr>
<td>dzrct (init, x, inc2x, y, inc2y, n, m, isign, scale, aux1, aux2, aux3, aux4);</td>
</tr>
</tbody>
</table>

On Entry

\( \text{init} \)

is a flag, where:

If \( \text{init} \neq 0 \), trigonometric functions and other parameters, depending on arguments other than \( x \), are computed and saved in \( \text{aux1} \). The contents of \( x \) and \( y \) are not used or changed.

If \( \text{init} = 0 \), the discrete Fourier transforms of the given sequences are computed. The only arguments that may change after initialization are \( x, y \), and \( \text{aux2} \). All scalar arguments must be the same as when the subroutine was called for initialization with \( \text{init} \neq 0 \).

Specified as: an 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 \( n+(m-1)\text{inc2x} \), containing numbers of the data type indicated in [Table 204](#). See “Notes” on [page 1071](#) for more details. (It can be declared as \( X(\text{inc2x},m) \).)

\( \text{inc2x} \)

is the stride between the first elements of the sequences in array \( X \). (If \( m = 1 \), this argument is ignored.) Specified as: an integer; \( \text{inc2x} \geq n \).

\( y \)

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: an integer; \( \text{inc2y} \geq (n/2)+1 \).

\( n \) is the length of each sequence to be transformed.
Specified as: an integer; \( n \leq 37748736 \) and must be one of the values listed in
[Acceptable Lengths for the Transforms” on page 1028]. 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 58.

\( m \) is the number of sequences to be transformed.
Specified as: an integer; \( m > 0 \).

isign
controls the direction of the transform, determining the sign Isign of the
exponent of \( W_n \), where:
If \( \text{isign} \) = positive value, \( \text{isign} = + \) (transforming time to frequency).
If \( \text{isign} \) = negative value, \( \text{isign} = - \) (transforming frequency to time).
Specified as: an integer; \( \text{isign} > 0 \) or \( \text{isign} < 0 \).

scale
is the scaling constant scale. See “Function” on page 1072 for its usage.
Specified as: a number of the data type indicated in Table 204 on page 1069
where \( \text{scale} > 0.0 \) or \( \text{scale} < 0.0 \).

aux1
is the working storage for this subroutine, where:
If \( \text{init} \neq 0 \), the working storage is computed.
If \( \text{init} = 0 \), the working storage is used in the computation of the Fourier
transforms.
Specified as: an area of storage, containing \( \text{naux1} \) long-precision real numbers.

naux1
is the number of doublewords in the working storage specified in \( \text{aux1} \).
Specified as: an integer; \( \text{naux1} > 14 \) (32-bit integer arguments) or 27 (64-bit integer arguments) and \( \text{naux1} \geq \) (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For values between 14 (32-bit integer arguments) or 27 (64-bit integer arguments) 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 51.

aux2
has the following meaning:
If \( \text{naux2} = 0 \) and error 2015 is unrecoverable, \( \text{aux2} \) 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 \( \text{naux2} \) long-precision real numbers.
On output, the contents are overwritten.

naux2
is the number of doublewords in the working storage specified in \( \text{aux2} \).
Specified as: an integer, where:

If \( \text{naux2} = 0 \) 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, \( \text{naux2} \geq (\text{minimum value required for successful processing}) \). 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 51.

**aux3**

This argument is provided for migration purposes only and is ignored.

Specified as: an area of storage, containing \( \text{naux3} \) long-precision real numbers.

**naux3**

This argument is provided for migration purposes only and is ignored.

Specified as: an integer.

**On Return**

\( y \) has the following meaning, where:

If \( \text{init} \neq 0 \), this argument is not used, and its contents remain unchanged.

If \( \text{init} = 0 \), 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, \( y_{ki}, k = 0, 1, \ldots, n/2; i = 1, 2, \ldots, m \).

Returned as: an array of (at least) length \( n/2 + 1 + (m-1)\text{inc2y} \), containing numbers of the data type indicated in Table 204 on page 1069. This array can be declared as \( Y(\text{inc2y}, m) \).

**aux1**

Is the working storage for this subroutine, where:

If \( \text{init} \neq 0 \), it contains information ready to be passed in a subsequent invocation of this subroutine.

If \( \text{init} = 0 \), its contents are unchanged.

Returned as: the contents are not relevant.

**Notes**

1. \( \text{aux1} \) should not be used by the calling program between calls to this subroutine with \( \text{init} \neq 0 \) and \( \text{init} = 0 \). However, it can be reused after intervening calls to this subroutine with different arguments.

2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for \( \text{init} \neq 0 \) and \( \text{init} = 0 \).

3. 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, \( \text{inc1x} \) and \( \text{inc1y} \) values are not a part of the argument list. For optimal performance, the \( \text{inc2x} \) and \( \text{inc2y} \) values should be close to their respective minimum values, which are given below:

\[
\begin{align*}
\min(\text{inc2x}) &= n \\
\min(\text{inc2y}) &= n/2+1
\end{align*}
\]
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 75.

Formulas

Processor-Independent Formulas for SRCFT for NAUX1 and NAUX2:

**NAUX1 Formulas**

For 32-bit integer arguments:

- If \( n \leq 16384 \), use \( naux1 = 25000 \).
- If \( n > 16384 \), use \( naux1 = 20000 + 0.82n \).

For 64-bit integer arguments:

- If \( n \leq 16384 \), use \( naux1 = 35000 \).
- If \( n > 16384 \), use \( naux1 = 30000 + 0.82n \).

**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**

For 32-bit integer arguments:

- If \( n \leq 4096 \), use \( naux1 = 22000 \).
- If \( n > 4096 \), use \( naux1 = 20000 + 1.64n \).

For 64-bit integer arguments:

- If \( n \leq 4096 \), use \( naux1 = 32000 \).
- If \( n > 4096 \), use \( naux1 = 30000 + 1.64n \).

**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:

\[
y_{ki} = scale \sum_{j=0}^{n-1} x_{ji} W_n^{[1-\text{sign}])jk}
\]

for:

\[
k = 0, 1, ..., n-1
\]

\[
i = 1, 2, ..., m
\]
where:

\[ W_n = e^{-2\pi j n / n} \]

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 on page 1363], [4 on page 1363], [26 on page 1364], and [27 on page 1364].

Two invocations of this subroutine are necessary:
1. With \( init \neq 0 \), 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 \( naux2=0 \), and \( naux2 \) is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

**Examples**

**Example 1**
This example shows an input array \( X \) with a set of \( m \) cosine sequences 
\[ \cos(2\pi jk/n), \quad j = 0, 1, \ldots, 15 \]  
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:

```fortran
REAL*4   X(0:65535)
COMPLEX*8 Y(0:32768)
REAL*8   AUX1(41928), AUX2(1), AUX3(1)
```

First, initialize \( AUX1 \) using the calling sequence shown below with \( INIT \neq 0 \). Then use the same calling sequence with \( INIT = 0 \) to do the calculation.

**Note:** Because \( NAUX2 = 0 \), this subroutine dynamically allocates the \( AUX2 \) working storage.

**Call Statement and Input:**

```fortran
CALL SRCFT(INIT, X, 16, Y, 9, 16, 4, 1, SCALE, AUX1, 41928, AUX2, 0, AUX3, 0)
```

**INIT**  
- 1 (for initialization)
- 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:

\[
\begin{align*}
(1.0000, & 0.0000) (0.0000, 0.0000) (0.0000, 0.0000) (0.0000, 0.0000) \\
(0.8000, & 0.0000) (0.5000, 0.0000) (0.0000, 0.0000) (0.0000, 0.0000) \\
(0.0000, & 0.0000) (0.0000, 0.0000) (0.5000, 0.0000) (0.0000, 0.0000) \\
(0.0000, & 0.0000) (0.0000, 0.0000) (0.0000, 0.0000) (0.5000, 0.0000) \\
(0.0000, & 0.0000) (0.0000, 0.0000) (0.0000, 0.0000) (0.0000, 0.0000) \\
(0.0000, & 0.0000) (0.0000, 0.0000) (0.0000, 0.0000) (0.0000, 0.0000) \\
(0.0000, & 0.0000) (0.0000, 0.0000) (0.0000, 0.0000) (0.0000, 0.0000)
\end{align*}
\]

**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 \( \cos(2\pi jk/n) \), \( j = 0, 1, \ldots, 15 \) 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.

Note: Because \( \text{NAUX2} = 0 \), this subroutine dynamically allocates the \( \text{AUX2} \)
working storage.

Call Statement and Input:

\[
\text{CALL SRCFT( 0 , X, 16 , Y, 9 , 16, 4 , 1, SCALE, AUX1, 41928 , AUX2, 0 , AUX3, 0 )}
\]

\( \text{SCALE} = 1.0/16 \)

\( X \) contains the following four sequences:

\[
\begin{array}{cccc}
1.0000 & 1.0000 & 1.0000 & 1.0000 \\
-1.0000 & -0.9239 & -0.7071 & -0.3827 \\
1.0000 & 0.7071 & 0.0000 & -0.7071 \\
-1.0000 & -0.3827 & 0.7071 & 0.9239 \\
1.0000 & 0.0000 & -1.0000 & 0.0000 \\
-1.0000 & 0.3827 & 0.7071 & -0.9239 \\
1.0000 & -0.7071 & 0.0000 & 0.7071 \\
-1.0000 & 0.9239 & -0.7071 & 0.3827 \\
1.0000 & -1.0000 & 1.0000 & -1.0000 \\
-1.0000 & 0.9239 & -0.7071 & 0.3827 \\
1.0000 & -0.7071 & 0.0000 & 0.7071 \\
-1.0000 & 0.3827 & 0.7071 & -0.9239 \\
1.0000 & 0.0000 & -1.0000 & 0.0000 \\
-1.0000 & -0.3827 & 0.7071 & 0.9239 \\
1.0000 & 0.7071 & 0.0000 & -0.7071 \\
-1.0000 & -0.9239 & -0.7071 & -0.3827 \\
\end{array}
\]

Output:

\( Y \) contains the following four sequences:

\[
\begin{array}{cccc}
(0.0000, 0.0000) & (0.0000, 0.0000) & (0.0000, 0.0000) & (0.0000, 0.0000) \\
(0.0000, 0.0000) & (0.0000, 0.0000) & (0.0000, 0.0000) & (0.0000, 0.0000) \\
(0.0000, 0.0000) & (0.0000, 0.0000) & (0.0000, 0.0000) & (0.0000, 0.0000) \\
(0.0000, 0.0000) & (0.0000, 0.0000) & (0.0000, 0.0000) & (0.0000, 0.0000) \\
(0.0000, 0.0000) & (0.0000, 0.0000) & (0.0000, 0.0000) & (0.0000, 0.0000) \\
(0.0000, 0.0000) & (0.0000, 0.0000) & (0.0000, 0.0000) & (0.0000, 0.0000) \\
(1.0000, 0.0000) & (0.0000, 0.0000) & (0.0000, 0.0000) & (0.0000, 0.0000) \\
(1.0000, 0.0000) & (0.0000, 0.0000) & (0.0000, 0.0000) & (0.0000, 0.0000) \\
\end{array}
\]

Example 3

This example uses the mixed-radix capability. The arrays are declared as
follows:

\[
\begin{array}{ll}
\text{REAL*8} & X(0:11) \\
\text{COMPLEX*16} & Y(0:6) \\
\text{REAL*8} & \text{AUX1(50), AUX2(1)} \\
\end{array}
\]

Arrays \( X \) and \( Y \) are made equivalent by the following statement, making them
occupy the same storage:

\[
\text{EQUIVALENCE (X, Y)}
\]

First, initialize \( \text{AUX1} \) using the calling sequence shown below with \( \text{INIT} \neq 0 \).
Then use the same calling sequence with \( \text{INIT} = 0 \) to do the calculation.

Note: Because \( \text{NAUX2} = 0 \), this subroutine dynamically allocates the \( \text{AUX2} \)
working storage.

Call Statement and Input:
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)

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)

Purpose

These subroutines compute a set of \( m \) real discrete \( n \)-point Fourier transforms of complex conjugate even data.

Table 205. Data Types

<table>
<thead>
<tr>
<th>( X )</th>
<th>( Y, \text{scale} )</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>SCRFT</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>DCRFT</td>
</tr>
</tbody>
</table>

Note:

1. Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C and C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL SCRFT ((\text{init, x, inc2x, y, inc2y, n, m, isign, scale, aux1, aux2, aux3, aux5)})</td>
<td><code>scrft (init, x, inc2x, y, inc2y, n, m, isign, scale, aux1, aux2, aux3, aux5);</code></td>
</tr>
<tr>
<td>CALL DCRFT ((\text{init, x, inc2x, y, inc2y, n, m, isign, scale, aux1, aux2, aux3, aux5)})</td>
<td><code>dcrft (init, x, inc2x, y, inc2y, n, m, isign, scale, aux1, aux2, aux3, aux5);</code></td>
</tr>
</tbody>
</table>

On Entry

\( \text{init} \)

is a flag, where:

If \( \text{init} \neq 0 \), trigonometric functions and other parameters, depending on arguments other than \( x \), are computed and saved in \( \text{aux1} \). The contents of \( x \) and \( y \) are not used or changed.

If \( \text{init} = 0 \), the discrete Fourier transforms of the given sequences are computed. The only arguments that may change after initialization are \( x \), \( y \), and \( \text{aux2} \). All scalar arguments must be the same as when the subroutine was called for initialization with \( \text{init} \neq 0 \).

Specified as: an 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, \( x_{ij} = 0, 1, ..., n/2, i = 1, 2, ..., m \). The sequences are assumed to be stored with stride 1.

Specified as: an array of (at least) length \( n/2+1+(m-1)\text{inc2x} \), containing numbers of the data type indicated in Table 205. This array can be declared as \( X(\text{inc2x},m) \).

\( \text{inc2x} \)

is the stride between the first elements of the sequences in array \( X \). (If \( m = 1 \), this argument is ignored.) Specified as: an integer; \( \text{inc2x} \equiv (n/2)+1 \).

\( y \)

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: an integer; inc2y ≥ n.

n

is the length of each sequence to be transformed.

Specified as: an integer; n ≤ 37748736 and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 1028. 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 58.

m

is the number of sequences to be transformed.

Specified as: an 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: an integer; isign > 0 or isign < 0.

scale

is the scaling constant scale. See "Function" on page 1080 for its usage.

Specified as: a number of the data type indicated in Table 205 on page 1077 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: an integer; naux1 > 13 (32-bit integer arguments) or 25 (64-bit integer arguments) and naux1 ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For values between 13 (32-bit integer arguments) or 25 (64-bit integer arguments) 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 51.

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: an 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, \( naux2 \geq \) (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 51.

**\( aux3 \)**

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: an integer.

**On Return**

**\( y \)**

Has the following meaning, where:

If \( init \neq 0 \), 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 205 on page 1077. See "Notes" for more details. (It can be declared as \( Y(inc2y,m) \).)

**\( aux1 \)**

Is the working storage for this subroutine, where:

If \( init \neq 0 \), 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 \( init \neq 0 \) and \( init = 0 \). However, it can be reused after intervening calls to this subroutine with different arguments.

2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for \( init \neq 0 \) and \( init = 0 \).

3. 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:

\[
\begin{align*}
\min(inc2y) &= n \\
\min(inc2x) &= n/2+1
\end{align*}
\]

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 75.

Formulas

Processor-Independent Formulas for SCRFT for NAUX1 and NAUX2:

**NAUX1 Formulas**

**For 32-bit integer arguments:**

If \( n \leq 16384 \), use \( naux1 = 25000 \).
If \( n > 16384 \), use \( naux1 = 20000 + 0.82n \).

**For 64-bit integer arguments:**

If \( n \leq 16384 \), use \( naux1 = 35000 \).
If \( n > 16384 \), use \( naux1 = 30000 + 0.82n \).

**NAUX2 Formulas**

If \( n \leq 16384 \), use \( naux2 = 20000 \).
If \( n > 16384 \), use \( naux2 = 20000 + 0.57n \).

Processor-Independent Formulas for DCRFT for NAUX1 and NAUX2:

**NAUX1 Formulas**

**For 32-bit integer arguments:**

If \( n \leq 4096 \), use \( naux1 = 22000 \).
If \( n > 4096 \), use \( naux1 = 20000 + 1.64n \).

**For 64-bit integer arguments:**

If \( n \leq 4096 \), use \( naux1 = 32000 \).
If \( n > 4096 \), use \( naux1 = 30000 + 1.64n \).

**NAUX2 Formulas**

If \( n \leq 4096 \), use \( naux2 = 20000 \).
If \( n > 4096 \), use \( naux2 = 20000 + 1.14n \).

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:

\[
y_{ki} = scale \sum_{j=0}^{n-1} x_{ji} W^{(i\text{sign})jk}
\]

for:

\( k = 0, 1, \ldots, n-1 \)
\( i = 1, 2, \ldots, m \)
where:

\[ W_n = e^{-2\pi i/n} \]

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 on page 1363], [4 on page 1363], [26 on page 1364], and [27 on page 1364].

Two invocations of this subroutine are necessary:
1. With \( init \neq 0 \), 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. \( inc2x < n/2+1 \)
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. \( naux1 \leq 13 \)
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 \( naux2 \neq 0 \), and \( naux2 \) is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

**Examples**

**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(1), 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:**
1. X shows the \( n/2+1 = 7 \) elements used in the computation.
2. Because NAUX2 = 0, this subroutine dynamically allocates the AUX2 working storage.

**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, 0, 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)
```

**Example 2**

This example shows another transform computation with different data using the same initialized array AUX1 as in Example 1.

**Note:** Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

**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, 0, 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)
```

Output:
Example 3

This example shows how to compute many transforms simultaneously. The arrays are declared as follows:

```plaintext
COMPLEX*8    X(0:8,2)
REAL*8       AUX1(50), AUX2(1), 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.

**Note:** Because `NAUX2 = 0`, this subroutine dynamically allocates the `AUX2` working storage.

Call Statement and Input:

```plaintext
CALL SCRFT(INIT, X, 9, Y, 16, 16, 2, 1, SCALE, AUX1, 50, AUX2, 0, AUX3, 0)
```

**INIT** = 1 (for initialization)
**INIT** = 0 (for computation)
**SCALE** = 1.0

`X` contains the following two sequences:

- \((1.0, 0.0)\)
- \((1.0, 0.0)\)
- \((1.0, 0.0)\)
- \((1.0, 0.0)\)
- \((1.0, 0.0)\)
- \((1.0, 0.0)\)
- \((1.0, 0.0)\)
- \((1.0, 0.0)\)

Output:

`Y` contains the following two sequences:

- \((16.0, 1.0)\)
- \((0.0, -1.0)\)
- \((0.0, 1.0)\)
- \((0.0, -1.0)\)
- \((0.0, 1.0)\)
- \((0.0, -1.0)\)
- \((0.0, 1.0)\)
- \((0.0, -1.0)\)

Example 4

This example shows the same array being used for input and output. The arrays are declared as follows:

```plaintext
COMPLEX*16   X(0:8,2)
REAL*8       AUX1(50), AUX2(1)
REAL*8       Y(0:17,2)
```
Arrays $X$ and $Y$ are made equivalent by the following statement, making them occupy the same storage:

\[
\text{EQUIVALENCE (X,Y)}
\]

This requires $\text{INC2Y} = 2(\text{INC2X})$. First, initialize $\text{AUX1}$ using the calling sequence shown below with $\text{INIT} \neq 0$. Then use the same calling sequence with $\text{INIT} = 0$ to do the calculation.

**Note:** Because $\text{NAUX2} = 0$, this subroutine dynamically allocates the $\text{AUX2}$ working storage.

**Call Statement and Input:**

\[
\begin{array}{cccccccccccc}
\text{INIT} & \text{X} & \text{INC2X} & \text{Y} & \text{INC2Y} & \text{N} & \text{M} & \text{ISIGN} & \text{SCALE} & \text{AUX1} & \text{AUX2} & \text{NAUX2} \\
\hline
\text{INIT} & 1 & \text{for initialization} \\
\text{INIT} & 0 & \text{for computation} \\
\text{SCALE} & 0.0625 \\
\end{array}
\]

$X$ 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)  
(-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  
1.0 0.0  
0.0 0.0  
0.0 0.0  
0.0 0.0  
0.0 0.0  
0.0 0.0
**SCOSF and DCOSF (Cosine Transform)**

**Purpose**

These subroutines compute a set of \( m \) real even discrete \( n \)-point Fourier transforms of cosine sequences of real even data.

**Table 206. Data Types**

<table>
<thead>
<tr>
<th>( X, Y, \text{scale} )</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SCOSF</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DCOSF</td>
</tr>
</tbody>
</table>

**Note:**

1. Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

**Syntax**

**Fortran**

```
CALL SCOSF | DCOSF (init, x, inc1x, inc2x, y, inc1y, inc2y, n, m, scale, aux1, aux1, aux2, aux2)
```

**C and C++**

```
scosf | dcosf (init, x, inc1x, inc2x, y, inc1y, inc2y, n, m, scale, aux1, aux1, aux2, aux2);
```

**On Entry**

- \( init \)
  
  is a flag, where:
  
  - If \( init \neq 0 \), 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 \( init \neq 0 \).
  
  Specified as: an 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 206.

- \( inc1x \)
  
  is the stride between the elements within each sequence in array \( X \).

  Specified as: an 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: an integer; \( inc2x > 0 \).

- \( y \)
  
  See On Return

- \( inc1y \)
  
  is the stride between the elements within each sequence in array \( Y \).
Specified as: an integer; \(\text{inc1y} > 0\).

\(\text{inc2y}\)

is the stride between the first elements of the sequences in array \(Y\). (If \(m = 1\), this argument is ignored.) Specified as: an integer; \(\text{inc2y} > 0\).

\(n\)

is the transform length. However, due to symmetry, only the first \(n/2+1\) values are given in the input and output.

Specified as: an integer; \(n \leq 37748736\) and must be one of the values listed in
“Acceptable Lengths for the Transforms” on page 1028. 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 58.

\(m\)

is the number of sequences to be transformed.

Specified as: an integer; \(m > 0\).

\(\text{scale}\)

is the scaling constant \(\text{scale}\). See “Function” on page 1088 for its usage.

Specified as: a number of the data type indicated in Table 206 on page 1085, where \(\text{scale} > 0.0\) or \(\text{scale} < 0.0\).

\(\text{aux1}\)

is the working storage for this subroutine, where:

If \(\text{init} \neq 0\), the working storage is computed.

If \(\text{init} = 0\), the working storage is used in the computation of the Fourier transforms.

Specified as: an area of storage, containing \(\text{naux1}\) long-precision real numbers.

\(\text{naux1}\)

is the number of doublewords in the working storage specified in \(\text{aux1}\).

Specified as: an integer; \(\text{naux1} \geq (\text{minimum value required for successful processing})\). 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 51.

\(\text{aux2}\)

has the following meaning:

If \(\text{naux2} = 0\) and error 2015 is unrecoverable, \(\text{aux2}\) 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 \(\text{naux2}\) long-precision real numbers. On output, the contents are overwritten.

\(\text{naux2}\)

is the number of doublewords in the working storage specified in \(\text{aux2}\).

Specified as: an integer, where:

If \(\text{naux2} = 0\) 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, \(\text{naux2} \geq (\text{minimum value required for successful processing})\). 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 51.

On Return

\( y \) has the following meaning, where:

If \( init \neq 0 \), 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 206 on page 1085.

\( aux1 \)

is the working storage for this subroutine, where:

If \( init \neq 0 \), 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 \( init \neq 0 \) and \( init = 0 \). However, it can be reused after
   intervening calls to this subroutine with different arguments.

2. When using the ESSL SMP Libraries, for optimal performance, the number of
   threads specified should be the same for \( init \neq 0 \) and \( init = 0 \).

3. 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 \( inc1y \))
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 1307. 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 75.

Formulas

Processor-Independent Formulas for SCOSF for NAUX1 and NAUX2:

NAUX1 Formulas

For 32-bit integer arguments:
If \( n \leq 16384 \), use \( \text{naux1} = 40000. \)
If \( n > 16384 \), use \( \text{naux1} = 20000 + .30n. \)

**For 64-bit integer arguments:**

If \( n \leq 16384 \), use \( \text{naux1} = 50000. \)
If \( n > 16384 \), use \( \text{naux1} = 30000 + .30n. \)

**NAUX2 Formulas**

If \( n \leq 16384 \), use \( \text{naux2} = 25000. \)
If \( n > 16384 \), use \( \text{naux2} = 20000 + .32n. \)

For the transposed case, where \( \text{inc2x} = 1 \) or \( \text{inc2y} = 1 \), 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**

**For 32-bit integer arguments:**

If \( n \leq 16384 \), use \( \text{naux1} = 35000. \)
If \( n > 16384 \), use \( \text{naux1} = 20000 + .60n. \)

**For 64-bit integer arguments:**

If \( n \leq 16384 \), use \( \text{naux1} = 45000. \)
If \( n > 16384 \), use \( \text{naux1} = 30000 + .60n. \)

**NAUX2 Formulas**

If \( n \leq 16384 \), use \( \text{naux2} = 20000. \)
If \( n > 16384 \), use \( \text{naux2} = 20000 + .64n. \)

For the transposed case, where \( \text{inc2x} = 1 \) or \( \text{inc2y} = 1 \), 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:

\[
y_{ki} = \text{scale} \left( .5x_{0,j} + .5(-1)^{k}x_{n/2,d} + \sum_{j=1}^{n/2-1} x_{ji} \cos \left( jk \left( 2\pi / n \right) \right) \right)
\]

for:

\[
k = 0, 1, \ldots, n/2
\]

\[
i = 1, 2, \ldots, m
\]

where:
$x_{ji}$ are elements of the sequences in array $X$, where each sequence contains the $n/2+1$ real nonredundant data $x_{ji} \, j = 0, 1, \ldots, n/2$.

$y_{ki}$ are elements of the sequences in array $Y$, where each sequence contains the $n/2+1$ real nonredundant data $y_{ki} \, k = 0, 1, \ldots, n/2$.

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 on page 1363], [4 on page 1363], [26 on page 1364], and [27 on page 1364].

Two invocations of this subroutine are necessary:
1. With init $\neq 0$, 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 $ $\leq 0$
3. $inc2x$ or $inc2y $ $\leq 0$
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$\neq 0$, and naux2 is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

**Examples**

**Example 1**

This example shows an input array $X$ with a set of $m$ cosine sequences of length $n/2+1$, $\cos(jk(2\pi/n))$, $j = 0, 1, \ldots, n/2$, 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(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.

Note: Because NAUX2 = 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

<table>
<thead>
<tr>
<th>INIT</th>
<th>X INCX</th>
<th>INC2X</th>
<th>Y INCY</th>
<th>INC2Y</th>
<th>N</th>
<th>M</th>
<th>SCALE</th>
<th>AUX1</th>
<th>NAUX1</th>
<th>AUX2</th>
<th>NAUX2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL SCOSF(INIT, X, 1, 18, Y, 1, 18, 32, 4, SCALE, AUX1, 414, AUX2, 0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 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.

**Note:** Because NAUX2= 0, this subroutine dynamically allocates the AUX2
working storage.

Call Statement and Input:

```call scosf(init, x, 1, 18, y, 1, 18, 32, 4, scale, aux1, 414, aux2, 0)```

<table>
<thead>
<tr>
<th>INIT</th>
<th>X INC1X INC2X</th>
<th>Y INC1Y INC2Y</th>
<th>N</th>
<th>M</th>
<th>SCALE</th>
<th>AUX1</th>
<th>NAUX1</th>
<th>AUX2</th>
<th>NAUX2</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT</td>
<td>X INC1X INC2X</td>
<td>Y INC1Y INC2Y</td>
<td>N</td>
<td>M</td>
<td>SCALE</td>
<td>AUX1</td>
<td>NAUX1</td>
<td>AUX2</td>
<td>NAUX2</td>
</tr>
<tr>
<td>INIT</td>
<td>X INC1X INC2X</td>
<td>Y INC1Y INC2Y</td>
<td>N</td>
<td>M</td>
<td>SCALE</td>
<td>AUX1</td>
<td>NAUX1</td>
<td>AUX2</td>
<td>NAUX2</td>
</tr>
<tr>
<td>INIT</td>
<td>X INC1X INC2X</td>
<td>Y INC1Y INC2Y</td>
<td>N</td>
<td>M</td>
<td>SCALE</td>
<td>AUX1</td>
<td>NAUX1</td>
<td>AUX2</td>
<td>NAUX2</td>
</tr>
</tbody>
</table>

**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
element has frequencies k = 14, 15, 16, 17. Because only the sequence data has
changed, initialization does not have to be done again.

**Note:** Because NAUX2= 0, this subroutine dynamically allocates the AUX2
working storage.

Call Statement and Input:

```call scosf( 0 , x , 1 , 18 , y , 1 , 18 , 32 , 4 , scale , aux1 , 414 , aux2 , 0 )```

<table>
<thead>
<tr>
<th>SCALE</th>
<th>X contains the following four sequences:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0000</td>
<td>1.0000  1.0000  1.0000  1.0000</td>
</tr>
<tr>
<td>-0.9239</td>
<td>-0.9808 -1.0000 -0.9808</td>
</tr>
<tr>
<td>0.7071</td>
<td>0.9239  1.0000  0.9239</td>
</tr>
<tr>
<td>-0.3827</td>
<td>-0.8315 -1.0000 -0.8315</td>
</tr>
<tr>
<td>0.0000</td>
<td>0.7071  1.0000  0.7071</td>
</tr>
<tr>
<td>0.3827</td>
<td>-0.5556 -1.0000 -0.5556</td>
</tr>
<tr>
<td>-0.7071</td>
<td>0.3827  1.0000  0.3827</td>
</tr>
<tr>
<td>0.9239</td>
<td>-0.1951 -1.0000 -0.1951</td>
</tr>
<tr>
<td>-1.0000</td>
<td>0.0000  1.0000  0.0000</td>
</tr>
<tr>
<td>0.9239</td>
<td>0.1951  -1.0000  0.1951</td>
</tr>
<tr>
<td>-0.7071</td>
<td>-0.3827 -1.0000 -0.3827</td>
</tr>
<tr>
<td>0.3827</td>
<td>0.5556  -1.0000  0.5556</td>
</tr>
<tr>
<td>0.0000</td>
<td>-0.7071  1.0000 -0.7071</td>
</tr>
<tr>
<td>-0.3827</td>
<td>0.8315  -1.0000  0.8315</td>
</tr>
<tr>
<td>0.7071</td>
<td>-0.9239  1.0000 -0.9239</td>
</tr>
<tr>
<td>-0.9239</td>
<td>0.9808  -1.0000  0.9808</td>
</tr>
<tr>
<td>1.0000</td>
<td>-1.0000  1.0000 -1.0000</td>
</tr>
</tbody>
</table>

**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
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  0.0000  0.0000
0.0000  16.0000  0.0000  0.0000
  
.  .  .  .
SSINF and DSINF (Sine Transform)

Purpose

These subroutines compute a set of \( m \) real even discrete \( n \)-point Fourier transforms of sine sequences of real even data.

Table 207. Data Types

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SSINF</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DSINF</td>
</tr>
</tbody>
</table>

Note:

1. Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

Fortran

```fortran
CALL SSINF | DSINF (init, x, inc1x, inc2x, y, inc1y, inc2y, n, m, scale, aux1, naux1, aux2, naux2)
```

C and C++

```c
ssinf | dsinf (init, x, inc1x, inc2x, y, inc1y, inc2y, n, m, scale, aux1, naux1, aux2, naux2);
```

On Entry

- **init**
  - is a flag, where:
    - If \( init \neq 0 \), 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 \( init \neq 0 \).
  - Specified as: an 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 \( 1+(n/2-1)\text{inc}1x+(m-1)\text{inc}2x \), containing numbers of the data type indicated in Table 207. 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: an 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: an integer; \( inc2x > 0 \).

- **y**
  - See On Return
**inc1y**

is the stride between the elements within each sequence in array $Y$.

Specified as: an 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: an 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: an integer; $n \leq 37748736$ and must be one of the values listed in “Acceptable Lengths for the Transforms” on page 1028. 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 58.

**m**

is the number of sequences to be transformed.

Specified as: an integer; $m > 0$.

**scale**

is the scaling constant $scale$. See “Function” on page 1096 for its usage.

Specified as: a number of the data type indicated in Table 207 on page 1093, 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: an integer; $naux1 \geq$ (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 51.

**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: an 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, \( n_{aux2} \geq (\text{minimum value required for successful processing}). \) 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 51.

On Return

\( y \) has the following meaning, where:

If \( \text{init} \neq 0 \), this argument is not used, and its contents remain unchanged.

If \( \text{init} = 0 \), 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 \( 1+(n/2-1)\text{inc1y}+(m-1)\text{inc2y} \), containing numbers of the data type indicated in Table 207 on page 1093.

\( \text{aux1} \)
is the working storage for this subroutine, where:

If \( \text{init} \neq 0 \), it contains information ready to be passed in a subsequent invocation of this subroutine.

If \( \text{init} = 0 \), its contents are unchanged.

Returned as: the contents are not relevant.

Notes

1. \( \text{aux1} \) should not be used by the calling program between calls to this subroutine with \( \text{init} \neq 0 \) and \( \text{init} = 0 \). However, it can be reused after intervening calls to this subroutine with different arguments.

2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for \( \text{init} \neq 0 \) and \( \text{init} = 0 \).

3. For optimal performance, the preferred value for \( \text{inc1x} \) and \( \text{inc1y} \) is 1. This implies that the sequences are stored with stride 1. In addition, \( \text{inc2x} \) and \( \text{inc2y} \) 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, \( \text{inc2x} \) (or \( \text{inc2y} \)) is 1 and \( \text{inc1x} \) (or \( \text{inc1y} \)) 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 \( \text{inc1x} \) and \( \text{inc1y} \) for \_SINF, you should use "STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)" on page 1307. 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 \( \text{inc1x} \) and \( \text{inc1y} \) must be equal, and \( \text{inc2x} \) and \( \text{inc2y} \) must be equal. In this case, output overwrites input. If \( m = 1 \), the \( \text{inc2x} \) and \( \text{inc2y} \) 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 75.

Formulas

Processor-Independent Formulas for SSINF for NAUX1 and NAUX2:

NAUX1 Formulas

For 32-bit integer arguments:
If \( n \leq 16384 \), use \( naux1 = 60000 \).
If \( n > 16384 \), use \( naux1 = 20000+.30n \).

**For 64-bit integer arguments:**

If \( n \leq 16384 \), use \( naux1 = 70000 \).
If \( n > 16384 \), use \( naux1 = 30000+.30n \).

**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:

\[(n/4+257)(\text{min}(128, m)).\]

Processor-Independent Formulas for DSINF for NAUX1 and NAUX2:

**NAUX1 Formulas**

**For 32-bit integer arguments:**

If \( n \leq 16384 \), use \( naux1 = 50000 \).
If \( n > 16384 \), use \( naux1 = 20000+.60n \).

**For 64-bit integer arguments:**

If \( n \leq 16384 \), use \( naux1 = 60000 \).
If \( n > 16384 \), use \( naux1 = 30000+.60n \).

**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:

\[(n/2+257)(\text{min}(128, m)).\]

**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:

\[y_{ki} = scale \sum_{j=0}^{n/2-1} x_{ji} \sin(jk(2\pi/n))\]

for:

- \( k = 0, 1, \ldots, n/2-1 \)
- \( i = 1, 2, \ldots, m \)

where:

1096  ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
$x_{0i} = 0.0$

$x_{ji}$ 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 on page 1363], [4 on page 1363], [26 on page 1364], and [27 on page 1364].

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. $m ≤ 0$
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.

**Examples**

**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(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.

Note: Because NAUX2=0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

\[
\text{CALL SSINF(INIT, } X, 1, 18, Y, 1, 18, 32, 3, \text{SCALE, AUX1, 414, AUX2, 0)}
\]

\[
\begin{array}{cccccccc}
\text{INIT} & X & \text{INC1} & \text{INC2} & Y & \text{INCY} & \text{INC2Y} & N & M \\hline
1 & 1 & 1 & 1 & 1 & 1 & 32 & 3 & \text{SCALE} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \text{SCALE} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \text{SCALE} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \text{SCALE} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \text{SCALE} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \text{SCALE} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \text{SCALE} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \text{SCALE} \\
\end{array}
\]

\[
X \text{ contains the following three sequences:}
\begin{align*}
0.0000 & & 0.0000 & & 0.0000 \\
0.1951 & & 0.3827 & & 0.5556 \\
0.3827 & & 0.7071 & & 0.9239 \\
0.5556 & & 0.9239 & & 0.9808 \\
0.7071 & & 1.0000 & & 0.7071 \\
0.8315 & & 0.9239 & & 0.1951 \\
0.9239 & & 0.7071 & & -0.3827 \\
0.9808 & & 0.3827 & & -0.8315 \\
1.0000 & & 0.0000 & & -1.0000 \\
0.9808 & & -0.3827 & & -0.8315 \\
0.9239 & & -0.7071 & & -0.3827 \\
0.8315 & & -0.9239 & & 0.1951 \\
0.7071 & & -1.0000 & & 0.7071 \\
0.5556 & & -0.9239 & & 0.9808 \\
0.3827 & & -0.7071 & & 0.9239 \\
0.1951 & & -0.3827 & & 0.5556 \\
\end{align*}
\]

Output:

\[
Y \text{ contains the following three sequences:}
\begin{align*}
0.0000 & & 0.0000 & & 0.0000 \\
0.0000 & & 0.0000 & & 0.0000 \\
0.0000 & & 8.0000 & & 0.0000 \\
0.0000 & & 0.0000 & & 8.0000 \\
0.0000 & & 0.0000 & & 0.0000 \\
0.0000 & & 0.0000 & & 0.0000 \\
0.0000 & & 0.0000 & & 0.0000 \\
0.0000 & & 0.0000 & & 0.0000 \\
0.0000 & & 0.0000 & & 0.0000 \\
0.0000 & & 0.0000 & & 0.0000 \\
0.0000 & & 0.0000 & & 0.0000 \\
0.0000 & & 0.0000 & & 0.0000 \\
0.0000 & & 0.0000 & & 0.0000 \\
0.0000 & & 0.0000 & & 0.0000 \\
\end{align*}
\]

\[\text{Example 2}\]

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 \(\text{scale} = 4.0/n\), 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.

**Note:** Because \( NAUX2 = 0 \), this subroutine dynamically allocates the AUX2
working storage.

Call Statement and Input:

\[
\text{Call Statement and Input:}
\]
\[
\begin{align*}
\text{INIT } & = 1 \text{ (for initialization)} \\
\text{INIT } & = 0 \text{ (for computation)} \\
\text{SCALE } & = 4.0/32 \\
\text{X } & = (\text{same sequences as in output } Y \text{ in Example 1})
\end{align*}
\]

Output:

\[
\begin{align*}
Y & = (\text{same sequences as in output } X \text{ in Example 1}) \\
\text{Example 3}
\end{align*}
\]

This example shows another computation using the same arguments initialized
in Example 1 and using different input sequence data. The data for this
e example has frequencies \( k = 14, 15, 17 \). Because only the sequence data has
changed, initialization does not have to be done again.

**Note:** Because \( NAUX2 = 0 \), this subroutine dynamically allocates the AUX2
working storage.

Call Statement and Input:

\[
\begin{align*}
\text{Call Statement and Input:}
\end{align*}
\]
\[
\begin{align*}
\text{SCALE } & = 1.0 \\
\text{X contains the following three sequences:}
\end{align*}
\]
\[
\begin{align*}
0.0000 & \quad 0.0000 & \quad 0.0000 \\
0.3827 & \quad 0.1951 & \quad -0.1951 \\
-0.7071 & \quad -0.3827 & \quad 0.3827 \\
0.9239 & \quad 0.5556 & \quad -0.5556 \\
-1.0000 & \quad -0.7071 & \quad 0.7071 \\
0.9239 & \quad 0.8315 & \quad -0.8315 \\
-0.7071 & \quad -0.9239 & \quad 0.9239 \\
0.3827 & \quad 0.9808 & \quad -0.9808 \\
0.8573 & \quad -1.0000 & \quad 1.0000 \\
-0.3827 & \quad 0.9808 & \quad -0.9808 \\
0.7071 & \quad -0.9239 & \quad 0.9239 \\
-0.9239 & \quad 0.8315 & \quad -0.8315 \\
1.0000 & \quad -0.7071 & \quad 0.7071 \\
-0.9239 & \quad 0.5556 & \quad -0.5556 \\
0.7071 & \quad -0.3827 & \quad 0.3827 \\
-0.3827 & \quad 0.1951 & \quad -0.1951
\end{align*}
\]

Output:

\[
\begin{align*}
Y contains the following three sequences:
\end{align*}
\]
ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
SCFT2 and DCFT2 (Complex Fourier Transform in Two Dimensions)

Purpose

These subroutines compute the two-dimensional discrete Fourier transform of complex data.

Table 208. Data Types

<table>
<thead>
<tr>
<th>X, Y</th>
<th>scale</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>SCFT2</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>DCFT2</td>
</tr>
</tbody>
</table>

Note:
1. Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

Fortran
CALL SCFT2 | DCFT2 (init, x, inc1x, inc2x, y, inc1y, inc2y, n1, n2, isign, scale, aux1, aux2, aux1, aux2, aux2)

C and C++
scft2 | dcf2 (init, x, inc1x, inc2x, y, inc1y, inc2y, n1, n2, isign, scale, aux1, aux1, aux2, aux2, aux2);

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: an 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 208.

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 1104 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, \textit{inc1x} is the leading dimension of the array and \textit{inc1x} ≥ \textit{n2}.

Specified as: an integer; \textit{inc1x} > 0. If \textit{inc2x} = 1, then \textit{inc1x} ≥ \textit{n2}.

\textit{inc2x}

is the stride between the elements in array \textit{X} for the second dimension.

If the array is stored in the transposed form, \textit{inc2x} = 1.

If the array is stored in the normal form, \textit{inc2x} is the leading dimension of the array and \textit{inc2x} ≥ \textit{n1}.

Specified as: an integer; \textit{inc2x} > 0. If \textit{inc1x} = 1, then \textit{inc2x} ≥ \textit{n1}.

\textit{y}

See \textit{On Return}.

\textit{inc1y}

is the stride between the elements in array \textit{Y} for the first dimension.

If the array is stored in the normal form, \textit{inc1y} = 1.

If the array is stored in the transposed form, \textit{inc1y} is the leading dimension of the array and \textit{inc1y} ≥ \textit{n2}.

Specified as: an integer; \textit{inc1y} > 0. If \textit{inc2y} = 1, then \textit{inc1y} ≥ \textit{n2}.

\textit{inc2y}

is the stride between the elements in array \textit{Y} for the second dimension.

If the array is stored in the transposed form, \textit{inc2y} = 1.

If the array is stored in the normal form, \textit{inc2y} is the leading dimension of the array and \textit{inc2y} ≥ \textit{n1}.

Specified as: an integer; \textit{inc2y} > 0. If \textit{inc1y} = 1, then \textit{inc2y} ≥ \textit{n1}.

\textit{n1}

is the length of the first dimension of the two-dimensional data in the array to be transformed.

Specified as: an integer; \textit{n1} ≥ 37748736 and must be one of the values listed in “Acceptable Lengths for the Transforms” on page 1028. 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 58.

\textit{n2}

is the length of the second dimension of the two-dimensional data in the array to be transformed.

Specified as: an integer; \textit{n2} ≥ 37748736 and must be one of the values listed in “Acceptable Lengths for the Transforms” on page 1028. 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 58.

\textit{isign}

controls the direction of the transform, determining the sign \textit{Isign} of the exponents of \textit{W_{n1}} and \textit{W_{n2}}, where:

If \textit{isign} = positive value, \( \textit{isign} = + \) (transforming time to frequency).

If \textit{isign} = negative value, \( \textit{isign} = - \) (transforming frequency to time).

Specified as: an integer; \textit{isign} > 0 or \textit{isign} < 0.

\textit{scale}

is the scaling constant \textit{scale}. See “Function” on page 1105 for its usage.
Specified as: a number of the data type indicated in Table 208 on page 1101
where \( scale > 0.0 \) or \( scale < 0.0 \).

**aux1**
is the working storage for this subroutine, where:
- If \( \text{init} \neq 0 \), the working storage is computed.
- If \( \text{init} = 0 \), the working storage is used in the computation of the Fourier transforms.

Specified as: an area of storage, containing \( \text{naux1} \) long-precision real numbers.

**naux1**
is the number of doublewords in the working storage specified in \( \text{aux1} \).
Specified as: an integer; \( \text{naux1} \geq \) (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 51.

**aux2**
has the following meaning:
- If \( \text{naux2} = 0 \) and error 2015 is unrecoverable, \( \text{aux2} \) 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 \( \text{naux2} \) long-precision real numbers.
On output, the contents are overwritten.

**naux2**
is the number of doublewords in the working storage specified in \( \text{aux2} \).
Specified as: an integer, where:
- If \( \text{naux2} = 0 \) 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, \( \text{naux2} \geq \) (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 51.

**On Return**

**y**
has the following meaning, where:
- If \( \text{init} \neq 0 \), this argument is not used, and its contents remain unchanged.
- If \( \text{init} = 0 \), 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 \( Y(k1(\text{inc1y})+k2(\text{inc2y})) \) for \( k1 = 0, 1, ..., n1-1 \) and \( k2 = 0, 1, ..., n2-1 \).

Returned as: an array of (at least) length \( 1+(n1-1)\text{inc1y}+(n2-1)\text{inc2y} \), containing numbers of the data type indicated in Table 208 on page 1101.
- If \( \text{inc1y} = 1 \), the output array is stored in normal form, and \( \text{inc2y} \geq n1 \).
- If \( \text{inc2y} = 1 \), the output array is stored in transposed form, and \( \text{inc1y} \geq n2 \).

See “Notes” on page 1104 for more details.
**aux1**

is the working storage for this subroutine, where:

If \( \text{init} \neq 0 \), it contains information ready to be passed in a subsequent invocation of this subroutine.

If \( \text{init} = 0 \), 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 \( \text{init} \neq 0 \) and \( \text{init} = 0 \). However, it can be reused after intervening calls to this subroutine with different arguments.

2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for \( \text{init} \neq 0 \) and \( \text{init} = 0 \).

3. If you specify the same array for \( X \) and \( Y \), then \( \text{inc1x} \) must equal \( \text{inc1y} \), and \( \text{inc2x} \) must equal \( \text{inc2y} \). 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 75.

4. By appropriately specifying the \( \text{inc} \) 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 \( n2 \).

5. Although the \( \text{inc} \) arguments for each array can be arbitrary, in most cases, one of the \( \text{inc} \) arguments is 1 for each array. If \( \text{inc1} = 1 \), the array is stored in normal form; that is, the first dimension of the array is along the columns. In this case, \( \text{inc2} \) is the leading dimension of the array and must be at least \( n1 \). Conversely, if \( \text{inc2} = 1 \), the array is stored in the transposed form; that is, the first dimension of the array is along the rows. In this case, \( \text{inc1} \) 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 "[Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines](https://www.ibm.com/support/knowledgecenter/SSGSGT_5.5.1/essl/esslguide/stride.html)" on page 1307 to determine this optimal value. Example 4 in the STRIDE subroutine description explains how it is used to find either \( \text{inc1} \) or \( \text{inc2} \).

**Formulas**

Processor-Independent Formulas for SCFT2 for \( \text{NAUX1} \) and \( \text{NAUX2} \):

The required values of \( \text{naux1} \) and \( \text{naux2} \) depend on \( n1 \) and \( n2 \).

**AUX1 Formulas**

For 32-bit integer arguments:

If \( \max(n1, n2) \leq 8192 \), use \( \text{naux1} = 40000 \).
If \( \max(n_1, n_2) > 8192 \), use \( aux1 = 40000+1.14(n_1+n_2) \).

For 64-bit integer arguments:
- If \( \max(n_1, n_2) \leq 8192 \), use \( aux1 = 60000 \).
- If \( \max(n_1, n_2) > 8192 \), use \( aux1 = 60000+1.14(n_1+n_2) \).

**NAUX2 Formulas**
- If \( \max(n_1, n_2) < 252 \), use \( aux2 = 20000 \).
- If \( \max(n_1, n_2) \geq 252 \), use \( aux2 = 20000+(2r+256)(s+2.28) \), where \( r = \max(n_1, n_2) \) and \( s = \min(64, n_1, n_2) \).

**Processor-Independent Formulas for DCFT2 for NAUX1 and NAUX2:**

The required values of \( aux1 \) and \( aux2 \) depend on \( n_1 \) and \( n_2 \).

**NAUX1 Formulas**

**For 32-bit integer arguments:**
- If \( \max(n_1, n_2) \leq 2048 \), use \( aux1 = 40000 \).
- If \( \max(n_1, n_2) > 2048 \), use \( aux1 = 40000+2.28(n_1+n_2) \).

**For 64-bit integer arguments:**
- If \( \max(n_1, n_2) \leq 2048 \), use \( aux1 = 60000 \).
- If \( \max(n_1, n_2) > 2048 \), use \( aux1 = 60000+2.28(n_1+n_2) \).

**NAUX2 Formulas**
- If \( \max(n_1, n_2) < 252 \), use \( aux2 = 20000 \).
- If \( \max(n_1, n_2) \geq 252 \), use \( aux2 = 20000+(2r+256)(s+2.28) \), where \( r = \max(n_1, n_2) \) and \( s = \min(64, n_1, n_2) \).

**Function**

The two-dimensional discrete Fourier transform of complex data in array \( X \), with results going into array \( Y \), is expressed as follows:

\[
y_{k1,k2} = scale \sum_{j_1=0}^{n1-1} \sum_{j_2=0}^{n2-1} X_{j1,j2} W_{n1}^{(j1)k1} W_{n2}^{(j2)k2}
\]

for:
- \( k1 = 0, 1, ..., n1-1 \)
- \( k2 = 0, 1, ..., n2-1 \)

where:

\[
W_{n1} = e^{-2\pi(\sqrt{-1})/n1}
\]
\[
W_{n2} = e^{-2\pi(\sqrt{-1})/n2}
\]

and where:
$x_{i,j}$ are elements of array $X$.
$y_{i,j}$ 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 on page 1363], [4 on page 1363], and [27 on page 1364].

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.

**Examples**

**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(1)
```

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.
Note: Because NAUX2 = 0, this subroutine dynamically allocates the AUX2 working storage.

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, 0)
```

```
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(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.

Note: Because NAUX2 = 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
CALL DCFT2(INIT, X , 1 , 6 , Y , 8 , 1 , 6 , 8 , -1 , SCALE, AUX1, 20000 , AUX2, 0)
```

```
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)

Purpose

These subroutines compute the two-dimensional discrete Fourier transform of real data in a two-dimensional array.

Table 209. Data Types

<table>
<thead>
<tr>
<th>X, scale</th>
<th>Y</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision</td>
<td>Short-precision complex</td>
<td>SRCFT2</td>
</tr>
<tr>
<td>Long-precision</td>
<td>Long-precision complex</td>
<td>DRCFT2</td>
</tr>
</tbody>
</table>

Note:

1. Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

<table>
<thead>
<tr>
<th></th>
<th>Fortran</th>
<th>C and C++</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CALL SRCFT2 (init, x, inc2x, y, n1, n2, isign, scale, aux1, aux2, aux3, naux3)</td>
<td>srcf2 (init, x, inc2x, y, n1, n2, isign, scale, aux1, aux2, aux3, naux3);</td>
</tr>
<tr>
<td></td>
<td>CALL DRCFT2 (init, x, inc2x, y, n1, n2, isign, scale, aux1, aux2, aux3, naux3)</td>
<td>drcf2 (init, x, inc2x, y, n1, n2, isign, scale, aux1, aux2, aux3, naux3);</td>
</tr>
</tbody>
</table>

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: an 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 209. See “Notes” on page 1110 for more details.

inc2x

is the leading dimension (stride between columns) of array X. Specified as: an integer; inc2x ≥ n1.

y

See On Return
inc2y
is the leading dimension (stride between columns) of array Y. Specified as: an integer; $inc2y \geq ((n1)/2)+1$.

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: an integer; $n1 \leq 37748736$ and must be one of the values listed in “Acceptable Lengths for the Transforms” on page 1028. 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 58.

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: an integer; $n2 \leq 37748736$ and must be one of the values listed in “Acceptable Lengths for the Transforms” on page 1028. 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 58.

isign
controls the direction of the transform, determining the sign $Isign$ of the exponents of $W_{n1}$ and $W_{n2}$, where:

If $isign = \text{positive value}$, $Isign = +$ (transforming time to frequency).

If $isign = \text{negative value}$, $Isign = -$ (transforming frequency to time).

Specified as: an integer; $isign > 0$ or $isign < 0$.

scale
is the scaling constant scale. See “Function” on page 1112 for its usage.

Specified as: a number of the data type indicated in Table 209 on page 1108 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: an integer; $naux1 \geq$ (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 51.

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 \( \text{aux2} \) long-precision real numbers. On output, the contents are overwritten.

\( \text{aux2} \)

is the number of doublewords in the working storage specified in \( \text{aux2} \).

Specified as: an integer, where:

If \( \text{aux2} = 0 \) 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, \( \text{aux2} \geq \) (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 51.

\( \text{aux3} \)

this argument is provided for migration purposes only and is ignored.

Specified as: an area of storage containing \( \text{aux3} \) long-precision real numbers.

\( \text{aux3} \)

this argument is provided for migration purposes only and is ignored.

Specified as: an integer.

On Return

\( y \)

has the following meaning, where:

If \( \text{init} \neq 0 \), this argument is not used, and its contents remain unchanged.

If \( \text{init} = 0 \), 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 \( \left( \frac{n1}{2} \right) + 1 \) rows of the array—that is, \( y_{k1,k2} \), where \( k1 = 0, 1, ..., \left( \frac{n1}{2} \right) \) and \( k2 = 0, 1, ..., n2-1 \).

Returned as: an \( \text{inc2y} \) by \( n2 \) array, containing numbers of the data type indicated in Table 209 on page 1108.

\( \text{aux1} \)

is the working storage for this subroutine, where:

If \( \text{init} \neq 0 \), it contains information ready to be passed in a subsequent invocation of this subroutine.

If \( \text{init} = 0 \), its contents are unchanged.

Returned as: the contents are not relevant.

Notes

1. \( \text{aux1} \) should not be used by the calling program between calls to this subroutine with \( \text{init} \neq 0 \) and \( \text{init} = 0 \). However, it can be reused after intervening calls to this subroutine with different arguments.

2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for \( \text{init} \neq 0 \) and \( \text{init} = 0 \).

3. If you specify the same array for \( X \) and \( Y \), then \( \text{inc2x} \) must equal \( 2(\text{inc2y}) \). 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 75.
For selecting optimal strides (or leading dimensions \( \text{inc2x} \) and \( \text{inc2y} \)) for your input and output arrays, you should use the STRIDE subroutine (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines) on page 1307. Example 5 in the STRIDE subroutine description explains how it is used for these subroutines.

**Formulas**

**Processor-Independent Formulas for SRCFT2 for NAUX1 and NAUX2**

The required values of \( \text{naux1} \) and \( \text{naux2} \) depend on \( n1 \) and \( n2 \).

**NAUX1 Formulas**

For 32-bit integer arguments:

If \( \max(n1/2, n2) \leq 8192 \), use \( \text{naux1} = 45000 \).

If \( \max(n1/2, n2) > 8192 \), use \( \text{naux1} = 40000+0.82n1+1.14n2 \).

For 64-bit integer arguments:

If \( \max(n1/2, n2) \leq 8192 \), use \( \text{naux1} = 65000 \).

If \( \max(n1/2, n2) > 8192 \), use \( \text{naux1} = 60000+0.82n1+1.14n2 \).

**NAUX2 Formulas**

If \( n1 \leq 16384 \) and \( n2 < 252 \), use \( \text{naux2} = 20000 \).

If \( n1 > 16384 \) and \( n2 < 252 \), use \( \text{naux2} = 20000+0.57n1 \).

If \( n2 \geq 252 \), 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 \( \text{naux1} \) and \( \text{naux2} \) depend on \( n1 \) and \( n2 \).

**NAUX1 Formulas**

For 32-bit integer arguments:

If \( n \leq 2048 \), use \( \text{naux1} = 42000 \).

If \( n > 2048 \), use \( \text{naux1} = 40000+1.64n1+2.28n2 \), where \( n = \max(n1/2, n2) \).

For 64-bit integer arguments:

If \( n \leq 2048 \), use \( \text{naux1} = 62000 \).

If \( n > 2048 \), use \( \text{naux1} = 60000+1.64n1+2.28n2 \), where \( n = \max(n1/2, n2) \).

**NAUX2 Formulas**

If \( n1 \leq 4096 \) and \( n2 < 252 \), use \( \text{naux2} = 20000 \).

If \( n1 > 4096 \) and \( n2 < 252 \), use \( \text{naux2} = 20000+1.14n1 \).

If \( n2 \geq 252 \), 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:

\[
y_{k1,k2} = scale \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} x_{j1,j2} W_{n1}^{j1k1} W_{n2}^{j2k2}
\]

for:

\[
k1 = 0, 1, ..., n1-1
\]

\[
k2 = 0, 1, ..., n2-1
\]

where:

\[
W_{n1} = e^{-2\pi(\sqrt{-1})/n1}
\]

\[
W_{n2} = e^{-2\pi(\sqrt{-1})/n2}
\]

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 \([1 \text{ on page 1363}], [4 \text{ on page 1363}], [26 \text{ on page 1364}], \) and \([27 \text{ on page 1364}]\).

Two invocations of this subroutine are necessary:
1. With \( init \neq 0 \), 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 \( naux2 \neq 0 \), and \( naux2 \) is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

**Examples**

**Example 1**

This example shows how to compute a two-dimensional transform. The arrays are declared as follows:

```plaintext
COMPLEX*8  Y(0:6,0:7)
REAL*4     X(0:11,0:7)
REAL*8     AUX1(1000), AUX2(1), AUX3(1)
```

First, initialize \( AUX1 \) using the calling sequence shown below with \( INIT \neq 0 \). Then use the same calling sequence with \( INIT = 0 \) to do the calculation.

**Note:** Because \( NAUX2=0 \), this subroutine dynamically allocates the \( AUX2 \) working storage.

**Call Statement and Input:**

```plaintext
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, 0, 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.

**Note:** Because \( NAUX2=0 \), this subroutine dynamically allocates the \( AUX2 \) working storage.

**Call Statement and Input:**

```plaintext
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, 0, 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:

```plaintext
COMPLEX*16  Y(0:8,0:7)
REAL*8     X(0:19,0:7)
REAL*8     AUX1(1000), AUX2(1), AUX3(1)
```

Arrays $X$ and $Y$ are made equivalent by the following statement, making them occupy the same storage.

```plaintext
EQUIVALENCE (X,Y)
```

This requires $inc2x \geq 2(inc2y)$. First, initialize $AUX1$ using the calling sequence shown below with $INIT \neq 0$. Then use the same calling sequence with $INIT = 0$ to do the calculation.

**Note:** Because $NAUX2 = 0$, this subroutine dynamically allocates the $AUX2$ working storage.

**Call Statement and Input:**

```plaintext
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, 0, AUX3, 0)
```

- **INIT** = 1 (for initialization)
- **INIT** = 0 (for computation)
- **SCALE** = 1.0/128.0

**$X$**

```plaintext
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
```

**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)

Purpose

These subroutines compute the two-dimensional discrete Fourier transform of complex conjugate even data in a two-dimensional array.

Table 210. Data Types

<table>
<thead>
<tr>
<th>X</th>
<th>Y, scale</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>SCRFT2</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>DCRFT2</td>
</tr>
</tbody>
</table>

Note:
1. Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see "Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL" on page 32.

Syntax

Fortran

CALL SCRFT2 (init, x, inc2x, y, inc2y, n1, n2, isign, scale, aux1, aux2, aux3, aux4, aux5, aux6)

CALL DCRFT2 (init, x, inc2x, y, inc2y, n1, n2, isign, scale, aux1, aux2, aux3, aux4, aux5, aux6)

C and C++

scrft2 (init, x, inc2x, y, inc2y, n1, n2, isign, scale, aux1, aux2, aux3, aux4, aux5, aux6);

dcrft2 (init, x, inc2x, y, inc2y, n1, n2, isign, scale, aux1, aux2, aux3, aux4, aux5, aux6);

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: an 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, $x_{j1,0}, j1 = 0, 1, ... , (n1)/2, j2 = 0, 1, ... , n2-1$. 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 210.

inc2x

is the leading dimension (stride between columns) of array X. Specified as: an integer; inc2x ≥ ((n1)/2)+1.
$y$ See [On Return](#).

$inc2y$

is the leading dimension (stride between the columns) of array $Y$.

Specified as: an integer; $inc2y \geq n1+2$.

$n1$

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: an integer; $n1 \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 1028. 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 58.

$n2$

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: an integer; $n2 \leq 37748736$ and must be one of the values listed in "Acceptable Lengths for the Transforms" on page 1028. 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 58.

$isign$

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: an integer; $isign > 0$ or $isign < 0$.

$scale$

is the scaling constant $scale$. See "Function" on page 1119 for its usage.

Specified as: a number of the data type indicated in Table 210 on page 1115, 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 $aux$.

Specified as: an integer; $naux1 \geq$ (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 51.

$aux2$

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 \( \text{naux2} \) long-precision real numbers. On output, the contents are overwritten.

\( \text{naux2} \)

is the number of doublewords in the working storage specified in \( \text{aux2} \).

Specified as: an integer, where:

If \( \text{naux2} = 0 \) 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, \( \text{naux2} \geq (\text{minimum value required for successful processing}) \). 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 51.

\( \text{aux3} \)

this argument is provided for migration purposes only and is ignored.

Specified as: an area of storage, containing \( \text{naux3} \) long-precision real numbers.

\( \text{naux3} \)

this argument is provided for migration purposes only and is ignored.

Specified as: an integer.

On Return

\( y \)

has the following meaning, where:

If \( \text{init} \neq 0 \), this argument is not used, and its contents remain unchanged.

If \( \text{init} = 0 \), 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 \( \text{inc2y} \) by (at least) \( n2 \) array, containing numbers of the data type indicated in Table 210 on page 1115. See “Notes” for more details.

\( \text{aux1} \)

is the working storage for this subroutine, where:

If \( \text{init} \neq 0 \), it contains information ready to be passed in a subsequent invocation of this subroutine.

If \( \text{init} = 0 \), its contents are unchanged.

Returned as: the contents are not relevant.

Notes

1. \( \text{aux1} \) should not be used by the calling program between program calls to this subroutine with \( \text{init} \neq 0 \) and \( \text{init} = 0 \). However, it can be reused after intervening calls to this subroutine with different arguments.

2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for \( \text{init} \neq 0 \) and \( \text{init} = 0 \).

3. If you specify the same array for \( X \) and \( Y \), then \( 2(\text{inc}2x) \) must equal \( \text{inc2y} \). 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 75.
4. 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 1307. Example 6 in the STRIDE subroutine description explains how it is used for these subroutines.

**Formulas**

Processor-Independent Formulas for SCRFT2 for NAUX1 and NAUX2

The required values of \(naux1\) and \(naux2\) depend on \(n1\) and \(n2\).

**NAUX1 Formulas**

For 32-bit integer arguments:

If \(\max(n1/2, n2) \leq 8192\), use \(naux1 = 45000\). If \(\max(n1/2, n2) > 8192\), use \(naux1 = 40000+0.82n1+1.14n2\).

For 64-bit integer arguments:

If \(\max(n1/2, n2) \leq 8192\), use \(naux1 = 65000\). If \(\max(n1/2, n2) > 8192\), use \(naux1 = 60000+0.82n1+1.14n2\).

**NAUX2 Formulas**

If \(n1 \leq 16384\) and \(n2 < 252\), use \(naux2 = 20000\).

If \(n1 > 16384\) and \(n2 < 252\), use \(naux2 = 20000+0.57n1\).

If \(n2 \geq 252\), 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**

For 32-bit integer arguments:

If \(n \leq 2048\), use \(naux1 = 42000\). If \(n > 2048\), use \(naux1 = 40000+1.64n1+2.28n2\), where \(n = \max(n1/2, n2)\).

For 64-bit integer arguments:

If \(n \leq 2048\), use \(naux1 = 62000\).

If \(n > 2048\), use \(naux1 = 60000+1.64n1+2.28n2\), where \(n = \max(n1/2, n2)\).

**NAUX2 Formulas**

If \(n1 \leq 4096\) and \(n2 < 252\), use \(naux2 = 20000\). If \(n1 > 4096\) and \(n2 < 252\), use \(naux2 = 20000+1.14n1\).

If \(n2 \geq 252\), 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:

$$W_{n1} = e^{-2\pi(\sqrt{-1})/n1}$$
$$W_{n2} = e^{-2\pi(\sqrt{-1})/n2}$$

and where:

$x_{j1,j2}$ are elements of array $X$.
$y_{k1,k2}$ are elements of array $Y$.
$\text{isign}$ is $+$ or $-$ (determined by argument $\text{isign}$).
$\text{scale}$ is a scalar value.

Because of the complex conjugate symmetry, the output in array $Y$ is real. For $\text{scale} = 1.0$ and $\text{isign}$ being positive, you obtain the discrete Fourier transform, a function of frequency. The inverse Fourier transform is obtained with $\text{scale} = 1.0/((n1)(n2))$ and $\text{isign}$ being negative. See references [1 on page 1363], [4 on page 1363], and [27 on page 1364].

Two invocations of this subroutine are necessary:
1. With $\text{init} \neq 0$, the subroutine tests and initializes arguments of the program, setting up the aux1 working storage.
2. With $\text{init} = 0$, 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. $\text{inc2x} < (n1)/2+1$
4. $\text{inc2y} < n1+2$
5. \( \text{scale} = 0.0 \)
6. \( \text{isign} = 0 \)
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. \( \text{naux1} \) 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 \( \text{naux2} \neq 0 \), and \( \text{naux2} \) is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

**Examples**

**Example 1**

This example shows how to compute a two-dimensional transform. The arrays are declared as follows:

\[
\begin{align*}
\text{REAL*4} & \quad \text{Y}(0:13,0:7) \\
\text{COMPLEX*8} & \quad \text{X}(0:6,0:7) \\
\text{REAL*8} & \quad \text{AUX1}(1000), \quad \text{AUX2}(1), \quad \text{AUX3}(1)
\end{align*}
\]

First, initialize \( \text{AUX1} \) using the calling sequence shown below with \( \text{INIT} \neq 0 \).
Then use the same calling sequence with \( \text{INIT} = 0 \) to do the calculation.

**Note:** Because \( \text{NAUX2} = 0 \), this subroutine dynamically allocates the \( \text{AUX2} \) working storage.

**Call Statement and Input:**

\[
\text{CALL SCRFT2(INIT, X, 7, Y, 14, 12, 8, -1, SCALE, AUX1, 1000, AUX2, 0, AUX3, 0)}
\]

<table>
<thead>
<tr>
<th>INIT</th>
<th>X INC2X</th>
<th>Y INC2Y</th>
<th>N1</th>
<th>N2</th>
<th>ISIGN</th>
<th>SCALE</th>
<th>AUX1</th>
<th>NAUX1</th>
<th>AUX2</th>
<th>NAUX2</th>
<th>AUX3</th>
<th>NAUX3</th>
</tr>
</thead>
</table>

\( \text{INIT} = 1 \) (for initialization)
\( \text{INIT} = 0 \) (for computation)
\( \text{SCALE} = 1.0 \times 10^{-2} \)

\( \text{X} \) is an array with 7 rows and 8 columns with \((1.0, 0.0)\) in all locations.

Output:

\[
\begin{bmatrix}
1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
\end{bmatrix}
\]

\[
\text{Y} =
\begin{bmatrix}
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
\end{bmatrix}
\]

**Example 2**

This example shows another transform computation with different data using the same initialized array \( \text{AUX1} \) in Example 1.
Note: Because NAUX2 = 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

\[
\text{SCALE} = 1.0/96.0
\]

\[
X \text{ is an array with 7 rows and 8 columns having (96.0, 0.0) in location } X(0,0) \text{ and (0.0, 0.0) in all others.}
\]

Output:

\[
Y = \begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
\end{bmatrix}
\]

Example 3

This example shows the same array being used for input and output. The arrays are declared as follows:

\[
\begin{align*}
\text{REAL*8} & \quad Y(0:17,0:7) \\
\text{COMPLEX*16} & \quad X(0:8,0:7) \\
\text{REAL*8} & \quad \text{AUX1(1000)}, \text{AUX2(1)}, \text{AUX3(1)}
\end{align*}
\]

Arrays X and Y are made equivalent by the following statement, making them occupy the same storage.

\[
\text{EQUIVALENCE} (X,Y)
\]

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.

Note: Because NAUX2 = 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

\[
\text{SCALE} = 1.0
\]

\[
X \text{ is an array with 9 rows and 8 columns having (1.0, 1.0) in location } X(4,2) \text{ and (0.0, 0.0) in all others.}
\]
Output:

\[
\begin{bmatrix}
2.0 & 2.0 & -2.0 & 2.0 & 2.0 & -2.0 & -2.0 \\
2.0 & -2.0 & -2.0 & 2.0 & 2.0 & -2.0 & 2.0 \\
-2.0 & -2.0 & 2.0 & 2.0 & -2.0 & 2.0 & 2.0 \\
-2.0 & 2.0 & 2.0 & -2.0 & 2.0 & 2.0 & -2.0 \\
2.0 & 2.0 & -2.0 & 2.0 & 2.0 & -2.0 & -2.0 \\
-2.0 & -2.0 & 2.0 & 2.0 & -2.0 & 2.0 & 2.0 \\
-2.0 & 2.0 & 2.0 & -2.0 & 2.0 & 2.0 & -2.0 \\
-2.0 & 2.0 & 2.0 & -2.0 & 2.0 & 2.0 & -2.0 \\
-2.0 & 2.0 & 2.0 & -2.0 & 2.0 & 2.0 & -2.0 \\
\end{bmatrix}
\]

\[
Y = \begin{bmatrix}
2.0 & 2.0 & -2.0 & 2.0 & 2.0 & -2.0 & -2.0 \\
2.0 & -2.0 & -2.0 & 2.0 & 2.0 & -2.0 & 2.0 \\
-2.0 & -2.0 & 2.0 & 2.0 & -2.0 & 2.0 & 2.0 \\
-2.0 & 2.0 & 2.0 & -2.0 & 2.0 & 2.0 & -2.0 \\
2.0 & 2.0 & -2.0 & 2.0 & 2.0 & -2.0 & -2.0 \\
-2.0 & -2.0 & 2.0 & 2.0 & -2.0 & 2.0 & 2.0 \\
-2.0 & 2.0 & 2.0 & -2.0 & 2.0 & 2.0 & -2.0 \\
-2.0 & 2.0 & 2.0 & -2.0 & 2.0 & 2.0 & -2.0 \\
\end{bmatrix}
\]
SCFT3 and DCFT3 (Complex Fourier Transform in Three Dimensions)

Purpose

These subroutines compute the three-dimensional discrete Fourier transform of complex data.

Table 211. Data Types

<table>
<thead>
<tr>
<th>X, Y</th>
<th>scale</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>SCFT3</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>DCFT3</td>
</tr>
</tbody>
</table>

Note:
1. 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.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

Fortran

CALL SCFT3 | DCFT3 (x, inc2x, inc3x, y, inc2y, inc3y, n1, n2, n3, isign, scale, aux, naux)

C and C++

scft3 | dcf3 (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, $inc2x(\geq n1)$, and $inc3x(\geq (n2)(inc2x))$, respectively.

Specified as: an array, containing numbers of the data type indicated in Table 211. If the array is dimensioned $X(LDA1,LDA2,LDA3)$, then $LDA1 = inc2x$, $(LDA1)(LDA2) = inc3x$, and $LDA3 \geq n3$. For information on how to set up this array, see “Setting Up Your Data” on page 1031. For more details, see “Notes” on page 1125.

inc2x

is the stride between the elements in array X for the second dimension.

Specified as: an integer; inc2x $\geq n1$.

inc3x

is the stride between the elements in array X for the third dimension.

Specified as: an integer; inc3x $\geq (n2)(inc2x)$.

y See On Return

inc2y

is the stride between the elements in array Y for the second dimension.

Specified as: an integer; inc2y $\geq n1$.

inc3y

is the stride between the elements in array Y for the third dimension.
Specified as: an integer; \( inc3y \geq (n2)(inc2y) \).

\( n1 \) is the length of the first dimension of the three-dimensional data in the array to be transformed.

Specified as: an integer; \( n1 \leq 37748736 \) and must be one of the values listed in “Acceptable Lengths for the Transforms” on page 1028. 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 58.

\( n2 \) is the length of the second dimension of the three-dimensional data in the array to be transformed.

Specified as: an integer; \( n2 \leq 37748736 \) and must be one of the values listed in “Acceptable Lengths for the Transforms” on page 1028. 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 58.

\( n3 \) is the length of the third dimension of the three-dimensional data in the array to be transformed.

Specified as: an integer; \( n3 \leq 37748736 \) and must be one of the values listed in “Acceptable Lengths for the Transforms” on page 1028. 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 58.

\( isign \) 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: an integer; \( isign > 0 \) or \( isign < 0 \).

\( scale \) is the scaling constant \( scale \). See “Function” on page 1127 for its usage.

Specified as: a number of the data type indicated in Table 211 on page 1123, 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: an 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 51.

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 \( 1, inc2y \geq n1 \), and \( inc3y(\geq (n2)(inc2y)) \), respectively.

Returned as: an array, containing numbers of the data type indicated in Table 211 on page 1123. If the array is dimensioned \( Y(LDA1,LDA2,LDA3) \), then \( LDA1 = inc2y \), \( (LDA1)(LDA2) = inc3y \), and \( LDA3 \geq n3 \). For information on how to set up this array, see “Setting Up Your Data” on page 1031. 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 Libraries 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 75.

2. You should use “STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)” on page 1307 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 1031.

Formulas

Processor-Independent Formulas for SCFT3 for N Aux:

Use the following formulas for calculating \( naux \):

For 32-bit integer arguments:

1. If \( \max(n2, n3) < 252 \) and:

   \[
   \begin{align*}
   &\text{If } n1 \leq 8192, \text{ use } naux = 60000. \\
   &\text{If } n1 > 8192, \text{ use } naux = 60000+2.28n1.
   \end{align*}
   \]

2. If \( n2 \geq 252, n3 < 252 \), and:

   \[
   \begin{align*}
   &\text{If } n1 \leq 8192, \text{ use } naux = 60000+\lambda. \\
   &\text{If } n1 > 8192, \text{ use } naux = 60000+2.28n1+\lambda, \\
   &\text{where } \lambda = (n2+256)(s+2.28) \\
   &\text{and } s = \min(64, n1).
   \end{align*}
   \]

3. If \( n2 < 252, n3 \geq 252 \), and:

   \[
   \begin{align*}
   &\text{If } n1 \leq 8192, \text{ use } naux = 60000+\psi. \\
   &\text{If } n1 > 8192, \text{ use } naux = 60000+2.28n1+\psi,
   \end{align*}
   \]
where $\psi = (n3+256)(s+2.28)$
and $s = \min(64, (n1)(n2))$.

4. If $n2 \geq 252$ and $n3 \geq 252$, use the larger of the values calculated for cases 2 and 3 above.

**For 64-bit integer arguments:**

1. If $\max(n2, n3) < 252$ and:
   - If $n1 \leq 8192$, use $naux = 90000$.
   - If $n1 > 8192$, use $naux = 90000+2.28n1$.
2. If $n2 \geq 252$, $n3 < 252$, and:
   - If $n1 \leq 8192$, use $naux = 90000+\lambda$.
   - If $n1 > 8192$, use $naux = 90000+2.28n1+\lambda$,

   where $\lambda = (n2+256)(s+2.28)$
   and $s = \min(64, n1)$.
3. If $n2 < 252$, $n3 \geq 252$, and:
   - If $n1 \leq 8192$, use $naux = 90000+\psi$.
   - If $n1 > 8192$, use $naux = 90000+2.28n1+\psi$,

   where $\psi = (n3+256)(s+2.28)$
   and $s = \min(64, (n1)(n2))$.
4. If $n2 \geq 252$ and $n3 \geq 252$, 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 $naux$:

**For 32-bit integer arguments:**

1. If $\max(n2, n3) < 252$ and:
   - If $n1 \leq 2048$, use $naux = 60000$.
   - If $n1 > 2048$, use $naux = 60000+4.56n1$.
2. If $n2 \geq 252$, $n3 < 252$, and:
   - If $n1 \leq 2048$, use $naux = 60000+\lambda$.
   - If $n1 > 2048$, use $naux = 60000+4.56n1+\lambda$,

   where $\lambda = ((2)n2+256)(s+4.56)$
   and $s = \min(64, n1)$.
3. If $n2 < 252$, $n3 \geq 252$, and:
   - If $n1 \leq 2048$, use $naux = 60000+\psi$.
   - If $n1 > 2048$, use $naux = 60000+4.56n1+\psi$,

   where $\psi = ((2)n3+256)(s+4.56)$
   and $s = \min(64, (n1)(n2))$.
4. If $n2 \geq 252$ and $n3 \geq 252$, use the larger of the values calculated for cases 2 and 3 above.

**For 64-bit integer arguments:**
1. If \( \max(n_2, n_3) < 252 \) and:
   
   If \( n_1 \leq 2048 \), use \( naux = 90000 \).
   
   If \( n_1 > 2048 \), use \( naux = 90000 + 4.56n_1 \).

2. If \( n_2 \geq 252, n_3 < 252 \), and:
   
   If \( n_1 \leq 2048 \), use \( naux = 90000 + \lambda \).
   
   If \( n_1 > 2048 \), use \( naux = 90000 + 4.56n_1 + \lambda \),
   
   where \( \lambda = ((2)n_2 + 256)(s + 4.56) \)
   
   and \( s = \min(64, n_1) \).

3. If \( n_2 < 252, n_3 \geq 252 \), and:
   
   If \( n_1 \leq 2048 \), use \( naux = 90000 + \psi \).
   
   If \( n_1 > 2048 \), use \( naux = 90000 + 4.56n_1 + \psi \),
   
   where \( \psi = ((2)n_3 + 256)(s + 4.56) \)
   
   and \( s = \min(64, (n_1)(n_2)) \).

4. If \( n_2 \geq 252 \) and \( n_3 \geq 252 \), 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:

\[
y_{k_1, k_2, k_3} = \text{scale} \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \sum_{j_3=0}^{n_3-1} x_{j_1, j_2, j_3} W_{n_1}^{(isign)j_1 k_1} W_{n_2}^{(isign)j_2 k_2} W_{n_3}^{(isign)j_3 k_3}
\]

for:

\[
k_1 = 0, 1, ..., n_1-1
\]
\[
k_2 = 0, 1, ..., n_2-1
\]
\[
k_3 = 0, 1, ..., n_3-1
\]

where:

\[
W_{n_1} = e^{-2\pi(\sqrt{-1})/n_1}
\]
\[
W_{n_2} = e^{-2\pi(\sqrt{-1})/n_2}
\]
\[
W_{n_3} = e^{-2\pi(\sqrt{-1})/n_3}
\]

and where:

\( x_{j_1, j_2, j_3} \) are elements of array \( X \).

\( y_{k_1, k_2, k_3} \) 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 on page 1363], [4 on page 1363], [5 on page 1363], [26 on page 1364], and [27 on page 1364].

**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.

**Examples**

**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(1)
```
Arrays \(X\) and \(Y\) are made equivalent by the following statement, making them occupy the same storage:

\[
\text{EQUIVALENCE }(X,Y)
\]

**Note:** Because \(\text{NAUX} = 0\), this subroutine dynamically allocates the \(\text{AUX}\) working storage.

Call Statement and Input:

\[
\begin{align*}
\text{SCALE} &= 1.0 \\
X, \text{INC}2X, \text{INC}3X &\quad Y, \text{INC}2Y, \text{INC}3Y &\quad N1, N2, N3, \text{ISIGN}, \text{SCALE}, \text{AUX}, \text{NAUX} \\
\text{CALL SCFT3}(X, 32, 2080, Y, 32, 2056, 32, 64, 40, 1, \text{SCALE}, \text{AUX}, 0)
\end{align*}
\]

\(\text{SCALE} = 1.0\)

\(X\) 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 \(Y(\text{ij}k)\), where \(\text{ij} = 1, 2048\) and \(j = 1, 40\). It remains unchanged elsewhere.
SRCFT3 and DRCFT3 (Real-to-Complex Fourier Transform in Three Dimensions)

Purpose

These subroutines compute the three-dimensional discrete Fourier transform of real data in a three-dimensional array.

Table 212. Data Types

<table>
<thead>
<tr>
<th>$X$, scale</th>
<th>$Y$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision complex</td>
<td>SRCFT3</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision complex</td>
<td>DRCFT3</td>
</tr>
</tbody>
</table>

Note:

1. 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.

2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

Fortran

```fortran
CALL SRCFT3 | DRCFT3 (x, inc2x, inc3x, y, inc2y, inc3y, n1, n2, n3, isign, scale, aux, aux)
```

C and C++

```c
srct3 | drcft3 (x, inc2x, inc3x, y, inc2y, inc3y, n1, n2, n3, isign, scale, aux, aux);
```

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, $inc2x(\geq n1)$, and $inc3x(\geq (n2)(inc2x))$, respectively.

Specified as: an array, containing numbers of the data type indicated in Table 212. If the array is dimensioned $X(LDA1,LDA2,LDA3)$, then $LDA1 = inc2x$, $(LDA1)(LDA2) = inc3x$, and $LDA3 \geq n3$. For information on how to set up this array, see “Setting Up Your Data” on page 1031. For more details, see “Notes” on page 1132.

$inc2x$

is the stride between the elements in array $X$ for the second dimension.

Specified as: an integer; $inc2x \equiv n1$.

$inc3x$

is the stride between the elements in array $X$ for the third dimension.

Specified as: an integer; $inc3x \equiv (n2)(inc2x)$.

$y$ See On Return

$inc2y$

is the stride between the elements in array $Y$ for the second dimension.

Specified as: an integer; $inc2y \equiv n1/2+1$. 

1130 ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
inc3y
is the stride between the elements in array Y for the third dimension.
Specified as: an integer; inc3y \geq (n2)(inc2y).

n1 is the length of the first dimension of the three-dimensional data in the array
to be transformed.
Specified as: an integer; n1 \leq 37748736 and must be one of the values listed in
"Acceptable Lengths for the Transforms" on page 1028. 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 58.

n2 is the length of the second dimension of the three-dimensional data in the
array to be transformed.
Specified as: an integer; n2 \leq 37748736 and must be one of the values listed in
"Acceptable Lengths for the Transforms" on page 1028. 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 58.

n3 is the length of the third dimension of the three-dimensional data in the array
to be transformed.
Specified as: an integer; n3 \leq 37748736 and must be one of the values listed in
"Acceptable Lengths for the Transforms" on page 1028. 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 58.

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: an integer; isign > 0 or isign < 0.

scale is the scaling constant scale. See "Function" on page 1134 for its usage.
Specified as: a number of the data type indicated in Table 212 on page 1130
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: an 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, \( \text{naux} \geq (\text{minimum value required for successful processing}) \). 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 51.

**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 \( 1, inc2y( \equiv n1/2+1), \) and \( inc3y( \equiv (n2)(inc2y)) \), respectively.

Returned as: an array, containing numbers of the data type indicated in Table 212 on page 1130. If the array is dimensioned \( Y(LDA1,LDA2,LDA3) \), then \( LDA1 = inc2y, (LDA1)(LDA2) = inc3y, \) and \( LDA3 \geq n3. \) For information on how to set up this array, see "Setting Up Your Data" on page 1031. 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 Libraries 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 75.

2. 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 1307 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 1031.

**Formulas**

**Processor-Independent Formulas for SRCFT3 for NAUX**

Use the following formulas for calculating \( \text{naux} \):

For 32-bit integer arguments:

1. If \( \max(n2, n3) < 252 \) and:

   - If \( n1 \leq 16384 \), use \( \text{naux} = 65000 \).
   - If \( n1 > 16384 \), use \( \text{naux} = 60000+1.39n1 \).

2. If \( n2 \geq 252, n3 < 252, \) and:

   - If \( n1 \leq 16384, \) use \( \text{naux} = 65000+\lambda. \)
   - If \( n1 > 16384, \) use \( \text{naux} = 60000+1.39n1+\lambda, \)

   where \( \lambda = (n2+256)(s+2.28) \) and \( s = \min(64, 1+n1/2). \)

3. If \( n2 < 252, n3 \geq 252, \) and:
If \( n_1 \leq 16384 \), use \( n_{aux} = 65000+\psi \).
If \( n_1 > 16384 \), use \( n_{aux} = 60000+1.39n_1+\psi \),

where \( \psi = (n_3+256)(s+2.28) \) and \( s = \min(64, (n_2)(1+n_1/2)) \).

4. If \( n_2 \geq 252 \) and \( n_3 \geq 252 \), use the larger of the values calculated for cases 2 and 3 above.

For 64-bit integer arguments:

1. If \( \max(n_2, n_3) < 252 \) and:
   - If \( n_1 \leq 16384 \), use \( n_{aux} = 95000 \).
   - If \( n_1 > 16384 \), use \( n_{aux} = 90000+1.39n_1 \).

2. If \( n_2 \geq 252, n_3 < 252 \), and:
   - If \( n_1 \leq 16384 \), use \( n_{aux} = 95000+\lambda \).
   - If \( n_1 > 16384 \), use \( n_{aux} = 90000+1.39n_1+\lambda \),

where \( \lambda = (n_2+256)(s+2.28) \) and \( s = \min(64, 1+n_1/2) \).

3. If \( n_2 < 252, n_3 \geq 252 \), and:
   - If \( n_1 \leq 16384 \), use \( n_{aux} = 95000+\psi \).
   - If \( n_1 > 16384 \), use \( n_{aux} = 90000+1.39n_1+\psi \),

where \( \psi = (n_3+256)(s+2.28) \) and \( s = \min(64, (n_2)(1+n_1/2)) \).

4. If \( n_2 \geq 252 \) and \( n_3 \geq 252 \), use the larger of the values calculated for cases 2 and 3 above.

If \( \text{inc2x} \) or \( \text{inc3x} \) 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:

\[ n_2(1+n_1/2) \]

Processor-Independent Formulas for DRCFT3 for NAUX

Use the following formulas for calculating \( n_{aux} \):

For 32-bit integer arguments:

1. If \( \max(n_2, n_3) < 252 \) and:
   - If \( n_1 \leq 4096 \), use \( n_{aux} = 62000 \).
   - If \( n_1 > 4096 \), use \( n_{aux} = 60000+2.78n_1 \).

2. If \( n_2 \geq 252, n_3 < 252 \), and:
   - If \( n_1 \leq 4096 \), use \( n_{aux} = 62000+\lambda \).
   - If \( n_1 > 4096 \), use \( n_{aux} = 60000+2.78n_1+\lambda \),

where \( \lambda = ((2)n_2+256)(s+4.56) \)
   and \( s = \min(64, n_1/2) \).

3. If \( n_2 < 252, n_3 \geq 252 \), and:
   - If \( n_1 \leq 4096 \), use \( n_{aux} = 62000+\psi \).
   - If \( n_1 > 4096 \), use \( n_{aux} = 60000+2.78n_1+\psi \),

where \( \psi = ((2)n_3+256)(s+4.56) \)
   and \( s = \min(64, n_2(1+n_1/2)) \).
4. If \( n_2 \geq 252 \) and \( n_3 \geq 252 \), use the larger of the values calculated for cases 2 and 3 above.

**For 64-bit integer arguments:**

1. If \( \max(n_2, n_3) < 252 \) and:
   - If \( n_1 \leq 4096 \), use \( naux = 92000 \).
   - If \( n_1 > 4096 \), use \( naux = 90000 + 2.78n_1 \).

2. If \( n_2 \geq 252, n_3 < 252 \), and:
   - If \( n_1 \leq 4096 \), use \( naux = 92000 + \lambda \).
   - If \( n_1 > 4096 \), use \( naux = 90000 + 2.78n_1 + \lambda \),

   where \( \lambda = ((2^{n_2+256})s+4.56) \)
   
   and \( s = \min(64, n_1/2) \).

3. If \( n_2 < 252, n_3 \geq 252 \), and:
   - If \( n_1 \leq 4096 \), use \( naux = 92000 + \psi \).
   - If \( n_1 > 4096 \), use \( naux = 90000 + 2.78n_1 + \psi \),

   where \( \psi = ((2^{n_3+256})s+4.56) \)
   
   and \( s = \min(64, n_2(1+n_1/2)) \).

4. If \( n_2 \geq 252 \) and \( n_3 \geq 252 \), 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:

\[
y_{k_1,k_2,k_3} = scale \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \sum_{j_3=0}^{n_3-1} x_{j_1,j_2,j_3} W_{n_1}^{[i(n_{j_1}+1)]} W_{n_2}^{[i(n_{j_2}+2)]} W_{n_3}^{[i(n_{j_3}+3)]}
\]

for:
\[
k_1 = 0, 1, ..., n_1-1 \\
k_2 = 0, 1, ..., n_2-1 \\
k_3 = 0, 1, ..., n_3-1
\]

where:

\[
W_{n_1} = e^{-2\pi(i/n_1)} \\
W_{n_2} = e^{-2\pi(i/n_2)} \\
W_{n_3} = e^{-2\pi(i/n_3)}
\]

and where:
\( x_{j_1,j_2,j_3} \) are elements of array \( X \).
\( y_{k_1,k_2,k_3} \) are elements of array \( Y \).
\( \text{isign} \) is + or - (determined by argument \( \text{isign} \)).
\( \text{scale} \) is a scalar value.

The output in array \( Y \) is complex. For \( \text{scale} = 1.0 \) and \( \text{isign} \) being positive, you obtain the discrete Fourier transform, a function of frequency. The inverse Fourier transform is obtained with \( \text{scale} = 1.0/((n_1)(n_2)(n_3)) \) and \( \text{isign} \) being negative. See references \[1 on page 1363\], \[4 on page 1363\], \[5 on page 1363\], \[26 on page 1364\], and \[27 on page 1364\].

**Error conditions**

**Resource Errors**

Error 2015 is unrecoverable, \( \text{naux} = 0 \), and unable to allocate work area.

**Computational Errors**

None

**Input-Argument Errors**

1. \( n_1 > 37748736 \)
2. \( n_2 > 37748736 \)
3. \( n_3 > 37748736 \)
4. \( \text{inc}_2x < n_1 \)
5. \( \text{inc}_3x < (n_2)(\text{inc}_2x) \)
6. \( \text{inc}_2y < n_1/2+1 \)
7. \( \text{inc}_3y < (n_2)(\text{inc}_2y) \)
8. \( \text{scale} = 0.0 \)
9. \( \text{isign} = 0 \)
10. The length of one of the transforms in \( n_1, n_2, \) or \( n_3 \) is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
11. Error 2015 is recoverable or \( \text{naux} \neq 0 \), and \( \text{naux} \) is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

**Examples**

**Example**

This example shows how to compute a three-dimensional transform. In this example, \( \text{INC}_2x \geq (2) (\text{INC}_2y) \) and \( \text{INC}_3x \geq (2) (\text{INC}_3y) \), so that the same array can be used for both input and output. The STRIDE subroutine is called to select good values for the \( \text{INC}_2y \) and \( \text{INC}_3y \) strides. Using the transform lengths (\( N_1 = 32, N_2 = 64, \) and \( N_2 = 40 \)) along with the output data type (short-precision complex: 'C'), STRIDE is called once for each stride needed. First, it is called for \( \text{INC}_2y \):

```
CALL STRIDE (N2,N1/2+1,INC2Y,'C',0)
```

The output value returned for \( \text{INC}_2y \) is 17. (This value is equal to \( N_1/2+1 \).) Then STRIDE is called again for \( \text{INC}_3y \):

```
CALL STRIDE (N3,N2*INC2Y,INC3Y,'C',0)
```

The output value returned for \( \text{INC}_3y \) is 1088. Because \( \text{INC}_3y \) is a multiple of \( \text{INC}_2y \)—that is, \( \text{INC}_3y = (N_2)(\text{INC}_2y) \)—\( Y \) 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(1)
```

Arrays X and Y are made equivalent by the following statement, making them occupy the same storage:

```
EQUIVALENCE (X,Y)
```

**Note:** Because NAUX= 0, this subroutine dynamically allocates the AX working storage.

**Call Statement and Input:**

```
CALL SRCFT3( X , 34 , 2176 , Y , 17 , 1088 , 32 , 64 , 40 , 1 , SCALE , AUX , 0 )
```

```
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)

Purpose

These subroutines compute the three-dimensional discrete Fourier transform of complex conjugate even data in a three-dimensional array.

Table 213. Data Types

<table>
<thead>
<tr>
<th>X</th>
<th>Y, scale</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short precision</td>
<td>Short precision real</td>
<td>SCRFT2</td>
</tr>
<tr>
<td>Long precision</td>
<td>Long precision real</td>
<td>DCRFT2</td>
</tr>
</tbody>
</table>

Note:
1. 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.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

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);

On Entry

x is the array X, containing the three-dimensional data to be transformed, where each element $x_{j_1,j_2,j_3}$ using zero-based indexing, is stored in

$x(j_1 + j_2(inc2x)+j_3(inc3x))$ for $j_1 = 0, 1, ..., n1/2$, $j_2 = 0, 1, ..., n2-1$, and $j_3 = 0, 1, ..., n3-1$. Due to complex conjugate symmetry, the input consists of only the first $n1/2+1$ values along the first dimension of the array, for $j_1 = 0, 1, ..., n1/2$.

The strides for the elements in the first, second, and third dimensions are assumed to be 1, $inc2x(\geq n1/2+1)$, and $inc3x(\geq (n2)(inc2x))$, respectively.

Specified as: an array, containing numbers of the data type indicated in Table 213. If the array is dimensioned $x(LDA1,LDA2,LDA3)$, then $LDA1 = inc2x$, $(LDA1)(LDA2) = inc3x$, and $LDA3 \geq n3$. For information on how to set up this array, see “Setting Up Your Data” on page 1031. For more details, see “Notes” on page 1139.

inc2x

is the stride between the elements in array X for the second dimension.

Specified as: an integer; $inc2x(\geq n1/2+1)$.

inc3x

is the stride between the elements in array X for the third dimension.

Specified as: an integer; $inc3x(\geq (n2)(inc2x))$.

y See On Return

inc2y

is the stride between the elements in array Y for the second dimension.
Specified as: an integer; \( inc2y \geq n1+2 \).

**inc3y**
is the stride between the elements in array \( Y \) for the third dimension.
Specified as: an integer; \( inc3y \geq (n2)(inc2y) \).

**n1** is the length of the first dimension of the three-dimensional data in the array to be transformed.
Specified as: an integer; \( n1 \leq 37748736 \) and must be one of the values listed in “Acceptable Lengths for the Transforms” on page 1028. 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 58.

**n2** is the length of the second dimension of the three-dimensional data in the array to be transformed.
Specified as: an integer; \( n2 \leq 37748736 \) and must be one of the values listed in “Acceptable Lengths for the Transforms” on page 1028. 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 58.

**n3** is the length of the third dimension of the three-dimensional data in the array to be transformed.
Specified as: an integer; \( n3 \leq 37748736 \) and must be one of the values listed in “Acceptable Lengths for the Transforms” on page 1028. 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 58.

**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: an integer; \( isign > 0 \) or \( isign < 0 \).

**scale**
is the scaling constant \( scale \). See “Function” on page 1141 for its usage.
Specified as: a number of the data type indicated in Table 213 on page 1137, 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: an 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, \( naux \geq (\text{minimum value required for successful processing}) \). 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 51.

**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 \( Y(k1+k2(\text{inc2y})+k3(\text{inc3y})) \) 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, \( \text{inc2y} \geq (n2)(\text{inc2x}) \), and \( \text{inc3y} \geq (n2)(\text{inc3x}) \), respectively.

Returned as: an array, containing numbers of the data type indicated in Table 213 on page 1137. If the array is dimensioned \( Y(LDA1,LDA2,LDA3) \), then \( LDA1 = \text{inc2y} \), \( (LDA1)(LDA2) = \text{inc3y} \), and \( LDA3 \geq n3 \). For information on how to set up this array, see “Setting Up Your Data” on page 1031. For more details, see “Notes.”

**Notes**

1. If you specify the same array for \( X \) and \( Y \), then \( \text{inc2y} \) must equal \( (2)(\text{inc2x}) \) and \( \text{inc3y} \) must equal \( (2)(\text{inc3x}) \). 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 75.

2. You should use “STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)” on page 1307 to determine the optimal values for the strides \( \text{inc2y} \) and \( \text{inc3y} \) for your output array. To obtain the best performance, you should use \( \text{inc2x} = \text{inc2y}/2 \) and \( \text{inc3x} = \text{inc3y}/2 \). 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 1031.

**Formulas**

**Processor-Independent Formulas for SCRFT3 for Calculating NAUX**

Use the following formulas for calculating \( naux \):

For 32-bit integer arguments:

1. If \( \max(n2, n3) < 252 \) and:

   - If \( n1 \leq 16384 \), use \( naux = 65000 \).
   - If \( n1 > 16384 \), use \( naux = 60000 + 1.39n1 \).

2. If \( n1 \geq 252 \), \( n3 < 252 \), and:

   - If \( n1 \leq 16384 \), use \( naux = 65000 + \lambda \).
   - If \( n1 > 16384 \), use \( naux = 60000 + 1.39n1 + \lambda \),

   where \( \lambda = (n2+256)(s+2.28) \) and \( s = \min(64, 1+n1/2) \).

3. If \( n2 < 252 \), \( n3 \geq 252 \), and:
If \( n1 \leq 16384 \), use \( naux = 65000 + \psi \).
If \( n1 > 16384 \), use \( naux = 60000 + 1.39n1 + \psi \),

where \( \psi = (n3 + 256)(s + 2.28) \)
and \( s = \min(64, (n2)(1 + n1/2)) \).

4. If \( n2 \geq 252 \) and \( n3 \geq 252 \), use the larger of the values calculated for cases 2 and 3 above.

For 64-bit integer arguments:

1. If \( \max(n2, n3) < 252 \) and:
   
   If \( n1 \leq 16384 \), use \( naux = 95000 \).
   If \( n1 > 16384 \), use \( naux = 90000 + 1.39n1 \).
2. If \( n2 \geq 252 \), \( n3 < 252 \), and:
   
   If \( n1 \leq 16384 \), use \( naux = 95000 + \lambda \).
   If \( n1 > 16384 \), use \( naux = 90000 + 1.39n1 + \lambda \),

   where \( \lambda = (n2 + 256)(s + 2.28) \)
   and \( s = \min(64, (n2)(1 + n1/2)) \).
3. If \( n2 < 252 \), \( n3 \geq 252 \), and:
   
   If \( n1 \leq 16384 \), use \( naux = 95000 + \psi \).
   If \( n1 > 16384 \), use \( naux = 90000 + 1.39n1 + \psi \),

   where \( \psi = (n3 + 256)(s + 2.28) \)
   and \( s = \min(64, (n2)(1 + n1/2)) \).
4. If \( n2 \geq 252 \) and \( n3 \geq 252 \), 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 \( naux \):

For 32-bit integer arguments:

1. If \( \max(n2, n3) < 252 \) and:
   
   If \( n1 \leq 4096 \), use \( naux = 62000 \).
   If \( n1 > 4096 \), use \( naux = 60000 + 2.78n1 \).
2. If \( n2 \geq 252 \), \( n3 < 252 \), and:
   
   If \( n1 \leq 4096 \), use \( naux = 62000 + \lambda \).
   If \( n1 > 4096 \), use \( naux = 60000 + 2.78n1 + \lambda \),

   where \( \lambda = ((2)n2 + 256)(s + 4.56) \)
   and \( s = \min(64, n1/2) \).
3. If \( n2 < 252 \), \( n3 \geq 252 \), and:
   
   If \( n1 \leq 4096 \), use \( naux = 62000 + \psi \).
   If \( n1 > 4096 \), use \( naux = 60000 + 2.78n1 + \psi \),
where $\psi = (2)n3+256)(s+4.56)$
and $s = \text{min}(64, n2(1+n1/2))$.

4. If $n2 \geq 252$ and $n3 \geq 252$, use the larger of the values calculated for cases 2 and 3 above.

For 64-bit integer arguments:

1. If $\max(n2, n3) < 252$ and:
   - If $n1 \leq 4096$, use $\text{naux} = 92000$.  
   - If $n1 > 4096$, use $\text{naux} = 90000+2.78n1$.

2. If $n2 \geq 252$, $n3 < 252$, and:
   - If $n1 \leq 4096$, use $\text{naux} = 92000+\lambda$.
   - If $n1 > 4096$, use $\text{naux} = 90000+2.78n1+\lambda$,

   where $\lambda = ((2)n2+256)(s+4.56)$
   and $s = \text{min}(64, n1/2)$.

3. If $n2 < 252$, $n3 \geq 252$, and:
   - If $n1 \leq 4096$, use $\text{naux} = 92000+\psi$.
   - If $n1 > 4096$, use $\text{naux} = 90000+2.78n1+\psi$,

   where $\psi = (2)n3+256)(s+4.56)$
   and $s = \text{min}(64, n2(1+n1/2))$.

4. If $n2 \geq 252$ and $n3 \geq 252$, 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:

$$y_{k1,k2,k3} = \text{scale} \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} \sum_{j3=0}^{n3-1} x_{j1,j2,j3} W_{n1}^{(isign)j1k1} W_{n2}^{(isign)j2k2} W_{n3}^{(isign)j3k3}$$

for:

- $k1 = 0, 1, ..., n1-1$
- $k2 = 0, 1, ..., n2-1$
- $k3 = 0, 1, ..., n3-1$

where:

$$W_{n1} = e^{-2\pi(\sqrt{-1})/n1}$$
$$W_{n2} = e^{-2\pi(\sqrt{-1})/n2}$$
$$W_{n3} = e^{-2\pi(\sqrt{-1})/n3}$$

and where:
$x_{i,j,k}$ are elements of array $X$.
$y_{i,j,k}$ 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 on page 1363, 4 on page 1363, 5 on page 1363, 26 on page 1364, and 27 on page 1364].

**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.

**Examples**

**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:

```plaintext
  COMPLEX*8  X(17,64,40)
  REAL*4     Y(34,64,40)
  REAL*8     AUX(1)
```

Arrays X and Y are made equivalent by the following statement, making them occupy the same storage:

```plaintext
EQUIVALENCE (X,Y)
```

Note: Because NAUX= 0, this subroutine dynamically allocates the AX working storage.

Call Statement and Input:

```plaintext
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, 0 )
```

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 contains the convolution and correlation subroutine descriptions.
SCON and SCOR (Convolution or Correlation of One Sequence with One or More Sequences)

Purpose

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 1032.

Syntax

Fortran

```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++

```c
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 \( init \neq 0 \), 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: an 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 \( 1+ (N_h-1) \lvert inc1h \rvert \), containing short-precision real numbers.

- **inc1h** is the stride between the elements within the sequence in array \( h \).

Specified as: an 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: an integer; \( inc1x > 0 \).

- **inc2x** is the stride between the first elements of the sequences in array \( x \).

Specified as: an integer; \( inc2x > 0 \).

- **y** See On Return
\textit{inc1y} is the stride between the elements within each sequence in output array \( Y \).
Specified as: an integer; \( \text{inc1y} > 0 \).

\textit{inc2y} is the stride between the first elements of each sequence in output array \( Y \).
Specified as: an integer; \( \text{inc2y} > 0 \).

\textit{nh} is the number of elements, \( N_h \), in the sequence in array \( H \).
Specified as: an integer; \( N_h > 0 \).

\textit{nx} is the number of elements, \( N_x \), in each sequence in array \( X \).
Specified as: an integer; \( N_x > 0 \).

\textit{m} is the number of sequences in array \( X \) to be convolved or correlated.
Specified as: an integer; \( m > 0 \).

\textit{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: an integer. It can have any value.

\textit{ny} is the number of elements, \( N_y \), in each sequence in array \( Y \).
Specified as: an integer; \( N_y > 0 \) for SCON and \( N_y \geq -N_h+1 \) for SCOR.

\textit{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.

\textit{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.

\textit{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.

\textit{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.

\textbf{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 \( 1 + (m-1)\text{inc2y} + (N_y-1)\text{inc1y} \), containing short-precision real numbers.

\textbf{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 \textit{"Concepts" on page 75}.  
2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for \( \text{init} 
eq 0 \) and \( \text{init} = 0 \).
3. 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 \[24 on page 1364\] and \[102 on page 1369\].

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. $nh$, $nx$, $ny$, or $m \leq 0$
2. $inc1h$, $inc1x$, $inc2x$, $inc1y$, or $inc2y \leq 0$

Examples

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:

```
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:
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:

\[
\begin{align*}
20.0 &\quad 18.0 &\quad 22.0 \\
30.0 &\quad 20.0 &\quad 18.0 \\
40.0 &\quad 30.0 &\quad 20.0 \\
48.0 &\quad 40.0 &\quad 30.0 \\
52.0 &\quad 48.0 &\quad 40.0 \\
50.0 &\quad 52.0 &\quad 48.0 \\
40.0 &\quad 50.0 &\quad 52.0 \\
29.0 &\quad 38.0 &\quad 47.0 \\
18.0 &\quad 25.0 &\quad 32.0 \\
8.0 &\quad 12.0 &\quad 16.0
\end{align*}
\]

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:

```plaintext
REAL*4 H(0:4999), X(0:49999), Y(0:49999)
REAL*8 AUX1, AUX2
```

Call Statement and Input:

```plaintext
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:

\[
\begin{align*}
1.0 &\quad 2.0 &\quad 3.0 \\
2.0 &\quad 1.0 &\quad 2.0 \\
3.0 &\quad 2.0 &\quad 1.0 \\
4.0 &\quad 3.0 &\quad 2.0 \\
5.0 &\quad 4.0 &\quad 3.0 \\
6.0 &\quad 5.0 &\quad 4.0 \\
5.0 &\quad 6.0 &\quad 5.0 \\
4.0 &\quad 5.0 &\quad 6.0 \\
3.0 &\quad 4.0 &\quad 5.0 \\
2.0 &\quad 3.0 &\quad 4.0
\end{align*}
\]

Output:

\( Y \) contains the following three sequences:
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)

**Purpose**

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.

**Note:**

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 on page 1363].
2. For long-precision data, you should use DDCON or DDCOR with the decimation rate, \( id \), equal to 1.
3. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see Reference [Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL] on page 32.

**Syntax**

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SCOND</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SCORD (h, inch, x, incx, y, incy, nh, nx, iy0, ny)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C and C++</th>
<th>scond</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>scord (h, inch, x, incx, y, incy, nh, nx, iy0, ny);</td>
</tr>
</tbody>
</table>

**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 \( 1+(N_h-1) \mid inch \mid \), containing short-precision real numbers.

- **inch** is the stride between the elements within the sequence in array \( H \).
  Specified as: an 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 \( 1+(N_x-1) \mid incx \mid \), containing short-precision real numbers.

- **incx** is the stride between the elements within the sequence in array \( X \).
  Specified as: an 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: an integer; \( incy > 0 \) or \( incy < 0 \).

- **nh** is the number of elements, \( N_h \), in the sequence in array \( H \).
  Specified as: an integer; \( N_h > 0 \).
\(nx\) is the number of elements, \(N_x\), in the sequence in array \(X\).

Specified as: an 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: an integer. It can have any value.

\(ny\) is the number of elements, \(N_y\), in the sequence in array \(Y\).

Specified as: an 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 \(1+(N_y-1)\lfloor incy \rceil\), 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 75.

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 SCOND and \((- nh+1) \leq k \leq (nx-1)\) 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:

\(y_i\) are elements of the sequence of length \(N_y\) in array \(Y\).

\(x_i\) are elements of the sequence of length \(N_x\) in array \(X\).

\(h_i\) are elements of the sequence of length \(N_h\) in array \(H\).
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 on page 1363].

**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 $ny \leq 0$
2. $inch$, $incx$, or $incy = 0$

**Examples**

**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:

```fortran
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:

```fortran
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:
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 )
```
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:

\[
\begin{array}{ccccccccccc}
H & INCH & X & INCX & Y & INCY & NH & NX & IY0 & NY \\
\hline
| | | | | | | | | | |
\end{array}
\]

\[
\text{CALL SCORD( H, 1, X, 1, Y, 1, 4, 8, -5, 15 )}
\]

Output:

\[
Y = ( 0.0, 0.0, 44.0, 81.0, 110.0, 130.0, 140.0, 160.0, 170.0, 104.0, 53.0, 18.0, 0.0, 0.0 )
\]

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:

\[
\begin{array}{ccccccccccc}
H & INCH & X & INCX & Y & INCY & NH & NX & IY0 & NY \\
\hline
| | | | | | | | | | |
\end{array}
\]

\[
\text{CALL SCORD( X, 1, H, 1, Y, 1, 4, 8, -7, 11 )}
\]

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:

\[
\begin{array}{ccccccccccc}
H & INCH & X & INCX & Y & INCY & NH & NX & IY0 & NY \\
\hline
| | | | | | | | | | |
\end{array}
\]

\[
\text{CALL SCORD( H, -1, X, 1, Y, 1, 4, 8, -3, 11 )}
\]

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)

Purpose

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:

1. Two invocations of these subroutines are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.
2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SCONF</th>
<th>SCORF (init, h, inc1h, x, inc1x, inc2x, y, inc1y, inc2y, nh, nx, m, iy0, ny, aux1, aux1, aux2, aux2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>sconf</td>
<td>scorf (init, h, inc1h, x, inc1x, inc2x, y, inc1y, inc2y, nh, nx, m, iy0, ny, aux1, aux1, aux2, aux2);</td>
</tr>
</tbody>
</table>

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: an integer. It can have any value.

`h`

is the array `H`, consisting of the sequence of length `Nh` to be convolved or correlated with the sequences in array `X`.

Specified as: an array of (at least) length `1+(Nh-1)*inc1h`, containing short-precision real numbers.

`inc1h`

is the stride between the elements within the sequence in array `H`.

Specified as: an integer; `inc1h > 0`.

`x`

is the array `X`, consisting of `m` input sequences of length `Nx`, each to be convolved or correlated with the sequence in array `H`.

Specified as: an array of (at least) length `1+(Nx-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: an integer; `inc1x > 0`. 
inc2x
is the stride between the first elements of the sequences in array X.
Specified as: an integer; inc2x > 0.

inc1y
is the stride between the elements within each sequence in output array Y.
Specified as: an integer; inc1y > 0.

inc2y
is the stride between the first elements of each sequence in output array Y.
Specified as: an integer; inc2y > 0.

nh
is the number of elements, \( N_h \), in the sequence in array H.
Specified as: an integer; \( N_h > 0 \).

nx
is the number of elements, \( N_x \), in each sequence in array X.
Specified as: an integer; \( N_x > 0 \).

m
is the number of sequences in array X to be convolved or correlated.
Specified as: an 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: an integer. It can have any value.

ny
is the number of elements, \( N_y \), in each sequence in array Y.
Specified as: an integer; \( N_y > 0 \).

aux1
is the working storage for this subroutine, where:

If \( \text{init} \neq 0 \), the working storage is computed.

If \( \text{init} = 0 \), the working storage is used in the computation of the convolutions.
Specified as: an area of storage, containing \( \text{naux1} \) long-precision real numbers.

naux1
is the number of doublewords in the working storage specified in aux1.
Specified as: an integer; \( \text{naux1} > 23 \) (32-bit integer arguments) or 45 (64-bit integer arguments) and \( \text{naux1} \geq \) (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For values between 23 (32-bit integer arguments) or 45 (64-bit integer arguments) 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 51.

aux2
has the following meaning:

If \( \text{aux2} = 0 \) 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 \( \text{naux2} \) 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: an integer, where:

If $\text{naux2} = 0$ 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, $\text{naux2} \geq$ (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 51.

**On Return**

$y$ has the following meaning, where:

If $\text{init} \neq 0$, this argument is not used, and its contents remain unchanged.

If $\text{init} = 0$, 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 $1+(N_y-1)\text{inc1y}+(m-1)\text{inc2y}$, containing short-precision real numbers.

**aux1**

is the working storage for this subroutine, where:

If $\text{init} \neq 0$, it contains information ready to be passed in a subsequent invocation of this subroutine.

If $\text{init} = 0$, 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 $\text{init} \neq 0$ and $\text{init} = 0$. However, it can be reused after intervening calls to this subroutine with different arguments.

2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for $\text{init} \neq 0$ and $\text{init} = 0$.

3. If you specify the same array for $X$ and $Y$, then $\text{inc1x}$ and $\text{inc1y}$ must be equal, and $\text{inc2x}$ and $\text{inc2y}$ must be equal. In this case, output overwrites input.

4. If you specify different arrays for $X$ and $Y$, they must have no common elements; otherwise, results are unpredictable. See “Concepts” on page 75.

5. If $iy0$ and $ny$ are such that output outside the basic range is needed, the subroutine stores zeros. These ranges are: $0 \leq k \leq N_x+N_y-2$ for SCONF and $1-N_h \leq k \leq N_x-1$ for SCORF.

**Formulas**

**Formulas for the Length of the Fourier Transform**

Before calculating the necessary sizes of *aux1* 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 $nf$:

\[
\begin{align*}
    iy0p &= \max(iy0, 0) \\
    ix0 &= \max((iy0p+1)-nh, 0) \\
    ih0 &= \max((iy0p+1)-nx, 0) \\
    nd &= ix0+ih0 \\
    n1 &= iy0+ny \\
    nxx &= \min(n1, nx)-ix0 \\
    nhh &= \min(n1, nh)-ih0 \\
    ntt &= nxx+nhh-1 \\
    nn1 &= n1-nd \\
    iyy0 &= iy0p-nd \\
    nzleft &= \max(0, nhh-iyy0-1) \\
    nzrt &= \min(nn1, ntt)-nxx \\
    nf &= \max(12, nxx+\max(nzleft, nzrt))
\end{align*}
\]

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 1028, 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.

**NAUX1 Formulas**

For 32-bit integer arguments:

If $n \leq 16384$, use $naux1 = 58000$.

If $n > 16384$, use $naux1 = 40000+2.14n$.

For 64-bit integer arguments:

If $n \leq 16384$, use $naux1 = 78000$.

If $n > 16384$, use $naux1 = 60000+2.14n$.

**NAUX2 Formulas**

If $n \leq 16384$, use $naux2 = 30000$.

If $n > 16384$, use $naux2 = 20000+1.07n$.

**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:
\[ y_{ki} = \min(N_x - 1, N_y - 1 - k) \]

\[ \sum_{j = \max(0, -k)} h_j x_{k+j} \]

for:

\[ k = iy_0, iy_0+1, ..., iy_0+N_y-1 \]
\[ i = 1, 2, ..., m \]

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 \( H \).

\( iy_0 \) 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 \( \text{naux1} \) and \( \text{naux2} \) 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 [24 on page 1364] and [102 on page 1369].

Two invocations of this subroutine are necessary:

1. With \( \text{init} \neq 0 \), the subroutine tests and initializes arguments of the program, setting up the \( aux1 \) working storage.
2. With \( \text{init} = 0 \), 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, \( \text{naux2} = 0 \), and unable to allocate work area.

**Computational Errors**

None

**Input-Argument Errors**

1. \( nh, nx, ny, \) or \( m \leq 0 \)
2. \( \text{inc1h}, \text{inc1x}, \text{inc2x}, \text{inc1y}, \) or \( \text{inc2y} \leq 0 \)
3. The resulting internal Fourier transform length \( n \), is too large. See "Convolutions and Correlations by Fourier Methods" on page 1034.
4. The subroutine has not been initialized with the present arguments.
5. \( \text{naux1} \leq 23 \)
6. \( \text{naux1} \) 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 \( \text{naux2} \neq 0 \), and \( \text{naux2} \) is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

**Examples**

**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 \( \text{AUX1} \) using the calling sequence shown below with \( \text{INIT} \neq 0 \). Then use the same calling sequence with \( \text{INIT} = 0 \) to do the calculation.

**Note:** Because \( \text{NAUX2} = 0 \), this subroutine dynamically allocates the \( \text{AUX2} \) working storage.

**Call Statement and Input:**

```
CALL SCONF(INIT, H, 8, X, 10, 1, Y, 1, 1, 8, 10, 1, 0, 17, AUX1, 128, AUX2, 0)
```

```
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 \( \text{NY} \) is 21 rather than 17, and the value for \( \text{IY0} \) 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 \( \text{AUX1} \) using the calling sequence shown below with \( \text{INIT} \neq 0 \). Then use the same calling sequence with \( \text{INIT} = 0 \) to do the calculation.

**Note:** Because \( \text{NAUX2} = 0 \), this subroutine dynamically allocates the \( \text{AUX2} \) working storage.

**Call Statement and Input:**
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 \( A\)\textsc{ux1} using the calling sequence shown below with \( \text{INIT} \neq 0 \). Then use the same calling sequence with \( \text{INIT} = 0 \) to do the calculation.

**Note:** Because \( \text{NAUX2} = 0 \), this subroutine dynamically allocates the \( \text{AUX2} \) working storage.

**Call Statement and Input:**

```
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
34.0
70.0
120.0
185.0
266.0
364.0
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
```

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:

```plaintext
REAL*4  H(8), X(10,1)
```

First, initialize AUX1 using the calling sequence shown below with INIT \( \neq 0 \). Then use the same calling sequence with INIT = 0 to do the calculation.

**Note:** Because NAUX2 = 0, this subroutine dynamically allocates the AUX2 working storage.

**Call Statement and Input:**

```plaintext
CALL SCORF(INIT, H, 1, X, 1, 1, Y, 1, 1, 8, 10, 1, -7, 17, AUX1, 128, AUX2, 0)
```

**Output:**

- **INIT** = 1(for initialization)
- **INIT** = 0(for computation)
- **H** = (same as input \( H \) in Example 1)
- **X** = (same as input \( X \) in Example 1)

**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)

**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 \( \neq 0 \). Then use the same calling sequence with INIT = 0 to do the calculation.

**Note:** Because NAUX2 = 0, this subroutine dynamically allocates the AUX2 working storage.

**Call Statement and Input:**

```plaintext
CALL SCORF(INIT, X, 1, H, 1, 1, Y, 1, 1, 8, 10, 1, -9, 17, AUX1, 128, AUX2, 0)
```

**Output:**

- **INIT** = 1(for initialization)
- **INIT** = 0(for computation)
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.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

\[
\text{INIT H INCIH X INCIX INC2X Y INC1Y INC2Y NH NX M IY0 NY AUX1 NAUX1 AUX2 NAUX2}
\]
\[
\text{CALL SCONF(INIT, H, 1, H, 1, 1, Y, 1, 1, 8, 8, 1, -7, 15, AUX1, 148, AUX2, 0)}
\]

\[
\text{INIT = 1(for initialization)}
\]
\[
\text{INIT = 0(for computation)}
\]

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.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

\[
\text{INIT H INCIH X INCIX INC2X Y INC1Y INC2Y NH NX M IY0 NY AUX1 NAUX1 AUX2 NAUX2}
\]
\[
\text{CALL SCONF(INIT, H, 1, X, 1, 10, Y, 1, 17, 8, 10, 2, -7, 17, AUX1, 148, AUX2, 0)}
\]

\[
\text{INIT = 1(for initialization)}
\]
\[
\text{INIT = 0(for computation)}
\]
\[
\text{H = (same as input H in Example 4)}
\]
\[
\text{X = (same as input X in Example 4)}
\]

Output:

Y contains the following two sequences:

\[
88.0 \quad 96.0
\]
\[
173.0 \quad 188.0
\]
\[
254.0 \quad 275.0
\]
\[
330.0 \quad 356.0
\]
\[
400.0 \quad 430.0
\]
\[
463.0 \quad 496.0
\]
\[
518.0 \quad 553.0
\]
\[
564.0 \quad 600.0
\]
SDCON, DDCON, SDCOR, and DDCOR (Convolution or Correlation with Decimated Output Using a Direct Method)

Purpose

These subroutines compute the convolution and correlation of a sequence with another sequence, with decimated output, using a direct method.

Table 214. Data Types

<table>
<thead>
<tr>
<th>H, x, y</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SDCON</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DDCON</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>SDCOR</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DDCOR</td>
</tr>
</tbody>
</table>

Note:

1. 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.

2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

Fortran: CALL SDCON | DDCON | SDCOR | DDCOR (h, inch, x, incx, y, iny, nh, nx, iy0, ny, id)
C and C++: sdcon | ddcon | sdcor | ddcor (h, inch, x, incx, y, iny, 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 214.

\(inch\)

is the stride between the elements within the sequence in array \(H\).

Specified as: an 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 214.

\(incx\)

is the stride between the elements within the sequence in array \(X\).

Specified as: an 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: an integer; $incy > 0$ or $incy < 0$.

**nh**

is the number of elements, $N_{hydr}$, in the sequence in array $H$.

Specified as: an integer; $N_{hydr} > 0$.

**nx**

is the number of elements, $N_{x}$, in the sequence in array $X$.

Specified as: an 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: an integer. It can have any value.

**ny**

is the number of elements, $N_{y}$, in the sequence in array $Y$.

Specified as: an 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: an 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 214 on page 1167.

**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 $iy0 \geq N_{hydr}-1$ for _DCON and _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 75.

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:
Correlation for SDCOR and DDCOR:

\[ y_k = \min(N_y-1,k) \sum_{j = \max(0,k-N_x+1)} h_j x_{k-j} \]

\[ y_k = \min(N_y-1, N_x-1-k) \sum_{j = \max(0,-k)} h_j x_{k+j} \]

for \( k = iy0, iy0+id, iy0+(2)id, \ldots, iy0+(N_y-1)id \)

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 \( Y \).

\( \text{min} \) and \( \text{max} \) select the minimum and maximum values, respectively.

It is assumed that elements outside the range of definition are zero. See reference [4 on page 1363].

**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. (See SCOND [Example 1]). 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 SCORD [Example 6]. 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**
Examples

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 for SCOND and using a decimation interval \( ID = 2 \).

**Note:** For further examples of use, see SCOND Example 1. 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:

\[
\begin{align*}
\text{CALL SDCON}(H, 1, X, 1, Y, 1, 4, 8, 0, 6, 2) \\
H &= (1.0, 2.0, 3.0, 4.0) \\
X &= (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
\end{align*}
\]

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 for SCORD and using a decimation interval \( ID = 2 \).

**Note:** For further examples of use, see SCORD Example 6. 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:

\[
\begin{align*}
\text{CALL SDCOR}(H, 1, X, 1, Y, 1, 4, 8, -3, 6, 2) \\
H &= (1.0, 2.0, 3.0, 4.0) \\
X &= (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
\end{align*}
\]

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 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:

\[
\begin{align*}
\text{CALL SDCON}(H, 1, X, 1, Y, 1, 4, 8, 0, 11, 1) \\
H &= (1.0, 2.0, 3.0, 4.0) \\
X &= (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)
\end{align*}
\]
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)

Purpose

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 on page 1363].

Syntax

<table>
<thead>
<tr>
<th></th>
<th>Fortran</th>
<th>C and C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL SACOR</td>
<td>(init, x, inc1x, inc2x, y, inc1y, inc2y, nx, m, ny, aux1, aux2, aux2)</td>
<td>sacor (init, x, inc1x, inc2x, y, inc1y, inc2y, nx, m, ny, aux1, aux2, aux2);</td>
</tr>
</tbody>
</table>

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: an integer. It can have any value.

x is the array X, consisting of m input sequences of length Nₓ, to be autocorrelated. Specified as: an array of (at least) length 1+(Nₓ-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: an integer; inc1x > 0.

inc2x is the stride between the first elements of the sequences in array X.

Specified as: an integer; inc2x > 0.

y See On Return

inc1y is the stride between the elements within each sequence in output array Y.

Specified as: an integer; inc1y > 0.

inc2y is the stride between the first elements of each sequence in output array Y.

Specified as: an integer; inc2y > 0.

nx is the number of elements, Nₓ, in each sequence in array X.

Specified as: an integer; Nₓ > 0.

m is the number of sequences in array X to be correlated.

Specified as: an integer; m > 0.

ny is the number of elements, Nᵧ, in each sequence in array Y.
Specified as: an 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 75.

2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for \( \text{init} \neq 0 \) and \( \text{init} = 0 \).

3. 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 [24 on page 1364] and [102 on page 1369].
Only one invocation of this subroutine is needed:

1. You do not need to invoke the subroutine with $\text{init} \neq 0$. If you do, however, the subroutine performs error checking, exits back to the calling program, and no computation is performed.

2. With $\text{init} = 0$, the subroutine performs the calculation of the convolutions or correlations.

**Error conditions**

**Computational Errors**

None

**Input-Argument Errors**

1. $nx$, $ny$, or $m \leq 0$

2. $inc1x$, $inc2x$, $inc1y$, or $inc2y \leq 0$ (or incompatible)

**Examples**

**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:

```plaintext
REAL*4  X(0:49999), Y(0:49999)
REAL*8  AUX1, AUX2
```

Call Statement and Input:

```plaintext
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:
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
 7.0  11.0  20.0
 2.0   6.0  12.0
0.0   0.0  0.0
0.0   0.0  0.0
SACORF (Autocorrelation of One or More Sequences Using the Mixed-Radix Fourier Method)

Purpose

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:

1. Two invocations of this subroutine are necessary: one to prepare the working storage for the subroutine, and the other to perform the computations.

2. On certain processors, SIMD algorithms may be used if alignment requirements are met. For further details, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 32.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SACORF (init, x, inc1x, inc2x, y, inc1y, inc2y, nx, m, ny, aux1, auxx1, aux2, auxx2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>sacorf (init, x, inc1x, inc2x, y, inc1y, inc2y, nx, m, ny, aux1, auxx1, aux2, auxx2);</td>
</tr>
</tbody>
</table>

On Entry

init is a flag, where:

If init \( \neq 0 \), 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 \( \neq 0 \).

Specified as: an 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: an integer; inc1x \( > 0 \).

inc2x is the stride between the first elements of the sequences in array \( X \).

Specified as: an integer; inc2x \( > 0 \).

\( y \) See **On Return**

inc1y is the stride between the elements within each sequence in output array \( Y \).

Specified as: an integer; inc1y \( > 0 \).

inc2y is the stride between the first elements of each sequence in output array \( Y \).
Specified as: an integer; \( \text{inc2y} > 0 \).

\( nx \) is the number of elements, \( N_x \), in each sequence in array \( X \).
Specified as: an integer; \( N_x > 0 \).

\( m \) is the number of sequences in array \( X \) to be correlated.
Specified as: an integer; \( m > 0 \).

\( ny \) is the number of elements, \( N_y \), in each sequence in array \( Y \).
Specified as: an integer; \( N_y > 0 \).

\( aux1 \)

is the working storage for this subroutine, where:

If \( \text{init} \neq 0 \), the working storage is computed.

If \( \text{init} = 0 \), 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: an integer; \( naux1 > 21 \) (32-bit integer arguments) or 43 (64-bit integer arguments) and \( naux1 \geq \) (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas. For values between 21 (32-bit integer arguments) or 43 (64-bit integer arguments) 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 51.

\( aux2 \)

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: an 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, \( naux2 \geq \) (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 51.

On Return

\( y \) has the following meaning, where:

If \( \text{init} \neq 0 \), this argument is not used, and its contents remain unchanged.
If \( \text{init} = 0 \), 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 \( 1+(N_y-1)\text{inc1y}+(m-1)\text{inc2y} \), containing short-precision real numbers.

\( \text{aux1} \)

is the working storage for this subroutine, where:

If \( \text{init} \neq 0 \), it contains information ready to be passed in a subsequent invocation of this subroutine.

If \( \text{init} = 0 \), its contents are unchanged.

Returned as: the contents are not relevant.

Notes

1. \( \text{aux1} \) should not be used by the calling program between calls to this subroutine with \( \text{init} \neq 0 \) and \( \text{init} = 0 \). However, it can be reused after intervening calls to this subroutine with different arguments.

2. When using the ESSL SMP Libraries, for optimal performance, the number of threads specified should be the same for \( \text{init} \neq 0 \) and \( \text{init} = 0 \).

3. If you specify the same array for \( X \) and \( Y \), then \( \text{inc1x} \) and \( \text{inc1y} \) must be equal and \( \text{inc2x} \) and \( \text{inc2y} \) must be equal. In this case, output overwrites input.

4. If you specify different arrays for \( X \) and \( Y \), they must have no common elements; otherwise, results are unpredictable. See “Concepts” on page 75.

5. If \( ny \) is such that output outside the basic range is needed, the subroutine stores zeros. This range is: \( 0 \leq k \leq nx-1 \).

Formulas

Formula for Calculating the Length of the Fourier Transform

Before calculating the necessary sizes of \( \text{naux1} \) and \( \text{naux2} \), 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 1028 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 \( \text{NAUX1} \) and \( \text{NAUX2} \)

The required values of \( \text{naux1} \) and \( \text{naux2} \) depend on the value determined for \( n \) in [Formula for Calculating the Length of the Fourier Transform] and the argument \( m \).

\( \text{NAUX1} \) Formulas:

For 32-bit integer arguments:

If \( n \leq 16384 \), use \( \text{naux1} = 55000 \).

If \( n > 16384 \), use \( \text{naux1} = 40000+1.89n \).

For 64-bit integer arguments:

If \( n \leq 16384 \), use \( \text{naux1} = 75000 \).

If \( n > 16384 \), use \( \text{naux1} = 60000+1.89n \).

\( \text{NAUX2} \) Formulas:
If $n \leq 16384$, use $n_{aux2} = 50000$.
If $n > 16384$, use $n_{aux2} = 40000 + 1.64n$.

**Function**

The autocorrelations of the sequences in array $X$ are expressed as follows:

$$y_{ki} = \sum_{j=0}^{N_y-1} x_{ji}x_{j+k,i}$$

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 $n_{aux1}$ and $n_{aux2}$ is the same length used by the Fourier transform subroutines called by this subroutine. See references [24 on page 1364] and [102 on page 1369].

Two invocations of this subroutine are necessary:

1. With $init \neq 0$, 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, $n_{aux2} = 0$, and unable to allocate work area.

**Computational Errors**

None

**Input-Argument Errors**

1. $nx$, $ny$, or $m \leq 0$
2. $inc1x$, $inc2x$, $inc1y$, or $inc2y \leq 0$ (or incompatible)
3. The resulting correlation is too long.
4. The subroutine has not been initialized with the present arguments.
5. $n_{aux1} \leq 21$
6. $n_{aux1}$ 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 $n_{aux2} \neq 0$, and $n_{aux2}$ is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
Examples

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:

\[
\begin{align*}
\text{REAL*4} & \quad X(0:49999), \quad Y(0:49999) \\
\text{REAL*8} & \quad \text{AUX1(2959), AUX2(1)}
\end{align*}
\]

First, initialize AUX1 using the calling sequence shown below with \text{INIT} ≠ 0. Then use the same calling sequence with \text{INIT} = 0 to do the calculation.

Note: Because NAUX2 = 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

\[
\begin{align*}
\text{INIT} & \quad X \quad \text{INC1X} \quad \text{INC2X} \quad \text{Y} \quad \text{INC1Y} \quad \text{INC2Y} \quad \text{NX} \quad \text{NY} \quad \text{AUX1} \quad \text{NAUX1} \quad \text{AUX2} \quad \text{NAUX2} \\
\text{CALL SACORF(INIT, X, 1, 7, Y, 1, 7, 3, 7, \text{AUX1}, 2959, \text{AUX2}, 0)}
\end{align*}
\]

\[
\begin{align*}
\text{INIT} & \quad = \quad 1 & \quad (\text{for initialization}) \\
\text{INIT} & \quad = \quad 0 & \quad (\text{for computation})
\end{align*}
\]

X contains the following three sequences:

\[
\begin{align*}
1.0 & \quad 2.0 & \quad 3.0 \\
2.0 & \quad 1.0 & \quad 2.0 \\
3.0 & \quad 2.0 & \quad 1.0 \\
4.0 & \quad 3.0 & \quad 2.0 \\
4.0 & \quad 4.0 & \quad 3.0 \\
3.0 & \quad 4.0 & \quad 4.0 \\
2.0 & \quad 3.0 & \quad 4.0
\end{align*}
\]

Output:

Y contains the following three sequences:

\[
\begin{align*}
59.0 & \quad 59.0 & \quad 59.0 \\
54.0 & \quad 50.0 & \quad 44.0 \\
43.0 & \quad 39.0 & \quad 30.0 \\
29.0 & \quad 27.0 & \quad 24.0 \\
16.0 & \quad 18.0 & \quad 21.0 \\
7.0 & \quad 11.0 & \quad 20.0 \\
2.0 & \quad 6.0 & \quad 12.0
\end{align*}
\]

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:

\[
\begin{align*}
59.0 & \quad 59.0 & \quad 59.0 \\
54.0 & \quad 50.0 & \quad 44.0 \\
43.0 & \quad 39.0 & \quad 30.0 \\
29.0 & \quad 27.0 & \quad 24.0 \\
16.0 & \quad 18.0 & \quad 21.0 \\
7.0 & \quad 11.0 & \quad 20.0 \\
2.0 & \quad 6.0 & \quad 12.0 \\
0.0 & \quad 0.0 & \quad 0.0 \\
0.0 & \quad 0.0 & \quad 0.0
\end{align*}
\]
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 contains the related-computation subroutine descriptions.
**SPOLY and DPOLY (Polynomial Evaluation)**

**Purpose**

These subroutines evaluate a polynomial of degree $k$, using coefficient vector $u$, input vector $x$, and output vector $y$:

$$y_i = u_0 + u_1 x_i + u_2 x_i^2 + \ldots + u_k x_i^k \quad \text{for } i = 1,2,\ldots,n$$

where $u_i$, $x_i$, and $y_i$ are elements of $u$, $x$, and $y$, respectively.

**Table 215. Data Types**

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>$u$, $x$, $y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SPOLY</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DPOLY</td>
</tr>
</tbody>
</table>

**Syntax**

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SPOLY</th>
<th>DPOLY $(u, \text{inc}u, k, x, \text{inc}x, y, \text{incy}, n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>spoly</td>
<td>dpoly $(u, \text{inc}u, k, x, \text{inc}x, y, \text{incy}, n)$</td>
</tr>
</tbody>
</table>

**On Entry**

- $u$ is the coefficient vector $u$ of length $k+1$. It contains elements $u_0, u_1, u_2, u_3, \ldots, u_k$, which are stored in this order. Specified as: a one-dimensional array of (at least) length $1+k|\text{inc}u|$, containing numbers of the data type indicated in Table 215.

- $\text{inc}u$ is the stride for vector $u$.
  Specified as: an integer. It can have any value.

- $k$ is the degree $k$ of the polynomial.
  Specified as: an integer; $k \geq 0$.

- $x$ is the input vector $x$ of length $n$. Specified as: a one-dimensional array of (at least) length $1+(n-1)|\text{inc}x|$, containing numbers of the data type indicated in Table 215.

- $\text{inc}x$ is the stride for vector $x$.
  Specified as: an integer. It can have any value.

- $y$ See On Return

- $\text{incy}$ is the stride for the output vector $y$. Specified as: an integer. It can have any value.

- $n$ is the number of elements in input vector $x$ and the number of resulting elements in output vector $y$.
  Specified as: an 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 215 on page 1183.

Notes

Vectors \(u\), \(x\), and \(y\) must have no common elements; otherwise, results are unpredictable. See “Concepts” on page 75.

Function

The evaluation of the polynomial:

\[y_i = u_0 + u_1x_i + u_2x_i^2 + \ldots + u_kx_i^k\quad \text{for} \ i = 1, 2, \ldots, n\]

is expressed as follows:

\[y_i = u_0 + x_i (u_1 + x_i (u_2 + \ldots + x_i (u_{k-1} + x_iu_k) \ldots)) \quad \text{for} \ i = 1, 2, \ldots, n\]

See reference [98 on page 1369] 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. \(n < 0\)

Examples

Example 1

This example shows a polynomial evaluation with the degree, \(K\), equal to 0.

Call Statement and Input:

\[
\begin{align*}
U & \quad \text{INCU} \quad K \quad X \quad \text{INCX} \quad Y \quad \text{INCY} \quad N \\
\text{CALL} & \quad \text{SPOLY}(U, \text{INCU}, 0, X, \text{INCX}, Y, 1, 3)
\end{align*}
\]

\[
\begin{align*}
U & \quad = \quad (4.0) \\
\text{INCU} & \quad = \quad \text{(not relevant)} \\
X & \quad = \quad \text{(not relevant)} \\
\text{INCX} & \quad = \quad \text{(not relevant)}
\end{align*}
\]

Output:

\[
\begin{align*}
Y & \quad = \quad (4.0, 4.0, 4.0)
\end{align*}
\]

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
 U , -1 , 3 , X , 1 , Y , 1 , 3 
```

```
CALL SPOLY( U , X , Y , N )
```

```
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
 U , 1 , 3 , X , 0 , Y , 1 , 3 
```

```
CALL SPOLY( U , X , Y , N )
```

```
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
 U , 1 , 3 , X , 2 , Y , -2 , 3 
```

```
CALL SPOLY( U , X , Y , N )
```

```
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)

Purpose

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 216. Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SIZC</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DIZC</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SIZC</td>
</tr>
<tr>
<td>C and C++</td>
<td>sizec</td>
</tr>
</tbody>
</table>

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 216.

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: an integer. It can have any value.

n is the number of elements in vector x. Specified as: an integer; n > 1.

i is the number of the zero crossing to be identified.

Specified as: an integer; i > 0.

ky See On Return

On Return

ky is the integer vector ky of length 2, containing elements ky₁ and ky₂, where:

If the i-th zero crossing is found:

• ky₁ = j, where j is the position of the element xₖ 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.
• ky₂ = i

If the i-th zero crossing is not found:

• ky₁ = 0
• ky₂ = the total number of zero crossings encountered in the scan.

Returned as: an array of (at least) length 2, containing integers.
Notes

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 ky1 as the position of the i-th zero crossing, and i is returned in ky2.

If idrx ≥ 0:

TRUE = (x_{j-1} < 0 and x_j ≥ 0) or (x_{j+1} ≥ 0 and x_j < 0) for j = 2, n

If idrx < 0:

TRUE = (x_{j-1} < 0 and x_j ≥ 0) or (x_{j+1} ≥ 0 and x_j < 0) for j = n-1, 1

If the position of the i-th zero crossing is not found, 0 is returned in y1 and the number of zero crossings encountered in the scan is returned in y2.

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 ≤ 1

2. i ≤ 0

Examples

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:

```c
CALL SIZC(X , 1 , 12 , 5 , KY)
```

```plaintext
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:
**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)

Purpose

These subroutines implement the first-order time-varying recursive equation, using initial value $s$, target vectors $u$ and $x$, and output vector $y$.

Table 217. Data Types

<table>
<thead>
<tr>
<th>$s$, $u$, $x$, $y$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>STREC</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DTREC</td>
</tr>
</tbody>
</table>

Syntax

Fortran

```
CALL STREC | DTREC (s, u, incu, x, incx, y, incy, n , iopt)
```

C and C++

```
strec | dtrc (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 217

$u$ 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 217

$incu$ is the stride for target vector $u$.

Specified as: an 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 217

$incx$ is the stride for target vector $x$. Specified as: an integer. It can have any value.

$y$ See On Return

$incy$ is the stride for output vector $y$. Specified as: an 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: an integer; $n \geq 0$.

$iopt$ this argument has no effect on the performance of the computation, but still must be Specified as: an 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 217.
Notes

Vectors \( u, x, \) and \( y \) must have no common elements; otherwise, results are unpredictable. See "Concepts" on page 75.

Function

The first-order time-varying recursive equation is expressed as follows:

\[
\begin{align*}
y_1 &= s + u_1 x_1 \\
y_2 &= u_2 y_1 + u_1 x_2 \\
& \quad \vdots \\
y_i &= u_i y_{i-1} + u_1 x_i \text{ for } i = 3, 4, \ldots, n
\end{align*}
\]

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 ≠ 0 or 1

Examples

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:

```
CALL STREC( 1.0 , U , 1 , X , 1 , Y , 1 , 8 , 0 )
```

\[
\begin{align*}
U &= (1.0, 2.0, 3.0, 3.0, 2.0, 1.0, 1.0, 2.0) \\
X &= (3.0, 2.0, 1.0, 1.0, 2.0, 3.0, 3.0, 2.0)
\end{align*}
\]

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:

```
CALL STREC( 1.0 , U , 2 , X , 1 , Y , 1 , 4 , 0 )
```

\[
\begin{align*}
U &= (1.0, . , 3.0, . , 2.0, . , 1.0, .) \\
X &= (3.0, 2.0, 1.0, 1.0, 2.0, 3.0, 3.0, 2.0)
\end{align*}
\]
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:

\[
\begin{align*}
\text{CALL STREC(} & \text{1.0 , } U , 1 , X , 2 , Y , -1 , 4 , 1 \text{)} \\
U & = (1.0, 2.0, 3.0, 3.0, 2.0, 1.0, 1.0, 2.0) \\
X & = (3.0, . , 1.0, . , 2.0, . , 3.0)
\end{align*}
\]

Output:

\[
Y = (90.0, 29.0, 9.0, 4.0)
\]
SQINT and DQINT (Quadratic Interpolation)

Purpose

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 \( \Omega \), and sample reflection times in vector \( t \). The result is returned in vector \( y \).

Table 218. Data Types

<table>
<thead>
<tr>
<th>( x, s, g, \Omega, t, y )</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SQINT</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DQINT</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SQINT</th>
<th>DQINT (( s, g, \Omega, x, incx, n, t, incl, y, incy, m ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>sqint</td>
<td>dqint (( s, g, \Omega, x, incx, n, t, incl, y, incy, m ));</td>
</tr>
</tbody>
</table>

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 218

\( g \) is the scalar \( g \), containing the sample interval.

Specified as: a number of the data type indicated in Table 218; \( g > 0.0 \).

\( \Omega \)

is the output scaling parameter \( \Omega \).

Specified as: a number of the data type indicated in Table 218

\( x \) is the vector \( x \) of length \( n \), containing the trace data.

Specified as: a one-dimensional array of (at least) length \( 1+(n-1)|\text{incx}| \), containing numbers of the data type indicated in Table 218

\( \text{incx} \)

is the stride for vector \( x \).

Specified as: an integer; \( \text{incx} > 0 \) or \( \text{incx} < 0 \).

\( n \) is the number of elements in vector \( x \).

Specified as: an 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 \( 1+(m-1)|\text{inct}| \), containing numbers of the data type indicated in Table 218

\( \text{inct} \)

is the stride for vector \( t \).

Specified as: an integer; \( \text{inct} > 0 \) or \( \text{inct} < 0 \).

\( y \)

See On Return

\( \text{incy} \)

is the stride for output vector \( y \).
Specified as: an 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: an 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 218 on page 1192.

**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.
\( \Omega \) 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:

\[
\begin{align*}
trace_i &= 3x_i - 3x_{i+1} + x_{i+2} \\
trace_{i+1} &= x_i \quad \text{for } i = 1, 2, ..., n \\
so &= s + 2.0 \\
w_i &= so + t_i / g \quad \text{for } i = 1, 2, ..., m \\
f_i &= \text{fraction part of } w_i \\
k_{i+1} &= \text{integer part of } w_i
\end{align*}
\]

**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 ERKSET 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 68.

**Input-Argument Errors**

1. $n < 3$
2. $m < 0$
3. $g \leq 0$
4. $incx = 0$
5. $inct = 0$
6. $incy = 0$

**Examples**

**Example 1**

This example shows a quadratic interpolation, using vectors with strides of 1.

Call Statement and Input:

```
CALL SQINT( 2.0, 1.0, 1.0, X, 1, 8, T, 1, Y, 1, 4 )
```

Output:

```
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)
```

**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:

```
CALL SQINT( 2.0, 1.0, 1.0, X, -1, 8, T, -1, Y, 1, 4 )
```

Output:

```
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)
```

**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:

```
CALL SQINT( 2.0, 1.0, 1.0, X, -2, 8, T, -1, Y, 2, 4 )
```

Output:

```
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:

\[
\begin{align*}
S & \quad G \quad OMEGA \quad X \quad INCX \quad N \quad T \quad INCT \quad Y \quad INCY \quad M \\
| & | & | & | & | & | & | & |
\end{align*}
\]

CALL SQINT( 3.0, 10.0, 1.0, X, 1, 8, T, 2, Y, 3, 4 )

\[
\begin{align*}
X & = (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0) \\
T & = (1.5, . , 2.5, . , 3.5, . , 4.5)
\end{align*}
\]

Output:
\[
\begin{align*}
Y & = (8.3, . , 8.5, . , 8.7, . , 8.9)
\end{align*}
\]
SWLEV, DWLEV, CWLEV, and ZWLEV (Wiener-Levinson Filter Coefficients)

Purpose

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 219. Data Types

<table>
<thead>
<tr>
<th>$x, u, y$</th>
<th>$aux$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Long-precision real</td>
<td>SWLEV</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DWLEV</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Long-precision complex</td>
<td>CWLEV</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision complex</td>
<td>ZWLEV</td>
</tr>
</tbody>
</table>

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);

**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 $1+(n-1)|\text{incx}|$, containing numbers of the data type indicated in Table 219

$\text{incx}$

is the stride for vector $x$.

Specified as: an integer; $\text{incx} > 0$.

$u$ 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 $1+(n-1)|\text{incu}|$, containing numbers of the data type indicated in Table 219

$\text{incu}$

is the stride for vector $u$.

Specified as: an integer. It can have any value.

$y$ See **On Return**

$\text{incy}$

is the stride for vector $y$. 
Specified as: an integer; \( \text{incy} > 0 \) or \( \text{incy} < 0 \).

\( n \) is the number of elements in vectors \( x \), \( u \), and \( y \).

Specified as: an integer; \( n \geq 0 \).

\( \text{aux} \) has the following meaning:

If \( \text{naux} = 0 \) and error 2015 is unrecoverable, \( \text{aux} \) is ignored.

Otherwise, it is the storage work area used by these subroutines.

Specified as: an area of storage of length \( \text{naux} \), containing numbers of the data type indicated in [Table 219 on page 1196].

\( \text{naux} \) is the size of the work area specified by \( \text{aux} \)—that is, the number of elements in \( \text{aux} \).

Specified as: an integer, where:

If \( \text{naux} = 0 \) 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, \( \text{naux} \geq 3n \).

You cannot use dynamic allocation if you need the information returned in \( \text{AUX(1)} \).

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 \( 1+(n-1)|\text{incy}| \), containing numbers of the data type indicated in [Table 219 on page 1196].

\( \text{aux} \) is the storage work area used by these subroutines, where if \( \text{naux} \neq 0 \):

If \( \text{AUX(1)} = 0.0 \), the input Toeplitz matrix is positive definite or negative definite.

If \( \text{AUX(1)} > 0.0 \), the input Toeplitz matrix is indefinite (that is, it is not positive definite and it is not negative definite). The value returned in \( \text{AUX(1)} \) is the order of the first submatrix of \( A \) that is indefinite. The subroutine continues processing. See reference [73 on page 1367] for information about under what circumstances your solution vector \( y \) would be valid.

All other values in \( \text{aux} \) are overwritten and are not significant.

Returned as: an area of storage of length \( \text{naux} \), containing numbers of the data type indicated in [Table 219 on page 1196] where \( \text{AUX(1)} \geq 0.0 \).

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 91.

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 92.
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 51.

**Function**

The computation of the coefficients of an \( n \)-point Wiener-Levinson filter in vector \( y \) is expressed as solving the following system:

\[
Ay = u
\]

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 \( x \).
- \( n \) 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 \([73 on page 1367], [35 on page 1365]\), 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. \( \text{incx} \neq 0 \)
3. \( \text{incy} = 0 \)
4. 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.

**Examples**

**Example 1**

This example shows how to compute filter coefficients in vector \( y \) by solving the system \( Ay = u \). Matrix \( A \) is:

\[
\begin{bmatrix}
50.0 & -8.0 & 7.0 & -5.0 \\
-8.0 & 50.0 & -8.0 & 7.0
\end{bmatrix}
\]
This input Toeplitz matrix is positive definite, as indicated by the zero value in \( \text{AUX}(1) \) on output.

Call Statement and Input:

\[
\begin{align*}
\text{X} & = (50.0, -8.0, 7.0, -5.0) \\
\text{U} & = (40.0, -10.0, 30.0, 20.0) \\
\text{AUX} & = (\text{not relevant})
\end{align*}
\]

Output:

\[
\begin{align*}
\text{Y} & = (0.7667, -0.0663, 0.5745, 0.5778) \\
\text{AUX} & = (0.0, \ldots, \ldots, \ldots, \ldots, \ldots)
\end{align*}
\]

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 \( \text{AUX}(1) \) on output.

Call Statement and Input:

\[
\begin{align*}
\text{X} & = (10.0, -8.0, 7.0, -5.0) \\
\text{U} & = (20.0, -10.0, 30.0, 20.0) \\
\text{AUX} & = (\text{not relevant})
\end{align*}
\]

Output:

\[
\begin{align*}
\text{Y} & = (5.1111, 5.5555, 12.2222, 10.4444) \\
\text{AUX} & = (0.0, \ldots, \ldots, \ldots, \ldots, \ldots)
\end{align*}
\]

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:

\[
\begin{align*}
\text{X} & = (10.0, \ldots, -8.0, \ldots, 7.0, \ldots, -5.0) \\
\text{U} & = (20.0, \ldots, 30.0, \ldots, -10.0, \ldots, 40.0) \\
\text{AUX} & = (\text{not relevant})
\end{align*}
\]

Output:
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 \( \text{AUX}(1) \) on output.

Call Statement and Input:

```
CALL ZWLEV( X , 1 , U , 1 , Y , 1 , 4 , AUX , 12 )
```

\( X = ((10.0, 0.0), (2.0, -3.0), (-3.0, 1.0), (1.0, 1.0)) \)
\( U = ((8.0, 3.0), (21.0, -5.0), (67.0, -13.0), (72.0, 11.0)) \)
\( \text{AUX} = (\text{not relevant}) \)

Output:

\( Y = ((1.0, 0.0), (3.0, 0.0), (5.0, 0.0), (7.0, 0.0)) \)
\( \text{AUX} = ((0.0, 0.0), . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . ) \)

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 \( \text{AUX}(1) \) on output.

Call Statement and Input:

```
CALL ZWLEV( X , 2 , U , -2 , Y , 1 , 4 , AUX , 12 )
```

\( X = ((10.0, 0.0), . . , (2.0, -3.0), . . , (-3.0, 1.0), . . , (1.0, 1.0)) \)
\( U = ((72.0, 11.0), . . , (67.0, -13.0), . . , (21.0, -5.0), . . , (8.0, 3.0), . . ) \)
\( \text{AUX} = (\text{not relevant}) \)

Output:

\( Y = ((1.0, 0.0), (3.0, 0.0), (5.0, 0.0), (7.0, 0.0)) \)
\( \text{AUX} = ((0.0, 0.0), . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . ) \)
Chapter 13. Sorting and Searching

The sorting and searching subroutines are described here.

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. The sorting subroutines perform sorts with or without index designations. The searching subroutines perform either a binary or sequential search.

Table 220. List of Sorting and Searching Subroutines

<table>
<thead>
<tr>
<th>Integer Subroutine</th>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISORT</td>
<td>SSORT</td>
<td>DSORT</td>
<td>“ISORT, SSORT, and DSORT (Sort the Elements of a Sequence)” on page 1204</td>
</tr>
<tr>
<td>ISORTX</td>
<td>SSORTX</td>
<td>DSORTX</td>
<td>“ISORTX, SSORTX, and DSORTX (Sort the Elements of a Sequence and Note the Original Element Positions)” on page 1206</td>
</tr>
<tr>
<td>ISORTS</td>
<td>SSORTS</td>
<td>DSORTS</td>
<td>“ISORTS, SSORTS, and DSORTS (Sort the Elements of a Sequence Using a Stable Sort and Note the Original Element Positions)” on page 1209</td>
</tr>
<tr>
<td>IBSRCH</td>
<td>SBSRCH</td>
<td>DBSRCH</td>
<td>“IBSRCH, SBSRCH, and DBSRCH (Binary Search for Elements of a Sequence X in a Sorted Sequence Y)” on page 1213</td>
</tr>
<tr>
<td>ISSRCH</td>
<td>SSSRCH</td>
<td>DSSRCH</td>
<td>“ISSRCH, SSSRCH, and DSSRCH (Sequential Search for Elements of a Sequence X in the Sequence Y)” on page 1217</td>
</tr>
</tbody>
</table>

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 78.

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.
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 \textit{i}opt argument.
Sorting and Searching Subroutines

This contains the sorting and searching subroutine descriptions.
ISORT, SSORT, and DSORT (Sort the Elements of a Sequence)

Purpose

These subroutines sort the elements of sequence $x$.

Table 221. Data Types

<table>
<thead>
<tr>
<th>$x$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>ISORT</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>SSORT</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DSORT</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL ISORT</th>
<th>SSORT</th>
<th>DSORT ($x$, $incx$, $n$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>isort</td>
<td>ssort</td>
<td>dsort ($x$, $incx$, $n$);</td>
</tr>
</tbody>
</table>

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 221.

$incx$ 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: an integer. It can have any value.

$n$ is the number of elements in sequence $x$. Specified as: an 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 221.

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 [89 on page 1368].

Error conditions

Resource Errors

Unable to allocate internal work area.
Computational Errors
None

Input-Argument Errors
n < 0

Examples

Example 1
This example shows a sequence x with a positive stride.

Call Statement and Input:

\[
\begin{align*}
X & \quad \text{INCX} \quad N \\
\text{CALL ISORT}(X, 2, 5)
\end{align*}
\]

\[
X = (2, \ldots, -1, \ldots, 5, \ldots, 4, \ldots, -2)
\]

Output:

\[
X = (-2, \ldots, -1, \ldots, 2, \ldots, 4, \ldots, 5)
\]

Example 2
This example shows a sequence x with a negative stride.

Call Statement and Input:

\[
\begin{align*}
X & \quad \text{INCX} \quad N \\
\text{CALL ISORT}(X, -1, 5)
\end{align*}
\]

\[
X = (2, -1, 5, 4, -2)
\]

Output:

\[
X = (5, 4, 2, -1, -2)
\]
ISORTX, SSORTX, and DSORTX (Sort the Elements of a Sequence and Note the Original Element Positions)

Purpose

These subroutines sort the elements of sequence \( x \). The original positions of the elements in sequence \( x \) are returned in the indices array, \( \text{INDX} \). 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 222. Data Types

<table>
<thead>
<tr>
<th>( x )</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>ISORTX</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>SSORTX</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DSORTX</td>
</tr>
</tbody>
</table>

Syntax

**Fortran**

```
CALL ISORTX | SSORTX | DSORTX (x, incx, n, indx)
```

**C and C++**

```
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 \( 1+(n-1)|\text{incx}| \) elements, containing numbers of the data type indicated in Table 222.

\( \text{incx} \)

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: an integer. It can have any value.

\( n \) is the number of elements in sequence \( x \). Specified as: an integer; \( n \geq 0 \).

\( \text{indx} \)

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 222.

\( \text{indx} \)

is the array, referred to as \( \text{INDX} \), 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 \( \text{INDX} \) are not reversed. See “Function” on page 1207.
Returned as: a one-dimensional array of length $n$, containing integers; $1 \leq (\text{INDX elements}) \leq n$.

**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:

$$x_1 \leq x_2 \leq x_3 \leq \ldots \leq x_n$$

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 $x$, 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 $\text{INDX}$ 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 $\text{INDX}$ are defined as follows:

$$\text{INDX}(k) = j \quad \text{for} \quad j = 1, n \quad \text{and} \quad k = 1, n$$

where $xx_k = x_j$.

To understand $\text{INDX}$ 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 $\text{INDX}$ is not affected by stride. The sequence elements of $x$ are assumed to be stored in your input array as follows:

$$X = (x_n, x_{n-1}, \ldots, x_1)$$

The sequence elements of $x$ 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 $\text{INDX}$ is calculated, if $xx_1 = x_{n-1}$, then $\text{INDX}(1) = n-1$.

If $n$ is 0, no computation is performed. See reference [89 on page 1368].

**Error conditions**

**Resource Errors**

Unable to allocate internal work area.

**Computational Errors**

None

**Input-Argument Errors**

$n < 0$

**Examples**

**Example 1**
This example shows how to sort a sequence x into ascending order by specifying a positive stride.

Call Statement and Input:

\[
\begin{align*}
&X \quad \text{INCX} \quad N \quad \text{INDX} \\
&\text{CALL ISORTX( X, 2, 5, INDX )} \\
&X = (2, . , -1, . , 5, . , 1, . , -2)
\end{align*}
\]

Output:

\[
\begin{align*}
&X = (-2, . , -1, . , 1, . , 2, . , 5) \\
&\text{INDX} = (5, 2, 4, 1, 3)
\end{align*}
\]

**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:

\[
\text{INDX} = (indx_1, indx_2, indx_3, indx_4, indx_5) = (1, 4, 2, 5, 3)
\]

For example, because output sequence element xx_4 = 2 is input sequence element x_5, then INDX(4) = 5.

Call Statement and Input:

\[
\begin{align*}
&X \quad \text{INCX} \quad N \quad \text{INDX} \\
&\text{CALL ISORTX( X, -1, 5, INDX )} \\
&X = (2, -1, 5, 1, -2)
\end{align*}
\]

Output:

\[
\begin{align*}
&X = (5, 2, 1, -1, -2) \\
&\text{INDX} = (1, 4, 2, 5, 3)
\end{align*}
\]
ISORTS, SSORTS, and DSORTS (Sort the Elements of a Sequence Using a Stable Sort and Note the Original Element Positions)

Purpose

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 223. Data Types

<table>
<thead>
<tr>
<th>x, work</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>ISORTS</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>SSORTS</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DSORTS</td>
</tr>
</tbody>
</table>

Syntax

Fortran

CALL ISORTS | SSORTS | DSORTS (x, incx, n, indx, work, lwork)

C and C++

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 223.

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: an integer. It can have any value.

n is the number of elements in sequence x. Specified as: an 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 223.

lwork

is the size of the work area specified by work— that is, the number of elements in work.

Specified as: an 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 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 223 on page 1209.

\( \text{indx} \)

is the array, referred to as \( \text{INDX} \), 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 \( \text{INDX} \) are not reversed. See "Function."

Returned as: a one-dimensional array of length \( n \), containing integers; \( 1 \leq (\text{INDX} \text{ elements}) \leq n \).

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:

\[ x_1 \leq x_2 \leq x_3 \leq \ldots \leq x_n \]

By specifying a negative stride for \( x \), the elements of input sequence \( x \) are assumed to be reversed in the array, \( (x_{n-1}, x_{n-2}, \ldots, x_1) \), thus producing a sort into descending order within the array.

In addition, the \( \text{INDX} \) 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 \( \text{INDX} \) are defined as follows:

\[ \text{INDX}(k) = j \quad \text{for} \quad j = 1, n \quad \text{and} \quad k = 1, n \]

where \( xx_k = x_j \)

To understand \( \text{INDX} \) 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 \( \text{INDX} \) is not affected by stride. The sequence elements of \( x \) are assumed to be stored in your input array as follows:

\[ X = (x_{n-1}, x_{n-2}, \ldots, x_1) \]

The sequence elements of \( x \) are stored in your output array by ESSL as follows:

\[ X = (xx_{n-1}, xx_{n-2}, \ldots, xx_1) \]

where the elements \( xx_k \) are the elements \( x_j \) sorted into descending order in \( X \). As an example of how \( \text{INDX} \) is calculated, if \( xx_1 = x_{n-1} \), then \( \text{INDX}(1) = n-1 \).

If \( n \) is 0, no computation is performed. See references [36 on page 1365] and [89 on page 1368].
Error conditions

Resource Errors
Unable to allocate internal work area.

Computational Errors
None

Input-Argument Errors
n < 0

Examples

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 \( \text{INDX}(2) = 2 \) and \( \text{INDX}(3) = 4 \).

Call Statement and Input:

\[
\begin{align*}
X & \quad \text{INCX} \quad N \quad \text{INDX} \quad \text{WORK} \quad \text{LWORK} \\
\text{CALL} \ ISORTS( & \quad X, \quad 2, \quad 5, \quad \text{INDX}, \quad \text{WORK}, \quad 5 )
\end{align*}
\]

\( X = (2, \ . \ , -1, \ . \ , 5, \ . \ , -1, \ . \ , -2) \)

Output:

\( X = (-2, \ . \ , -1, \ . \ , -1, \ . \ , 2, \ . \ , 5) \)

\( \text{INDX} = (5, \ 2, \ 4, \ 1, \ 3) \)

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, \( \text{INDX} \) is defined as follows:

\( \text{INDX} = (indx_1, \ indx_2, \ indx_3, \ indx_4, \ indx_5) = (1, 2, 4, 5, 3) \)

For example, because output sequence element \( xx_4 = 2 \) is input sequence element \( x_5 \), then \( \text{INDX}(4) = 5 \). Also, because this is a stable sort, the -1 elements remain in the same relative order in the output sequence, indicated by \( \text{INDX}(2) = 2 \) and \( \text{INDX}(3) = 4 \).

Call Statement and Input:

\[
\begin{align*}
X & \quad \text{INCX} \quad N \quad \text{INDX} \quad \text{WORK} \quad \text{LWORK} \\
\text{CALL} \ ISORTS( & \quad X, \quad -1, \quad 5, \quad \text{INDX}, \quad \text{WORK}, \quad 5 )
\end{align*}
\]

\( X = (2, \ -1, \ 5, \ -1, \ -2) \)

Output:
\[ X = (5, 2, -1, -1, -2) \]
\[ \text{INDX} = (1, 2, 4, 5, 3) \]
IBSRCH, SBSRCH, and DBSRCH (Binary Search for Elements of a Sequence X in a Sorted Sequence Y)

Purpose

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 \( \text{INDX} \), and, optionally, return codes indicating whether the exact elements were found are returned in array \( \text{RC} \).

Table 224. Data Types

<table>
<thead>
<tr>
<th>( x, y )</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>IBSRCH</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>SBSRCH</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DBSRCH</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL IBSRCH</th>
<th>SBSRCH</th>
<th>DBSRCH (( x, \text{incx}, n, y, \text{incy}, m, \text{INDX}, \text{RC}, \text{iOPT} ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>ibsrch</td>
<td>sbsrch</td>
<td>dbsrch (( x, \text{incx}, n, y, \text{incy}, m, \text{INDX}, \text{RC}, \text{iOPT} ))</td>
</tr>
</tbody>
</table>

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 224. It must have at least \( 1+(n-1) \mid \text{incx} \mid \) elements.

\( \text{incx} \)

is the stride for sequence \( x \).

Specified as: an integer. It can have any value.

\( n \)

is the number of elements in sequence \( x \) and arrays \( \text{INDX} \) and \( \text{RC} \).

Specified as: an 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 \( 1+(m-1) \mid \text{incy} \mid \), containing numbers of the data type indicated in Table 224.

\( \text{incy} \)

is the stride for sequence \( y \).

Specified as: an integer. It can have any value.

\( m \)

is the number of elements in sequence \( y \). Specified as: an integer; \( m \geq 0 \).

\( \text{INDX} \)

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: an 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 integers; 1 ≤ (INDX elements) ≤ m+1.

cr  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 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 1204.

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 \(y_j = x_i, j = 1, m\), \(\text{INDX}(i) = \min(j)\)

if all \(y_j = x_i, j = 1, n\), \(\text{INDX}(i) = n+1\)

And for \(i = 1, n\)

if \(x_i = y_{\text{INDX}(i)}\), \(\text{RC}(i) = 0\)

if \(x_i \neq y_{\text{INDX}(i)}\), \(\text{RC}(i) = 1\)

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
- \(\text{INDX}\) is the array of length \(n\) of indices
- \(\text{RC}\) is the array of length \(n\) of return codes

See reference [89 on page 1368]. If \(n\) is 0, no search is performed. If \(m\) is 0, then:

\(\text{INDX}(i) = 1\) and \(\text{RC}(i) = 1\) 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 1204.

Error conditions

Computational Errors

None

Input-Argument Errors

1. \(n < 0\)
2. \(m < 0\)
3. \(\text{iopt} \neq 0\) or \(1\)

Examples

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 IBRSCH( 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:

\(\text{INDX} = (1, 11, 3, 2, 8)\)

\(\text{RC} = (1, 1, 0, 0, 1)\)

Example 2

This example shows the same calling sequence as in Example 1, except that it includes the \(\text{IOPT}\) argument, specified as 1. This is equivalent to using the calling sequence in Example 1 and gives the same results.
Call Statement and Input:

\[
\begin{align*}
    X & \quad \text{INCX} & \quad N & \quad Y & \quad \text{INCY} & \quad M & \quad \text{INDX} & \quad RC & \quad \text{IOPT} \\
    \text{CALL IBSRCH(} & \quad X & \quad 2 & , & \quad 5 & , & \quad Y & \quad 1 & , & \quad 10 & , & \quad \text{INDX} & , & \quad RC & , & \quad 1 \quad )
\end{align*}
\]

**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 \( \text{IOPT} \) is specified as 0.

Call Statement and Input:

\[
\begin{align*}
    X & \quad \text{INCX} & \quad N & \quad Y & \quad \text{INCY} & \quad M & \quad \text{INDX} & \quad RC & \quad \text{IOPT} \\
    \text{CALL IBSRCH(} & \quad X & \quad -2 & , & \quad 5 & , & \quad Y & \quad 1 & , & \quad 10 & , & \quad \text{INDX} & , & \quad RC & , & \quad 0 \quad )
\end{align*}
\]

\[
\begin{align*}
    X & = (-3, \ldots, 125, \ldots, 30, \ldots, 20, \ldots, 70) \\
    Y & = (10, 20, 30, 30, 40, 50, 60, 80, 90, 100)
\end{align*}
\]

Output:

\[
\begin{align*}
    \text{INDX} & = (8, 2, 3, 11, 1) \\
    \text{RC} & = \text{(not relevant)}
\end{align*}
\]

**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 \( \text{IOPT} \) is specified as 0.

Call Statement and Input:

\[
\begin{align*}
    X & \quad \text{INCX} & \quad N & \quad Y & \quad \text{INCY} & \quad M & \quad \text{INDX} & \quad RC & \quad \text{IOPT} \\
    \text{CALL IBSRCH(} & \quad X & \quad 2 & , & \quad 5 & , & \quad Y & \quad -1 & , & \quad 10 & , & \quad \text{INDX} & , & \quad RC & , & \quad 0 \quad )
\end{align*}
\]

\[
\begin{align*}
    X & = (-3, \ldots, 125, \ldots, 30, \ldots, 20, \ldots, 70) \\
    Y & = (100, 90, 80, 60, 50, 40, 30, 30, 20, 10) \\
    \text{RC} & = \text{(not relevant)}
\end{align*}
\]

Output:

\[
\begin{align*}
    \text{INDX} & = (1, 11, 3, 2, 8)
\end{align*}
\]
ISSRCH, SSSRCH, and DSSRCH (Sequential Search for Elements of a Sequence X in the Sequence Y)

Purpose

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 225. Data Types

<table>
<thead>
<tr>
<th>x, y</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>ISSRCH</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>SSSRCH</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DSSRCH</td>
</tr>
</tbody>
</table>

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);

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 225.

incx is the stride for sequence x.

Specified as: an integer. It can have any value.

n is the number of elements in sequence x and array INDX.

Specified as: an integer; n ≥ 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 225.

incy is the stride for sequence y.

Specified as: an integer. It can have any value.

m is the number of elements in sequence y. Specified as: an integer; m ≥ 0.

idir indicates the search direction, where:

If idir ≥ 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: an integer. It can have any value.

**indx**

See [On Return](#).

**On Return**

**indx**

is the array, referred to as \( \text{INDX} \), containing the \( n \) indices that indicate the positions of the elements of sequence \( x \) in sequence \( y \), where:

- If \( idir \geq 0 \), the first occurrence of the element found in sequence \( y \) is indicated in array \( \text{INDX} \).
- If \( idir < 0 \), the last occurrence of the element found in sequence \( y \) is indicated in array \( \text{INDX} \).

In all cases, if no match is found, 0 is indicated in array \( \text{INDX} \).

Returned as: a one-dimensional array of length \( n \), containing integers; \( 0 \leq (\text{INDX} \text{ elements}) \leq m \).

**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 \( \text{INDX} \), indicating the positions of the elements of sequence \( x \) in sequence \( y \). The positions indicated in array \( \text{INDX} \) 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 \( \text{INDX} \) for that position.

The results returned in array \( \text{INDX} \) are expressed as follows:

For \( i = 1, n \)

for all \( y_j = x_i \), \( j = 1, m \)

\[
\text{INDX}(i) = \min(j), \text{ if } idir \geq 0
\]

\[
\text{INDX}(i) = \max(j), \text{ if } idir < 0
\]

if all \( y_j \neq x_i \), \( j = 1, m \)

\[
\text{INDX}(i) = 0
\]

where:

- \( x \) is a sequence of length \( n \), containing the search elements.
- \( y \) is a sequence of length \( m \) to be searched.
- \( \text{INDX} \) is the array of length \( n \) of indices.

See reference [89 on page 1368](#). 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. \( m < 0 \)

Examples

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:

\[
X \quad \text{INCX} \quad N \quad Y \quad \text{INCY} \quad M \quad \text{IDIR} \quad \text{INDX}
\]

\[
\text{CALL ISSRCH}( X, 1, 3, Y, 2, 8, 1, \text{INDX})
\]

\[
X = (0, 12, 3)
\]

\[
Y = (0, 8, 12, 0, 1, 4, 0, 2)
\]

Output:

\[
\text{INDX} = (1, 3, 0)
\]

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:

\[
X \quad \text{INCX} \quad N \quad Y \quad \text{INCY} \quad M \quad \text{IDIR} \quad \text{INDX}
\]

\[
\text{CALL ISSRCH}( X, 2, 3, Y, 2, 8, -1, \text{INDX})
\]

\[
X = (0, 12, 3)
\]

\[
Y = (0, 8, 12, 0, 1, 4, 0, 2)
\]

Output:

\[
\text{INDX} = (7, 3, 0)
\]

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:

\[
X \quad \text{INCX} \quad N \quad Y \quad \text{INCY} \quad M \quad \text{IDIR} \quad \text{INDX}
\]

\[
\text{CALL ISSRCH}( X, -1, 3, Y, -2, 8, 1, \text{INDX})
\]

\[
X = (0, 12, 3)
\]

\[
Y = (0, 8, 12, 0, 1, 4, 0, 2)
\]

Output:

\[
\text{INDX} = (0, 6, 2)
\]

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:

\[
\begin{array}{cccccccc}
X & INCX & N & Y & INCY & M & IDIR & INDX \\
\hline
\end{array}
\]

CALL ISSRCH( X, -2, 3, Y, -1, 8, -1, INDX )

\[
\begin{align*}
X &= (0, . , 12, . , 3) \\
Y &= (0, 8, 12, 0, 1, 4, 0, 2)
\end{align*}
\]

Output:

\[
\begin{align*}
INDX &= (0, 6, 8)
\end{align*}
\]
Chapter 14. Interpolation

The interpolation subroutines are described here.

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 226).

Table 226. List of Interpolation Subroutines

<table>
<thead>
<tr>
<th>Descriptive Name</th>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polynomial Interpolation</td>
<td>SPINT</td>
<td>DPINT</td>
<td>“SPINT and DPINT (Polynomial Interpolation)” on page 1223</td>
</tr>
<tr>
<td>Local Polynomial Interpolation</td>
<td>STPINT</td>
<td>DTPINT</td>
<td>“STPINT and DTPINT (Local Polynomial Interpolation)” on page 1228</td>
</tr>
<tr>
<td>Cubic Spline Interpolation</td>
<td>SCSINT</td>
<td>DCSINT</td>
<td>“SCSINT and DCSINT (Cubic Spline Interpolation)” on page 1232</td>
</tr>
<tr>
<td>Two-Dimensional Cubic Spline Interpolation</td>
<td>SCSIN2</td>
<td>DCSIN2</td>
<td>“SCSIN2 and DCSIN2 (Two-Dimensional Cubic Spline Interpolation)” on page 1237</td>
</tr>
</tbody>
</table>

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 64.
Interpolation Subroutines

This contains the interpolation subroutine descriptions.
SPINT and DPINT (Polynomial Interpolation)

Purpose

These subroutines compute the Newton divided difference coefficients and perform a polynomial interpolation through a set of data points at specified abscissas.

Table 227. Data Types

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPINT</td>
<td>Short-precision real</td>
</tr>
<tr>
<td>DPINT</td>
<td>Long-precision real</td>
</tr>
</tbody>
</table>

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);

On Entry

- **x** is the vector 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 227.

- **y** is the vector 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 227.

- **n** is the number of elements in vectors *x*, *y*, and *c*—that is, the number of data points. Specified as: an integer; *n* ≥ 0.

- **c** is the vector of length *n*, where:
  - If *ninit* ≤ 0, all elements of *c* are undefined on entry.
  - If *ninit* > 0, *c* contains the Newton divided difference coefficients, *c*<sub>j</sub>, for *j* = 1, *ninit*, for the interpolating polynomial through the data points (*x*<sub>j</sub>, *y*<sub>j</sub>) for *j* = 1, *ninit*. If *ninit* < *n*, the values of *c*<sub>j</sub> 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 227.

- **ninit** indicates the following:
  - If *ninit* ≤ 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*<sub>j</sub>, *y*<sub>j</sub>) 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 \( n_{init} < n \), the first \( n_{init} \) Newton divided difference coefficients in \( c \) were computed for the data points \((x_j, y_j)\) for \( j = 1, n_{init} \). The coefficients are updated for the additional data points \((x_j, y_j)\) for \( j = n_{init}+1, n \) on this entry.

Specified as: an integer; \( n_{init} \leq n \).

\( t \) 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 227 on page 1223.

\( s \) 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: an integer; \( m \geq 0 \).

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 227 on page 1223.

\( n_{init} \) 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 \( n_{init} \).) Returned as: an integer; \( n_{init} = n \).

\( s \) 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 227 on page 1223.

Notes
1. In your C program, argument \( n_{init} \) 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:

\( x_i \) are elements of vector \( x \).
\( y_i \) are elements of vector \( y \).

The interpolated value at each \( t_i \) is returned in \( s_i \) for \( i = 1, m \). See references [22 on page 1364] and [63 on page 1367]. The interpolating values returned in \( s \) are computed using the Newton divided difference coefficients, as defined here.
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.

For SPINT, the Newton divided differences and interpolating values are accumulated in long precision.

Newton Divided Differences and Interpolating Values:

The Newton divided differences of the following data points:

$(x_j, y_j)$ for $j = 1, n$

where $x_j \neq x_i$ if $j \neq i$ 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 $j = 1, 2, ..., n$

$\delta_1 y_j = \frac{(y_{j+1} - y_j)}{(x_{j+1} - x_j)}$ for $j = 1, 2, ..., n-1$

For $k = 2, 3, ..., n-1$:

$\delta_k y_j = \frac{(\delta_{k-1} y_{j+1} - \delta_{k-1} y_j)}{(x_{j+k} - x_j)}$ 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:

$c_n = y_n$

$c_{n-1} = \delta_1 y_{n-1}$

$c_{n-2} = \delta_2 y_{n-2}$

$
\vdots$

$c_1 = \delta_{n-1} y_1$

Also, the interpolating values in $s$, in terms of $c$, are as follows for $i = 1, m$: 

Chapter 14. Interpolation 1225
\[ s_i = c_n + (t_i - x_n) c_{n-1} + (t_i - x_{n-1}) (t_i - x_n) c_{n-2} + \ldots + (t_i - x_2) (t_i - x_3) \ldots (t_i - x_n) c_1 \]

**Error conditions**

**Computational Errors**

None

**Input-Argument Errors**

1. \( n < 0 \)
2. \( \text{ninit} > n \)
3. \( m < 0 \)

**Examples**

**Example 1**

This example shows a quadratic polynomial interpolation on the initial call with the specified data points; that is, \( \text{NINIT} = 0 \), and \( C \) contains all undefined values. On output, \( \text{NINIT} \) and \( C \) are updated with new values.

Call Statement and Input:

```
<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>N</th>
<th>C</th>
<th>NINIT</th>
<th>T</th>
<th>S</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL SPINT(X, Y, 3, C, 0, T, S, 2 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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, \( \text{NINIT} = N = 3 \), and \( C \) contains the values defined on output in Example 1. On output here, the values in \( \text{NINIT} \) and \( C \) are unchanged.

Call Statement and Input:

```
<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>N</th>
<th>C</th>
<th>NINIT</th>
<th>T</th>
<th>S</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL SPINT(X, Y, 3, C, 3, T, S, 2 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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, $0 < \text{NINIT} < N$. On output here, the values in \text{NINIT} and \text{C} are updated. The interpolating polynomial is a degree of 4.

Call Statement and Input:

\[
\begin{array}{ccccccccc}
X & Y & N & C & \text{NINIT} & T & S & M \\
\hline
\end{array}
\]

\[
\begin{array}{ccccccccc}
& & & & & & & \\
\text{CALL SPINT(} & X, & Y, & 5, & C, & 3, & T, & S, & 2 \text{)} \\
\end{array}
\]

\[
\begin{align*}
X &= (-0.50, 0.00, 1.00, -1.00, 0.50) \\
Y &= (0.25, 0.00, 1.00, 1.10, 0.26) \\
C &= (1.00, 1.00, 1.00, . , . ) \\
T &= (-0.10, 0.10)
\end{align*}
\]

Output:

\[
\begin{align*}
\text{C} &= (0.04, -0.06, 1.02, -0.56, 0.26) \\
\text{NINIT} &= 5 \\
\text{S} &= (0.0072, 0.0130)
\end{align*}
\]
STPINT and DTPINT (Local Polynomial Interpolation)

Purpose

These subroutines perform a polynomial interpolation at specified abscissas, using data points selected from a table of data.

Table 228. Data Types

<table>
<thead>
<tr>
<th></th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x, y, t, s, aux$</td>
<td>STPINT</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>STPINT</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DTPINT</td>
</tr>
</tbody>
</table>

Syntax

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL STPINT</td>
</tr>
<tr>
<td>C and C++</td>
<td>stpint</td>
</tr>
</tbody>
</table>

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 228.

$y$ 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 228.

$n$ is the number of elements in vectors $x$ and $y$—that is, the number of data points. Specified as: an integer; $n \geq 0$.

$nint$ is the number of data points to be used in the interpolation at any given point.

Specified as: an 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 228.

$s$ 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: an 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 228 on page 1228. On output, the contents are overwritten.

\( \text{naux} \)

is the size of the work area specified by \( \text{aux} \)—that is, the number of elements in \( \text{aux} \).

Specified as: an integer, where:

If \( \text{naux} = 0 \) 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, it must have the following value:

For STPINT

\[
\text{naux} \geq \text{nint} + m
\]

For 32-bit integer arguments

For 64-bit integer arguments

\[
\text{naux} \geq \text{nint} + 2m
\]

For DTPINT

\[
\text{naux} \geq \text{nint} + m
\]

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 228 on page 1228.

Notes

1. Vectors \( x, y \), and \( t \) must have no common elements with vector \( s \) or work area \( \text{aux} \); otherwise, results are unpredictable. See “Concepts” on page 75.

2. The elements of vector \( x \) must be distinct and must be sorted into ascending order; that is, \( x_1 < x_2 < \ldots < x_n \). Otherwise, results are unpredictable. For details on how to do this, see “ISORT, SSORT, and DSORT (Sort the Elements of a Sequence)” on page 1204.

3. The elements of vector \( t \) should be sorted into ascending order; that is, \( t_1 \leq t_2 \leq t_3 \leq \ldots \leq t_m \). Otherwise, performance is affected.

4. You have the option of having the minimum required value for \( \text{naux} \) dynamically returned to your program. For details, see “Using Auxiliary Storage in ESSL” on page 51.

Function

Polynomial interpolation is performed at specified abscissas, \( t_i \) for \( i = 1, m \), in vector \( t \), using \( \text{nint} \) points selected from the following data:

\[(x_j, y_j) \quad \text{for } j = 1, n\]

where:

\( x_j \) 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 \(k = \text{nint}/2\):

For \(t_i \leq x_{k+1}\), the first \(\text{nint}\) points are used.
For \(t_i > x_{\text{nint}+k}\), the last \(\text{nint}\) points are used.
Otherwise, points \(h\) through \(h+\text{nint}-1\) are used, where:

\[x_{h+k-1} < t_i \leq x_{h+k}\]

The interpolated value at each \(t_i\) is returned in \(s_i\) for \(i = 1, m\). See references [22 on page 1364] and [63 on page 1367]. If \(n, \text{nint}, \) 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 1224.

For STPINT, the Newton divided differences and interpolating values are accumulated in long precision.

**Error conditions**

**Resource Errors**
- Error 2015 is unrecoverable, \(\text{naux} = 0\), and unable to allocate work area.

**Computational Errors**
- None

**Input-Argument Errors**
1. \(n < 0\)
2. \(n\text{int} < 0\) or \(n\text{int} > n\)
3. \(m < 0\)
4. Error 2015 is recoverable or \(\text{naux} \neq 0\), and \(\text{naux}\) 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.

**Examples**

**Example 1**

This example shows interpolation using two data points—that is, linear interpolation—at each \(t_i\) value.

Call Statement and Input:

```
CALL STPINT( X, Y, 10, 2, T, S, 5, AUX, 7 )
```

\[
X = (0.0, 0.4, 1.0, 1.5, 2.1, 2.6, 3.0, 3.4, 3.9, 4.3)
\]
\[
Y = (1.0, 2.0, 3.0, 4.0, 5.0, 5.0, 4.0, 4.0, 3.0, 2.0, 1.0)
\]

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:
\begin{verbatim}
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)
\end{verbatim}
SCSINT and DCSINT (Cubic Spline Interpolation)

Purpose

These subroutines compute the coefficients of the cubic spline through a set of data points and evaluate the spline at specified abscissas.

<table>
<thead>
<tr>
<th>Table 229. Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>x, y, C, t, s</td>
</tr>
<tr>
<td>Subroutine</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C and C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL SCSINT</td>
<td>DCSINT (x, y, c, n, init, t, s, m)</td>
</tr>
</tbody>
</table>

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 229.

- 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 229.

- 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.

- 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: an 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 $\leq 0$, 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 \( \text{init} = 1 \), the coefficients are uninitialized. The value in \( c_{11} \) is used as the spline derivative at \( x_1 \).

If \( \text{init} = 2 \), the coefficients are uninitialized. The value in \( c_{21} \) is used as the spline derivative at \( x_n \).

If \( \text{init} = 3 \), 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 \( \text{init} > 3 \), 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: an 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 229 on page 1232](#).

\( s \) 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: an 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 229 on page 1232](#).

\( \text{init} \) 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 \( \text{init} \).) Returned as: an integer; \( \text{init} = 4 \).

\( s \) 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 229 on page 1232](#).

[Notes](#)

1. In your C program, argument \( \text{init} \) 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, \( x_1 < x_2 < \ldots < x_n \). Otherwise, results are unpredictable. For details on how to do this, see [“ISORT, SSORT, and DSORT (Sort the Elements of a Sequence)” on page 1204](#).

[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:

\( (x_j, y_j) \quad \text{for} \quad j = 1, n \)

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 [22 on page 1364] and [63 on page 1367]. 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:

\[
 s_i = c_{j1} + c_{j2} (x_i - t_i) + c_{j3} (x_i - t_i)^2 + c_{j4} (x_i - t_i)^3 \quad \text{for} \quad i = 1, m 
\]

where:

\[
 j = 1 \quad \text{for} \quad t_i \leq x_1 \\
 j = k \quad \text{for} \quad x_j < t_i \leq x_k \quad \text{such that} \quad x_{k-1} < t_i \leq x_k \\
 j = n \quad \text{for} \quad x_n < t_i
\]

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 cubic function through the points whose derivative at \( x_1 \) is \( c_{11} \).
- If \( n = 2 \) and \( init = 2 \), the constructed spline is a cubic 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. \( m < 0 \)

**Examples**

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
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)
```

```
C = 0.1
```

```
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
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)
```

Chapter 14. Interpolation 1235
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:

```
<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>C</th>
<th>N</th>
<th>INIT</th>
<th>T</th>
<th>S</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
<td>C</td>
<td>N</td>
<td>INIT</td>
<td>T</td>
<td>S</td>
<td>M</td>
</tr>
</tbody>
</table>
```

```
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)
```

```
C = [-1.0 . . . .
     0.1 . . . .
     . . . . . .
     . . . . . .
     . . . . . .]
```

Output:

```
C =  
   0.000 1.000 3.230 1.230
   1.000 -1.770 -0.460 1.230
   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:

```
<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>C</th>
<th>N</th>
<th>INIT</th>
<th>T</th>
<th>S</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
<td>C</td>
<td>N</td>
<td>INIT</td>
<td>T</td>
<td>S</td>
<td>M</td>
</tr>
</tbody>
</table>
```

```
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)

Purpose

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 230. Data Types

<table>
<thead>
<tr>
<th>x, y, Z, t, u, aux, S</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SCSIN2</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DCSIN2</td>
</tr>
</tbody>
</table>

Syntax

Fortran

CALL SCSIN2 | DCSIN2 (x, y, z, n1, n2, ldz, t, u, m1, m2, s, lds, aux,iaux)

C and C++

scsin2 | dcsin2 (x, y, z, n1, n2, ldz, t, u, m1, m2, s, lds, aux,iaux);

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 230.

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 230.

z is the matrix Z, containing the data at (x, y) 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 230.

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: an integer; n1 ≥ 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: an integer; n2 ≥ 0.

ldz is the leading dimension of the array specified for z.

Specified as: an integer; ldz > 0 and ldz ≥ 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 230.
**u** is the vector \( u \) of length \( m_2 \), containing the y-coordinates at which the spline is evaluated.

Specified as: a one-dimensional array of (at least) length \( m_2 \), containing numbers of the data type indicated in Table 230 on page 1237.

**\( m_1 \)** is the number of elements in vector \( t \)—that is, the number of x-coordinates at which the spline interpolation is evaluated. Specified as: an integer; \( m_1 \geq 0 \).

**\( m_2 \)** is the number of elements in vector \( u \)—that is, the number of y-coordinates at which the spline interpolation is evaluated. Specified as: an integer; \( m_2 \geq 0 \).

**s** See On Return.

**lds**

is the leading dimension of the array specified for \( s \).

Specified as: an integer; \( lds > 0 \) and \( lds \geq m_1 \).

**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 numbers of the data type indicated in Table 228 on page 1228. 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: an 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:

**For SCSIN2**

**For 32-bit integer arguments**

\[
naux \geq (10)(\max(n_1, n_2)) + (n_2 + 1)(m_1) + 2(m_2)
\]

**For 64-bit integer arguments**

\[
naux \geq (10)(\max(n_1, n_2)) + (n_2 + 2)(m_1) + 3(m_2)
\]

**For DCSIN2**

\[
naux \geq (10)(\max(n_1, n_2)) + (n_2 + 1)(m_1) + 2(m_2)
\]

**On Return**

**s** is the matrix \( S \) with elements \( s_{kh} \) that contain the interpolation values at \((t_k, u_h)\) for \( k = 1, m_1 \) and \( h = 1, m_2 \). Returned as: an \( lds \) by (at least) \( m_2 \) array, containing numbers of the data type indicated in Table 230 on page 1237.

**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 \textit{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, $x_1 < x_2 < ... < x_{n1}$ and $y_1 < y_2 < ... < y_{n2}$. Otherwise, results are unpredictable. For details on how to do this, see “ISORT, SSORT, and DSORT (Sort the Elements of a Sequence)” on page 1204.

4. You have the option of having the minimum required value for $n_{aux}$ dynamically returned to your program. For details, see “Using Auxiliary Storage in ESSL” on page 51.

Function

Interpolation is performed at a specified set of points:

$$(t_k, u_h) \quad \text{for } k = 1, m1 \text{ and } h = 1, m2$$

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 $j = 1, n2$:

$$z_{ij} \quad \text{for } i = 1, n1 \text{ and } j = 1, n2$$

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:

$$(x_i, z_{ij}) \quad \text{for } i = 1, n1$$

The following interpolation values are then computed:

$$s_{y_j}(t_k) \quad \text{for } k = 1, m1$$

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:

$$\left(y_j, s_{y_j}(t_k)\right) \quad \text{for } j = 1, n2$$
The following interpolation values are then computed:

\[ s_{kh} = s_{lk}(u_h) \quad \text{for} \quad l = 1, m2 \]

See references [63 on page 1367] and [71 on page 1367]. 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 \( naux \neq 0 \), 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.

**Examples**

**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:**

\[
\begin{align*}
X & = (0.0, 0.2, 0.3, 0.4, 0.5, 0.7) \\
Y & = (0.0, 0.2, 0.3, 0.4, 0.6) \\
Z & = \begin{bmatrix}
0.000 & 0.008 & 0.027 & 0.064 & 0.216 \\
0.008 & 0.016 & 0.035 & 0.072 & 0.224 \\
0.027 & 0.035 & 0.054 & 0.091 & 0.243 \\
0.064 & 0.072 & 0.091 & 0.128 & 0.280 \\
0.125 & 0.133 & 0.152 & 0.189 & 0.341 \\
0.343 & 0.351 & 0.370 & 0.407 & 0.559 
\end{bmatrix} \\
T & = (0.10, 0.15, 0.25, 0.35) \\
U & = (0.05, 0.25, 0.45)
\end{align*}
\]

**Output:**
$S = \begin{bmatrix}
0.001 & 0.017 & 0.095 \\
0.003 & 0.019 & 0.097 \\
0.016 & 0.031 & 0.110 \\
0.043 & 0.059 & 0.137
\end{bmatrix}$
Chapter 15. Numerical Quadrature

The numerical quadrature subroutines are described.

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 231. List of Numerical Quadrature Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPTNQ</td>
<td>DPTNQ</td>
<td>“SPTNQ and DPTNQ (Numerical Quadrature Performed on a Set of Points)” on page 1247</td>
</tr>
<tr>
<td>SGLNQ†</td>
<td>DGLNQ†</td>
<td>“SGLNQ and DGLNQ (Numerical Quadrature Performed on a Function Using Gauss-Legendre Quadrature)” on page 1250</td>
</tr>
<tr>
<td>SGLNQ2†</td>
<td>DGLNQ2†</td>
<td>“SGLNQ2 and DGLNQ2 (Numerical Quadrature Performed on a Function Over a Rectangle Using Two-Dimensional Gauss-Legendre Quadrature)” on page 1253</td>
</tr>
<tr>
<td>SGLGQ†</td>
<td>DGLGQ†</td>
<td>“SGLGQ and DGLGQ (Numerical Quadrature Performed on a Function Using Gauss-Laguerre Quadrature)” on page 1259</td>
</tr>
<tr>
<td>SGRAQ†</td>
<td>DGRAQ†</td>
<td>“SGRAQ and DGRAQ (Numerical Quadrature Performed on a Function Using Gauss-Rational Quadrature)” on page 1262</td>
</tr>
<tr>
<td>SGHMQ†</td>
<td>DGHMQ†</td>
<td>“SGHMQ and DGHMQ (Numerical Quadrature Performed on a Function Using Gauss-Hermite Quadrature)” on page 1266</td>
</tr>
</tbody>
</table>

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

Use Considerations

This 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 [33 on page 1365], [72 on page 1367], and [111 on page 1369].

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 subroutine and the function evaluation to achieve optimal performance. Details on how to do this are given in “Notes” on page 1254.
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 64.

Programming Considerations for the SUBF Subroutine

This describes how to design and code the subf subroutine for use by the numerical quadrature subroutines.

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_j, where t_j 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 here. Examples of coding a subf subroutine in C, and C++ are provided in Example

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 language-related coding rules for setting up subf in your program are described in the following:

- “Setting Up a User-Supplied Subroutine for ESSL in Fortran” on page 133
- “Setting Up a User-Supplied Subroutine for ESSL in C” on page 153
- “Setting Up a User-Supplied Subroutine for ESSL in C++” on page 168
Numerical Quadrature Subroutines

This contains the numerical quadrature subroutine descriptions.
SPTNQ and DPTNQ (Numerical Quadrature Performed on a Set of Points)

Purpose

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 232. Data Types

<table>
<thead>
<tr>
<th>x, y, xyint, eest</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SPTNQ</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DPTNQ</td>
</tr>
</tbody>
</table>

Syntax

Fortran

```fortran
CALL SPTNQ | DPTNQ (x, y, n, xyint, eest)
```

C and C++

```c
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 232.

\(y\) 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 232.

\(n\) 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 1248.

Specified as: an 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 232.

\(eest\)

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 [33 on page 1365] and [72 on page 1367].
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 1204.

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 [72 on page 1367] and [33 on page 1365].

Error conditions

Computational Errors
None

Input-Argument Errors
$n < 2$

Examples
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:

\[
\begin{array}{c|c|c|c|c|c|}
X & Y & N & XYINT & EEST \\
\hline
\end{array}
\]

CALL SPTNQ( X, Y, 10, XYINT, EEST )

\[
\begin{align*}
X &= (4.3, 3.9, 3.4, 3.0, 2.6, 2.1, 1.5, 1.0, 0.4, 0.0) \\
Y &= (3.3, 3.5, 3.0, 4.0, 4.5, 5.0, 4.0, 3.0, 2.0, 1.0)
\end{align*}
\]

Output:

XYINT = -15.137
EEST = 0.003
SGLNQ and DGLNQ (Numerical Quadrature Performed on a Function Using Gauss-Legendre Quadrature)

Purpose

These functions approximate the integral of a real valued function over a finite interval, using the Gauss-Legendre Quadrature method of specified order.

<table>
<thead>
<tr>
<th>Table 233. Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>a, b, Result</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>SGLNQ</th>
<th>DGLNQ (subf, a, b, n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>sglq</td>
<td>dglnq (subf, a, b, n);</td>
</tr>
</tbody>
</table>

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 1244.

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.

Specified as: a number of the data type indicated in Table 233

b

is the upper limit of integration, b.

Specified as: a number of the data type indicated in Table 233

n

is the order of the quadrature method to be used.

Specified as: an 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 233

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 1251, 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 1244.
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 \( 2n \):

\[
\int_{a}^{b} f(x) \, dx
\]

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 \( 2n \). See references [33 on page 1365] and [111 on page 1369]. 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.

Examples

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:

\[
\int_{0.0}^{2.0} (x^2 + e^x) \, dx
\]

The user-supplied subroutine \( \text{FUN1} \), which evaluates the integrand function, is coded in Fortran as follows:

```fortran
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:

```fortran
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)

Purpose

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 234. Data Types

<table>
<thead>
<tr>
<th>a, b, c, d, Z, Result</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGLNQ2</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGLNQ2</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>SGLNQ2</th>
<th>DGLNQ2 (subf, a, b, n1, c, d, n2, z, ldz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>sglqnq2</td>
<td>dglnq2 (subf, a, b, n1, c, d, n2, z, ldz);</td>
</tr>
</tbody>
</table>

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 1244.

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 234.

\( b \)

is the upper limit of integration, \( b \), for the first variable integrated.

Specified as: a number of the data type indicated in Table 234.

\( n1 \)

is the order of the quadrature method to be used for the first variable integrated.

Specified as: an 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 234.

\( d \)

is the upper limit of integration, \( d \), for the second variable integrated.

Specified as: a number of the data type indicated in Table 234.

\( n2 \)

is the order of the quadrature method to be used for the second variable integrated.

Specified as: an 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 234 on page 1253.

**ldz**

is the size of the leading dimension of the array specified for z.

Specified as: an integer; ldz > 0 and ldz ≥ n1.

**On Return**

**Function value**

is the approximation of the integral. Returned as: a number of the data type indicated in Table 234 on page 1253.

**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 1244.

**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 1250 and references [33 on page 1365] and [111 on page 1369]. 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:

\[
\int_{u_1}^{u_2} \int_{r_1}^{r_2} f(x, y) \, dx \, dy = \int_{r_1}^{r_2} \int_{u_1}^{u_2} f(x, y) \, dy \, dx
\]

Results are mathematically equivalent. However, because the algorithm is computed in a different way, results may not be bitwise identical.

Table 235 shows how to assign your variables to the \_GLNQ2 and \textit{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.

Table 235. How to Assign Your Variables for \( x-y \) Integration Versus \( y-x \) Integration

<table>
<thead>
<tr>
<th>_GLNQ2 and SUBF Arguments</th>
<th>Variables for ( x-y ) Integration</th>
<th>Variables for ( y-x ) Integration</th>
</tr>
</thead>
<tbody>
<tr>
<td>For _GLNQ2:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( a )</td>
<td>( r_1 )</td>
<td>( u_1 )</td>
</tr>
<tr>
<td>( b )</td>
<td>( r_2 )</td>
<td>( u_2 )</td>
</tr>
<tr>
<td>( n_1 )</td>
<td>( \text{(order for } x) )</td>
<td>( \text{(order for } y) )</td>
</tr>
<tr>
<td>( c )</td>
<td>( u_1 )</td>
<td>( r_1 )</td>
</tr>
<tr>
<td>( d )</td>
<td>( u_2 )</td>
<td>( r_2 )</td>
</tr>
<tr>
<td>( n_2 )</td>
<td>( \text{(order for } y) )</td>
<td>( \text{(order for } x) )</td>
</tr>
<tr>
<td>For \textit{subf}:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( s )</td>
<td>( x )</td>
<td>( y )</td>
</tr>
<tr>
<td>( t )</td>
<td>( y )</td>
<td>( x )</td>
</tr>
<tr>
<td>( n_1 )</td>
<td>( \text{(order for } x) )</td>
<td>( \text{(order for } y) )</td>
</tr>
<tr>
<td>( n_2 )</td>
<td>( \text{(order for } y) )</td>
<td>( \text{(order for } x) )</td>
</tr>
</tbody>
</table>

**Error conditions**

**Computational Errors**

None

**Input-Argument Errors**

1. \( ldz \leq 0 \)
2. \( n_1 > ldz \)
3. \( n_1 \) or \( n_2 \) is not an allowable value, as listed in the syntax for this argument.

**Examples**

**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:

\[ \int_{-2.0}^{2.0} \int_{0.0}^{2.0} (e^x \sin y) \, dx \, dy \]

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
   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]);
   }
```

1256   ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
Using C++ for SUBF:

The user-supplied subroutine FUN1, which evaluates the integrand function, is coded in C++ as follows:

```c
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]);
    }
}
```

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 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:

$$\int_{0.0}^{20.0} \int_{0.0}^{1.0} (\cos x \sin y) \, dx \, dy = \int_{0.0}^{1.0} \int_{0.0}^{20.0} (\cos x \sin y) \, dy \, dx$$

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:

```fortran
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
```
\[ Z(I,J) = \cos(T(J)) \times \sin(S(I)) \]

CONTINUE
RETURN
END

**Note:** The same coding principles for achieving good performance that are noted in Example 1 also apply to this user-supplied subroutine \texttt{FUN2}.

Program Statements and Input:

EXTERNAL \texttt{FUN2}.

\[ \text{SUBF A B N1 C D N2 Z LDZ} \]

\[ \text{YXINT = SGLNQ2( FUN2, 0.0, 20.0, 48, 0.0, 1.0, 5, Z, 48 )} \]

\[ \text{FUN2 = (see above)} \]
\[ \text{Z = (not relevant)} \]

Output:

\[ \text{YXINT = 0.4981} \]
SGLGQ and DGLGQ (Numerical Quadrature Performed on a Function Using Gauss-Laguerre Quadrature)

**Purpose**

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 236. Data Types*

<table>
<thead>
<tr>
<th>a, b, Result</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGLGQ</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGLGQ</td>
</tr>
</tbody>
</table>

**Syntax**

<table>
<thead>
<tr>
<th>Fortran</th>
<th>SGLGQ</th>
<th>DGLGQ (subf, a, b, n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>sglgq</td>
<td>dglgq (subf, a, b, n);</td>
</tr>
</tbody>
</table>

**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 1244.

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 236

*b* is the scaling constant *b* for the exponential.

Specified as: a number of the data type indicated in Table 236 *b* > 0 or *b* < 0.

*n* is the order of the quadrature method to be used.

Specified as: an 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 236

**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 1260, 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 1244.

**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:

\[
\begin{align*}
(a, \infty) & \quad \text{if } b > 0 \\
(-\infty, a) & \quad \text{if } b < 0
\end{align*}
\]

The method of order \( n \) is theoretically exact for integrals of the following form,
where \( f \) is a polynomial of degree less than \( 2n \):

\[
\int_a^\infty f(x)e^{-bx}dx \quad \text{if } b > 0
\]

\[
\int_{-\infty}^a f(x)e^{-bx}dx \quad \text{if } b < 0
\]

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 [33 on page 1365] and [111 on page 1369]. 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.

**Examples**

**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 \texttt{FUN1}, which evaluates the integrand function, is
coded in Fortran as follows:

```fortran
SUBROUTINE FUN1 (T,Y,N)
INTEGER*4 N
REAL*4 T(*),Y(*)
```

DO 1 I=1,N
  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
  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)

Purpose

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 237. Data Types

<table>
<thead>
<tr>
<th>a, b, Result</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGRAQ</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGRAQ</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>SGRAQ</th>
<th>DGRAQ (subf, a, b, n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>sgraq</td>
<td>dgraq (subf, a, b, n)</td>
</tr>
</tbody>
</table>

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 1244.

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 237

b

is the centering constant b for the integrand.

Specified as: a number of the data type indicated in Table 237

n

is the order of the quadrature method to be used.

Specified as: an 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 237

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," 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 1244.

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:

\[
(a, \infty) \quad \text{if } a + b > 0 \\
(-\infty, a) \quad \text{if } a + b < 0
\]

The method of order \( n \) is theoretically exact for integrals of the following form, where \( f \) is a polynomial of degree less than \( 2n \):

\[
\int_{a}^{\infty} f\left(\frac{1}{x+b}\right) \frac{1}{(x+b)^2} \, dx \quad \text{if } a + b > 0 \\
\int_{-\infty}^{a} f\left(\frac{1}{x+b}\right) \frac{1}{(x+b)^2} \, dx \quad \text{if } a + b < 0
\]

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 \( 2n \):

\[
f\left(\frac{1}{x+b}\right) \frac{1}{(x+b)^2}
\]

See references [33 on page 1365] and [111 on page 1369]. The result is returned as the function value to a Fortran, C, or C++ 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.

Examples

Example 1
This example shows how to compute the integral of the function \( f \) given by:

\[
f(x) = \left(\frac{e^{1.0}}{x}\right) / x^2
\]

over the interval \((-\infty, -2.0)\), using the Gauss-Rational method with 10 points:
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 \((4.0, \infty)\), 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 )

Output:
XINT = 2.00
SGHMQ and DGHMQ (Numerical Quadrature Performed on a Function Using Gauss-Hermite Quadrature)

Purpose

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>
<thead>
<tr>
<th>Table 238. Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>a, b, Result</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>SGHMQ</th>
<th>DGHMQ (subf, a, b, n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>sghmq</td>
<td>dghmq (subf, a, b, n);</td>
</tr>
</tbody>
</table>

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 1244.

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 238

b

is the scaling constant b for the exponential.

Specified as: a number of the data type indicated in Table 238; b > 0.

n

is the order of the quadrature method to be used.

Specified as: an 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 238

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 1267, 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 1244.
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 \(\infty\). The method of order \(n\) is theoretically exact for integrals of the following form, where \(f\) is a polynomial of degree less than \(2n\):

\[
\int_{-\infty}^{\infty} f(x) e^{-b(x-a)^2} \, dx
\]

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 \(2n\):

\[
f(x) e^{-b(x-a)^2}
\]

See references [33 on page 1365] and [111 on page 1369]. The result is returned as the function value to a Fortran, C, or C++ 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.

Examples

Example

This example shows how to compute the integral of the function \(f\) given by:

\[
f(x) = x^2 e^{-2(x+5.0)^2}
\]

over the interval \((-\infty, \infty)\), using the Gauss-Hermite method with 4 points:

\[
\int_{-\infty}^{\infty} \left(x^2 e^{-2(x+5.0)^2}\right) dx
\]

The user-supplied subroutine \textsc{fun1}, which evaluates the integrand function, is coded in Fortran as follows:

```fortran
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 here.

Overview of the Random Number Generation Subroutines

Random number generation subroutines generate uniformly distributed random numbers or normally distributed random numbers using one of the following algorithms:

- SIMD-oriented Mersenne Twister algorithm
- Multiplicative congruential methods
- Polar methods
- Tausworth's exclusive-or algorithm

Table 239. List of Random Number Generation Initialization Subroutines

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITRNG</td>
<td>“INITRNG (Initialize Random Number Generators)” on page 1271</td>
</tr>
</tbody>
</table>

Table 240. List of Random Number Generation Subroutines

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SURNG</td>
<td>DURNG</td>
<td>“SURNG and DURNG (Generate a Vector of Uniformly Distributed Pseudo-Random Numbers)” on page 1276</td>
</tr>
<tr>
<td>SNRNG</td>
<td>DNRNG</td>
<td>“SNRNG and DNRNG (Generate a Vector of Normally Distributed Pseudo-Random numbers)” on page 1279</td>
</tr>
<tr>
<td>SURAND</td>
<td>DURAND</td>
<td>“SURAND and DURAND (Generate a Vector of Uniformly Distributed Random Numbers)” on page 1283</td>
</tr>
<tr>
<td>SNRAND</td>
<td>DNRAND</td>
<td>“SNRAND and DNRAND (Generate a Vector of Normally Distributed Random Numbers)” on page 1286</td>
</tr>
<tr>
<td>SURXOR§</td>
<td>DURXOR§</td>
<td>“SURXOR and DURXOR (Generate a Vector of Long Period Uniformly Distributed Random Numbers)” on page 1289</td>
</tr>
</tbody>
</table>

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

Use Considerations

If you need a very long period random number generator, you should select the following subroutines:

- SURNG rather than SURAND or SURXOR
- DURNG rather than DURAND or DURXOR
- SNRNG rather than SNRAND
- DNRNG rather than DNRAND.
Random Number Generation Subroutines

This contains the random number generation subroutine descriptions.
INTRNG (Initialize Random Number Generators)

Purpose

This subroutine initializes the selected pseudo-random number generator for use in subsequent calls to SURNG, DURNG, SNRNG or DNRNG. To generate a repeatable or non-repeatable vector of pseudo-random numbers, follow the call to INTRNG with one or more calls to SURNG, DURNG, SNRNG or DNRNG.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL INTRNG (iopt, irepeat, iseed, liseed, istate, listate)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>initrng (iopt, irepeat, iseed, liseed, istate, listate);</td>
</tr>
</tbody>
</table>

On Entry

iopt
indicates the random number generator desired for use, where:

If \( \textit{iopt} = 1 \), a single-precision, SIMD-oriented Mersenne Twister pseudo-random number generator with a period of \( 2^{19937} - 1 \) (SFMT19937) is used.

If \( \textit{iopt} = 2 \), a long-precision, SIMD-oriented Mersenne Twister pseudo-random number generator with a period of \( 2^{19937} - 1 \) (DSFMT19937) is used.

Specified as: an integer; \( \textit{iopt} = 1 \) or 2.

irepeat
indicates whether repeatable or non-repeatable pseudo-random number sequences will be generated, where:

If \( \textit{irepeat} = 0 \), the pseudo-random number generator uses values from \( \textit{iseed} \) to generate repeatable pseudo-random number sequences.

If \( \textit{irepeat} = 1 \), the pseudo-random number generator uses hardware-generated values to generate non-repeatable pseudo-random number sequences.

Specified as: an integer; \( \textit{irepeat} = 0 \) or 1.

iseed
If \( \textit{irepeat} = 0 \), \( \textit{iseed} \) is an array containing the initial seed values to use in initializing the pseudo-random number generator to generate repeatable pseudo-random number sequences.

If \( \textit{irepeat} = 1 \), \( \textit{iseed} \) is ignored.

Specified as: a one-dimensional integer array of (at least) length \( \text{max}(1,\textit{liseed}) \).

liseed
is the number of elements in array ISEE0, where:

If \( \textit{irepeat} = 0 \), \( \textit{liseed} \) is determined as follows:

**32-bit integer environment**

If \( \textit{iopt} = 1 \) or 2, \( \textit{liseed} \geq 624 \).

**64-bit integer environment**

If \( \textit{iopt} = 1 \) or 2, \( \textit{liseed} \geq 312 \).

Note: If \( \textit{irepeat} = 0 \) and insufficient seeds are provided, the seed values supplied in \( \textit{iseed} \) are used and this subroutine initializes the remaining seed values on the basis of the supplied seed values.
If irepeat = 1, liseed is ignored.

Specified as: If irepeat = 0, an integer > 0.

istate
See "On Return".

istate
If listate ≠ -1, listate is the number of elements in the array istate, where listate depends on both the environment the subroutine is running in and the value of iopt, as follows:

32-bit integer environment
- If iopt = 1, listate ≥ 696.
- If iopt = 2, listate ≥ 839.

64-bit integer environment
- If iopt = 1, listate ≥ 348.
- If iopt = 2, listate ≥ 420.

If listate = -1, an istate size query is assumed. The subroutine returns the minimum required size of istate in the output argument listate.

Specified as: an integer; -1 or > 0.

On Return
istate
If listate > 0 on entry, istate contains information about the pseudo-random number generator and the initial seeds for use in subsequent calls to SURNG, DURNG, SNRNG or DNRNG.

If listate = -1 on entry, then istate is unchanged.

Returned as: a one-dimensional integer array of (at least) length max(1,listate)

listate
If listate = -1 on entry, then on return it contains the minimum required size of istate.

Otherwise, it remains unchanged.

Returned as: an integer.

Notes
1. In your C program, argument listate must be passed by reference.
2. For a 64-bit integer environment where iopt = 1 or iopt = 2, if liseed is larger than \(2^{31}\), only the first \(2^{31}-1\) iseed values are used to initialize the istate output value.
3. iseed and istate must have no common elements; otherwise, results are unpredictable.

Function
This subroutine initializes the selected pseudo-random number generator for use in subsequent calls to SURNG, DURNG, SNRNG or DNRNG. To generate a repeatable or non-repeatable vector of pseudo-random numbers, follow the call to INITRNG with one or more calls to SURNG, DURNG, SNRNG or DNRNG.

The following pseudo-random number generators are supported:
1. SIMD-oriented fast Mersenne Twister pseudo-random number generator SFMT19937 (see [96 on page 1369]) with a period length equal to $2^{19937}-1$ of the produced sequence.

2. Double precision floating point SFMT19937 pseudo-random number generator DSFMT19937 ((see [97 on page 1369]) with a period length equal to $2^{19937}-1$ of the produced sequence.

See references [95 on page 1369], [96 on page 1369] and [97 on page 1369].

Error conditions

Computational Errors
None

Input-Argument Errors
1. $iopt \neq 1$ or $2$
2. $irepeat = 0$ and $liseed < 1$
3. In a 32-bit integer environment:
   - $iopt = 1$ and $listate \neq -1$ and $listate < 696$.
   - $iopt = 2$ and $listate \neq -1$ and $listate < 839$.
4. In a 64-bit integer environment:
   - $iopt = 1$ and $listate \neq -1$ and $listate < 348$.
   - $iopt = 2$ and $listate \neq -1$ and $listate < 420$.

Examples

Example 1

This example shows a call to INITRNG to find the optimal size of the $istate$ array needed by the SFMT19937 pseudo-random number generator.

Call Statement and Input:

```
<table>
<thead>
<tr>
<th>IOPT</th>
<th>IREPEAT</th>
<th>ISEED</th>
<th>LISEED</th>
<th>ISTATE</th>
<th>LISTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IREPEAT</td>
<td>ISEED</td>
<td>LISEED</td>
<td>ISTATE</td>
<td></td>
</tr>
</tbody>
</table>
```

CALL INITRNG( 1, IREPEAT, ISEED, LISEED, ISTATE, -1 )

Output:

LISTATE = 696 (in a 32-bit integer environment)

LISTATE = 348 (in a 64-bit integer environment)

Example 2

This example shows a call to INITRNG to find the optimal size of the $istate$ array needed by the DSFMT19937 pseudo-random number generator.

Call Statement and Input:

```
<table>
<thead>
<tr>
<th>IOPT</th>
<th>IREPEAT</th>
<th>ISEED</th>
<th>LISEED</th>
<th>ISTATE</th>
<th>LISTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>IREPEAT</td>
<td>ISEED</td>
<td>LISEED</td>
<td>ISTATE</td>
<td></td>
</tr>
</tbody>
</table>
```

CALL INITRNG( 2, IREPEAT, ISEED, LISEED, ISTATE, -1 )

Output:

LISTATE = 839 (in a 32-bit integer environment)

LISTATE = 420 (in a 64-bit integer environment)

Example 3

This example shows how to initialize the $istate$ array with the seed values for the SFMT19937 pseudo-random number generator to generate repeatable random sequences in a subsequent call to SURNG or SNRNG.
Call Statement and Input:

<table>
<thead>
<tr>
<th>IOPT</th>
<th>IREPEAT</th>
<th>ISEED</th>
<th>LISEED</th>
<th>ISTATE</th>
<th>LISTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL INITRNG( 1, 0, ISEED, LISEED, ISTATE, 1000 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In a 32-bit integer environment:

ISEED(1) = 0
ISEED(2) = 1234
ISEED(3) = 0
ISEED(4) = 5678
LISEED = 4

In a 64-bit integer environment:

ISEED(1) = 1234
ISEED(2) = 5678
LISEED = 2

Output:

istate contains an array of seeds for the SFMT19937 pseudo-random number generator to generate repeatable random sequences, which can be used in a subsequent call to SURNG or SNRNG.

Example 4

This example shows how to initialize the istate array with the seed values for the SFMT19937 pseudo-random number generator to generate repeatable random sequences in a subsequent call to DURNG or DNRNG.

Call Statement and Input:

<table>
<thead>
<tr>
<th>IOPT</th>
<th>IREPEAT</th>
<th>ISEED</th>
<th>LISEED</th>
<th>ISTATE</th>
<th>LISTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL INITRNG( 2, 0, ISEED, LISEED, ISTATE, 1000 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In a 32-bit integer environment:

ISEED(1) = 0
ISEED(2) = 1234
ISEED(3) = 0
ISEED(4) = 5678
LISEED = 4

In a 64-bit integer environment:

ISEED(1) = 1234
ISEED(2) = 5678
LISEED = 2

Output:

istate contains an array of seeds for the SFMT19937 pseudo-random number generator, which can be used in a subsequent call to DURNG or DNRNG.

Example 5

This example shows how to initialize the istate array with the seed values for the SFMT19937 pseudo-random number generator to generate non-repeateable random sequences in a subsequent call to DURNG or DNRNG.

Call Statement and Input:

<table>
<thead>
<tr>
<th>IOPT</th>
<th>IREPEAT</th>
<th>ISEED</th>
<th>LISEED</th>
<th>ISTATE</th>
<th>LISTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL INITRNG( 2, 1, ISEED, LISEED, ISTATE, 1000 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output:
i\text{state} \text{ contains } \text{ an array of seeds for the DSFMT19937 pseudo-random number generator, which can be used in a subsequent call to DURNG or DNRNG.}
SURNG and DURNG (Generate a Vector of Uniformly Distributed Pseudo-Random Numbers)

Purpose

These subroutines generate a repeatable or non-repeatable vector \( x \) of uniform pseudo-random numbers uniformly distributed over the interval \([a, b]\).

For the initial call to these subroutines, you must initialize the pseudo-random number generator with a preceding call to INITRNG.

Table 241. Data Types

<table>
<thead>
<tr>
<th>( x, a, b )</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SURNG</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DURNG</td>
</tr>
</tbody>
</table>

Syntax

| Fortran                      | CALL SURNG | DURNG \((n, a, b, x, istate, listate)\) |
|------------------------------|------------|
| C and C++                    | surng      | durng \((n, a, b, x, istate, listate)\); |

On Entry

\( n \) is the number of pseudo-random numbers to be generated.

Specified as: an integer; \( n \geq 0 \).

\( a \) is the left boundary of the interval \([a, b]\).

Specified as: a number of the data type indicated in Table 241.

\( b \) is the right boundary of the interval \([a, b]\).

Specified as: a number of the data type indicated in Table 241.

\( x \) See "On Return".

\( istate \)

is an array containing information about the current state of the pseudo-random number generator.

Note: If you are invoking this subroutine for the first time, \( istate \) must be the output of a preceding call to subroutine INITRNG, as follows:

- For SURNG, INITRNG must have been invoked with \( iopt = 1 \)
- For DURNG, INITRNG must have been invoked with \( iopt = 2 \)

Specified as: a one-dimensional integer array of (at least) length \( listate \).

\( listate \)

is the number of elements in the array \( istate \) and depends on both the environment the subroutine is running in and the value of \( iopt \) specified on the previous call to INITRNG, as follows:

32-bit integer environment

- If INITRNG was called with \( iopt = 1 \), \( listate \geq 696 \).
- If INITRNG was called with \( iopt = 2 \), \( listate \geq 839 \).

64-bit integer environment
If INITRNG was called with $iopt = 1$, $listate \geq 348$.
If INITRNG was called with $iopt = 2$, $listate \geq 420$.

Specified as: an integer; $listate > 0$.

On Return

$x$ is a vector of length $n$, containing the uniformly distributed pseudo-random numbers.

Returned as: a one-dimensional array of (at least) length $n$, containing numbers of the data type indicated in Table 241 on page 1276.

$istate$ is an array of length $listate$ containing updated information about the state of the pseudo-random number generator for use in subsequent calls to this subroutine.

Returned as: a one-dimensional integer array of (at least) length $listate$.

Notes

$x$ and $istate$ must have no common elements; otherwise, results are unpredictable.

Function

These subroutines generate a repeatable or non-repeatable vector $x$ of uniform pseudo-random numbers uniformly distributed over the interval $[a, b]$.

For the initial call to these subroutines, you must initialize the pseudo-random number generator with a preceding call to INITRNG.

The computation involves the following steps:
1. Retrieve the information for the initialized pseudo-random number generator.
2. Generate the uniformly distributed sequence with the selected pseudo-random number generator.
3. Scale the sequence of pseudo-random numbers.

See references [95 on page 1369], [96 on page 1369] and [97 on page 1369].

If $n$ is 0, no computation is performed.

Error conditions

Computational Errors
None

Input-Argument Errors
1. $n < 0$
2. $a \geq b$
3. $istate$ is not initialized (by a preceding call to INITRNG)
4. $istate$ is initialized (by a preceding call to INITRNG):
   • With $iopt = 2$ for a call to SURNG
   • With $iopt = 1$ for a call to DURNG
5. $listate$ is less than the minimum required value.
Examples

Example 1

This example shows a call to SURNG to generate 10 uniformly distributed short-precision pseudo-random numbers between 0.0 and 1.0.

Call Statement and Input:

```
N A B X ISTATE LISTATE
| | | | | |
CALL SURNG( 10 , 0.0 , 1.0 , X , ISTATE , 1000 )
```

ISTATE = (same as output ISTATE in Example 3)

Note: For the initial call to SURNG, you must initialize the pseudo-random number generator with a preceding call to INITRNG (see Example 3).

Output:

```
X =
0.439785
0.064906
0.385660
0.695451
0.496463
0.154272
0.002247
0.725402
0.037238
0.892588
```

ISTATE = contains the updated state of the pseudo-random number generator.

Example 2

This example shows a call to DURNG to generate 10 uniformly distributed long-precision pseudo-random numbers between 0.0 and 1.0.

Call Statement and Input:

```
N A B X ISTATE LISTATE
| | | | | |
CALL DURNG( 10 , 0.0 , 1.0 , X , ISTATE , 1000 )
```

ISTATE = (same as output ISTATE in Example 4)

Note: For the initial call to DURNG, you must initialize the pseudo-random number generator with a preceding call to INITRNG (see Example 4).

Output:

```
X =
0.948207
0.388311
0.758121
0.430842
0.261129
0.693552
0.113275
0.607048
0.192948
0.669879
```

ISTATE = contains the updated state of the pseudo-random number generator.
SNRNG and DNRNG (Generate a Vector of Normally Distributed Pseudo-Random numbers)

Purpose

These subroutines generate a repeatable or non-repeatable vector \( x \) of normally distributed pseudo-random numbers normally distributed with a mean of \( rmean \) and a standard deviation of \( sigma \), using the BoxMuller2 method.

For the initial call to these subroutines, you must initialize the pseudo-random number generator with a preceding call to INITRNG.

Table 242. Data Types

<table>
<thead>
<tr>
<th>( x, rmean, sigma )</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SNRNG</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DNRNG</td>
</tr>
</tbody>
</table>

Syntax

Fortran

```
CALL SNRNG | DNRNG (n, rmean, sigma, x, istate, listate)
```

C and C++

```
SNRNG | DNRNG (n, rmean, sigma, x, istate, listate);
```

On Entry

\( n \) is the number of pseudo-random numbers to be generated.

Specified as: an integer; \( n \) must be an even number and \( n \geq 0 \).

\( rmean \)

is the mean value of the distribution.

Specified as: a number of the data type indicated in Table 242.

\( sigma \)

is the standard deviation value of the distribution.

Specified as: a number of the data type indicated in Table 242.

\( x \) See "On Return".

\( istate \)

is an array containing information about the current state of the pseudo-random number generator.

Note: If you are invoking this subroutine for the first time, \( istate \) must be the non-zero output of a preceding call to subroutine INITRNG, as follows:

- For SNRNG, INITRNG must have been invoked with \( iopt = 1 \)
- For DNRNG, INITRNG must have been invoked with \( iopt = 2 \)

Specified as: a one-dimensional integer array of (at least) length \( listate \).

\( listate \)

is the number of elements in the array \( istate \) and depends on both the environment the subroutine is running in and the value of \( iopt \) specified on the previous call to INITRNG, as follows:

32-bit pointer environment

- If INITRNG was called with \( iopt = 1 \), \( listate \geq 696 \).
• If INITRNG was called with $iopt = 2$, $listate \geq 839$.

64-bit pointer environment
• If INITRNG was called with $iopt = 1$, $listate \geq 348$.
• If INITRNG was called with $iopt = 2$, $listate \geq 420$.

Specified as: an integer; $listate > 0$.

On Return
$x$ 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 242 on page 1279

$listate$ is an array of length $listate$ containing updated information about the state of the pseudo-random number generator for use in subsequent calls to this subroutine.

Returned as: a one-dimensional integer array of (at least) length $listate$.

Notes
$x$ and $listate$ must have no common elements; otherwise, results are unpredictable.

Function
These subroutines generate a repeatable or non-repeatable vector $x$ of normally distributed pseudo-random numbers normally distributed with a mean of $rmean$ and a standard deviation of $sigma$, using the BoxMuller2 method.

For the initial call to these subroutines, you must initialize the pseudo-random number generator with a preceding call to INITRNG.

The computation involves the following steps:
1. Retrieve the information for the initialized pseudo-random number generator.
2. Generate the uniformly distributed sequence with the selected pseudo-random number generator.
3. Generate the normally distributed sequence using the BoxMuller2 method.

See references \[13 on page 1364\], \[95 on page 1369\], \[96 on page 1369\] and \[97 on page 1369\].

If $n$ is 0, no computation is performed.

Error conditions

Computational Errors
None

Input-Argument Errors
1. $n < 0$ or $n$ is an odd number
2. $sigma \leq 0$
3. $listate$ is not initialized (by a preceding call to INITRNG)
4. $listate$ is initialized:
- With \( iopt = 2 \) for a call to SNRNG
- With \( iopt = 1 \) for a call to DNRNG

5. listate is less than the minimum required value.

Examples

Example 1

This example shows a call to SNRNG to generate 10 normally distributed short-precision pseudo-random numbers with a mean value of 0.0 and a standard deviation of 1.0.

Call Statement and Input:

```
CALL SNRNG( 10, 0.0, 1.0, X, ISTATE, 1000 )
```

ISTATE = (same as output ISTATE in Example 3)

**Note:** For the initial call to SNRNG, you must initialize the pseudo-random number generator with a preceding call to INITRNG (see Example 3).

Output:

```
X = [-0.426951, 0.988221, 0.929709, -0.331744, -0.965826, 0.662854, 0.066279, -0.010326, 0.172133, 0.215101]
```

ISTATE = contains the updated state of the pseudo-random number generator.

Example 2

This example shows a call to DNRNG to generate 10 normally distributed long-precision pseudo-random numbers with a mean value of 0.0 and a standard deviation of 1.0.

Call Statement and Input:

```
CALL DNRNG( 10, 0.0, 1.0, X, ISTATE, 1000 )
```

ISTATE = (same as output ISTATE in Example 4)

**Note:** For the initial call to DNRNG, you must initialize the pseudo-random number generator with a preceding call to INITRNG (see Example 4).

Output:

```
X = [-0.426951, -1.570857, -1.858332, -0.709286, -1.528250, -0.010326, 0.729566, -0.270181]
```
| 0.305498 |
| -0.383550 |
| 0.573548 |
| -0.315877 |

ISTATE = contains the updated state of the pseudo-random number generator.
SURAND and DURAND (Generate a Vector of Uniformly Distributed Random Numbers)

Purpose

These subroutines generate vector \( x \) of uniform \((0,1)\) pseudo-random numbers, using the multiplicative congruential method with a user-specified seed.

Table 243. Data Types

<table>
<thead>
<tr>
<th>( x )</th>
<th>( seed )</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Long-precision real</td>
<td>SURAND</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DURAND</td>
</tr>
</tbody>
</table>

Note: If you need a very long period random number generator, use SURXOR and DURXOR instead of these subroutines.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL SURAND</th>
<th>DURAND ((seed, n, x))</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>surand</td>
<td>durand ((seed, n, x));</td>
</tr>
</tbody>
</table>

On Entry

\( seed \)

is the initial value used to generate the random numbers.

Specified as: a number of the data type indicated in Table 243. 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 \( 1.0 \leq seed < (2147483647.0 = 2^{31}-1) \).

Note: \( seed \) is always a long-precision real number, even in SURAND.

\( n \)

is the number of random numbers to be generated.

Specified as: an integer; \( n \geq 0 \).

\( x \) 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 243. It is a whole number whose value is \( 1.0 \leq seed < (2147483647.0 = 2^{31}-1) \).

\( x \) 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 243.

Notes

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:

\[ s_i = (a(s_{i-1})) \mod(m) = (a's_0) \mod(m) \]
\[ x_i = s_i/m \quad \text{for } i = 1, 2, ..., n \]

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 [90 on page 1368] and [94 on page 1369]. 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 $seed \geq 2147483647.0$

**Examples**

**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, 0.5014866557925740, 0.4895853057920864, 0.4602344475967038, 0.1603607578018497, 0.1832563756887132, 0.9899062002030695, 0.3535068129904134)
```
SNRAND and DNRAND (Generate a Vector of Normally Distributed Random Numbers)

Purpose

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 244. Data Types

<table>
<thead>
<tr>
<th>x, aux</th>
<th>seed</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Long-precision real</td>
<td>SNRAND</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DNRAND</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th></th>
<th>CALL SNRAND</th>
<th>DNRAND (seed, n, x, aux, naux)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>snrand</td>
<td>dnrand (seed, n, x, aux, naux);</td>
</tr>
</tbody>
</table>

On Entry

**seed**

is the initial value used to generate the random numbers.

Specified as: a number of the data type indicated in Table 244. It must be a whole number; that is, the fraction part must be 0. Its value must be $1.0 \leq seed < (2^{31}-1)$.

**Note:** seed is always a long-precision real number, even in SNRAND.

**n**

is the number of random numbers to be generated.

Specified as: an integer; n must be an even number and $n \geq 0$.

**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 244. They can have any value.

**naux**

is the size of the work area specified by aux.

Specified as: an 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, $naux \geq n/2$.

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 244 on page 1286](#). It is a whole number whose value is \(1.0 \leq \text{seed} < (2^{31} - 1)\).

\(x\) 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 244 on page 1286](#).

**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 \(aux\) dynamically returned to your program. For details, see "Using Auxiliary Storage in ESSL" on page 51.

**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 [90 on page 1368](#).

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 \(1\):

\[
\begin{align*}
  y_j &= 2u_{2j-1} - 1 \\
  z_j &= 2u_{2j} - 1
\end{align*}
\]

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 \(p_j \geq 1\), 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:

\[
\begin{align*}
  x_{2i-1} &= y_j \left((-2 \ln p_j) / p_j\right)^{0.5} \\
  x_{2i} &= z_j \left((-2 \ln p_j) / p_j\right)^{0.5}
\end{align*}
\]

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, \(aux\) = 0, and unable to allocate work area.
Computational Errors
None

Input-Argument Errors
1. \( n < 0 \) or \( n \) is an odd number
2. \( \text{seed} < 1.0 \) or \( \text{seed} \geq 2147483647.0 \)
3. Error 2015 is recoverable or \( \text{naux} \neq 0 \), and \( \text{naux} \) is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

Example 1

This example shows a call to SNRAND to generate 10 random numbers.

Call Statement and Input:

\[
\begin{align*}
\text{SEED} & \quad \text{N} & \quad \text{X} & \quad \text{AUX} & \quad \text{NAUX} \\
\text{CALL} & \quad \text{SNRAND( SEED, 10, X, AUX, 5 )} \\
\end{align*}
\]

\[
\begin{align*}
\text{SEED} & \quad = \quad 80629.0 \\
\end{align*}
\]

Note: It is important to note that \( \text{SEED} \) is a long-precision number, even though \( X \) contains short-precision numbers.

Output:

\[
\begin{align*}
\text{SEED} & \quad = \quad 48669425.0 \\
X & \quad = \quad (0.660649538, \\
& \quad 1.312503695, \\
& \quad 1.906438112, \\
& \quad 0.014065863, \\
& \quad -0.800935328, \\
& \quad -3.058144093, \\
& \quad -0.397426069, \\
& \quad -0.370634943, \\
& \quad -0.064151444, \\
& \quad -0.275887042) \\
\end{align*}
\]

Example 2

This example shows a call to DNRAND to generate 10 random numbers.

Call Statement and Input:

\[
\begin{align*}
\text{SEED} & \quad \text{N} \quad \text{X} \quad \text{AUX} \quad \text{NAUX} \\
\text{CALL} & \quad \text{DNRAND( SEED, 10, X, AUX, 5 )} \\
\end{align*}
\]

\[
\begin{align*}
\text{SEED} & \quad = \quad 80629.0 \\
\end{align*}
\]

Output:

\[
\begin{align*}
\text{SEED} & \quad = \quad 48669425.0 \\
X & \quad = \quad (0.6606495655963802, \\
& \quad 1.312503775886106, \\
& \quad 1.906438137943730, \\
& \quad 0.0140658628770495, \\
& \quad -0.800935314494653, \\
& \quad -3.058144123924853, \\
& \quad -0.3974260645722100, \\
& \quad -0.3706349643478605, \\
& \quad -0.0641514433729399, \\
& \quad -0.275887636332470) \\
\end{align*}
\]
SURXOR and DURXOR (Generate a Vector of Long Period Uniformly Distributed Random Numbers)

Purpose

These subroutines generate a vector $x$ of uniform $[0,1)$ pseudo-random numbers, using the Tausworthe exclusive-or algorithm.

Table 245. Data Types

<table>
<thead>
<tr>
<th>$x$, vseed</th>
<th>iseed</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Integer</td>
<td>SURXOR</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Integer</td>
<td>DURXOR</td>
</tr>
</tbody>
</table>

Syntax

Fortran

```
CALL SURXOR | DURXOR (iseed, n, x, vseed)
```

C and C++

```
surxor | durxor (iseed, n, x, vseed);
```

On Entry

$iseed$

has the following meaning, where:

If $iseed \neq 0$, $iseed$ is the initial value used to generate the random numbers. You specify $iseed \neq 0$ 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: an integer, as indicated in Table 245.

$n$

is the number of random numbers to be generated.

Specified as: an integer; $n \geq 0$.

$x$

See On Return.

$vseed$

is the work area used by this subroutine and has the following meaning, where:

If $iseed \neq 0$, $vseed$ 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 245.

On Return

$iseed$

is set to 0 for subsequent calls to SURXOR or DURXOR. Returned as: an integer, as indicated in Table 245.

$x$

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 245 on page 1289.

$vseed$

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 245 on page 1289.

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:

- $p > q$
- $k = 1, 2, ...$
- $z$ is a bit vector.
- and where:

$$\oplus$$ is the bitwise exclusive-or operation.

For details, see references [62 on page 1367], [88 on page 1368], and [113 on page 1370]. The values of $p$ and $q$ are selected according to the criteria stated in reference [120 on page 1370].
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.

**Examples**

**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,
    0.32990755617007059,
    0.66245283144861666,
    0.24998782843358081,
    0.00566308101257373,
    0.73292147005172925,
    0.74863359794102236,
    0.80505169697755319,
    0.99302499462139138,
    0.04418740640269125,
    0.68912952155409579,
    0.31010611495627916,
    0.63241786342211936,
    0.32994081459690583,
    0.80593898487597615)
Chapter 17. Utilities

The utility subroutines are described here.

Overview of the Utility Subroutines

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

Table 246. List of Utility Subroutines

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINFO</td>
<td>“EINFO (ESSL Error Information-Handler Subroutine)” on page 1296</td>
</tr>
<tr>
<td>ERRSAV</td>
<td>“ERRSAV (ESSL ERRSAV Subroutine)” on page 1299</td>
</tr>
<tr>
<td>ERRSET</td>
<td>“ERRSET (ESSL ERRSET Subroutine)” on page 1300</td>
</tr>
<tr>
<td>ERRSTR</td>
<td>“ERRSTR (ESSL ERRSTR Subroutine)” on page 1302</td>
</tr>
<tr>
<td>IVSSET§</td>
<td>Set the Vector Section Size (VSS) for the ESSL/370 Scalar Library</td>
</tr>
<tr>
<td>IEVOPS§</td>
<td>Set the Extended Vector Operations Indicator for the ESSL/370 Scalar Library</td>
</tr>
<tr>
<td>IESSL</td>
<td>“IESSL (Determine the Level of ESSL Installed)” on page 1303</td>
</tr>
<tr>
<td>SETGPUS</td>
<td>“SETGPUS (Set the Number of GPUs and Identify Which GPUs ESSL Should Use)” on page 1305</td>
</tr>
<tr>
<td>STRIDE</td>
<td>“STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)” on page 1307</td>
</tr>
<tr>
<td>DSRSM</td>
<td>“DSRSM (Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode)” on page 1323</td>
</tr>
<tr>
<td>DGKTRN</td>
<td>“DGKTRN (For a General Sparse Matrix, Convert Between Diagonal-Out and Profile-In Skyline Storage Mode)” on page 1327</td>
</tr>
<tr>
<td>DSKTRN</td>
<td>“DSKTRN (For a Symmetric Sparse Matrix, Convert Between Diagonal-Out and Profile-In Skyline Storage Mode)” on page 1332</td>
</tr>
</tbody>
</table>

§ 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.

Use Considerations

This 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.

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 1031.

**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 contains the utility subroutine descriptions.
EINFO (ESSL Error Information-Handler Subroutine)

Purpose

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 247. 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 149.

Table 247. Computational Error Information Returned by EINFO

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Receiver</th>
<th>Type of Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>2100</td>
<td><em>inf1</em></td>
<td>Lower range of a vector</td>
</tr>
<tr>
<td></td>
<td><em>inf2</em></td>
<td>Upper range of a vector</td>
</tr>
<tr>
<td>2101</td>
<td><em>inf1</em></td>
<td>Index of the eigenvalue that failed to converge</td>
</tr>
<tr>
<td></td>
<td><em>inf2</em></td>
<td>Number of iterations after which it failed to converge</td>
</tr>
<tr>
<td>2102</td>
<td><em>inf1</em></td>
<td>Index of the last eigenvector that failed to converge</td>
</tr>
<tr>
<td></td>
<td><em>inf2</em></td>
<td>Number of iterations after which it failed to converge</td>
</tr>
<tr>
<td>2103</td>
<td><em>inf1</em></td>
<td>Index of the pivot with zero value</td>
</tr>
<tr>
<td>2104</td>
<td><em>inf1</em></td>
<td>Index of the last pivot with nonpositive value</td>
</tr>
<tr>
<td>2105</td>
<td><em>inf1</em></td>
<td>Index of the pivot element near zero causing factorization to fail</td>
</tr>
<tr>
<td>2107</td>
<td><em>inf1</em></td>
<td>Index of the singular value that failed to converge</td>
</tr>
<tr>
<td></td>
<td><em>inf2</em></td>
<td>Number of iterations after which it failed to converge</td>
</tr>
<tr>
<td>2109</td>
<td><em>inf1</em></td>
<td>Iteration count when it was determined that the matrix was not definite</td>
</tr>
<tr>
<td>2114</td>
<td><em>inf1</em></td>
<td>Index of the last eigenvalue that failed to converge</td>
</tr>
<tr>
<td></td>
<td><em>inf2</em></td>
<td>Number of iterations after which it failed to converge</td>
</tr>
<tr>
<td>2115</td>
<td><em>inf1</em></td>
<td>Order of the leading minor that was discovered to have a nonpositive determinant</td>
</tr>
<tr>
<td>2117</td>
<td><em>inf1</em></td>
<td>Column number for which pivot value was near zero</td>
</tr>
<tr>
<td>2118</td>
<td><em>inf1</em></td>
<td>Row number for which pivot value was near zero</td>
</tr>
<tr>
<td>2120</td>
<td><em>inf1</em></td>
<td>Row number of empty row where factorization failed</td>
</tr>
<tr>
<td>2121</td>
<td><em>inf1</em></td>
<td>Column number of empty column where factorization failed</td>
</tr>
<tr>
<td>2126</td>
<td><em>inf1</em></td>
<td>Row number for which pivot value was unacceptable</td>
</tr>
</tbody>
</table>
Table 247. Computational Error Information Returned by EINFO (continued)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Receiver</th>
<th>Type of Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>2145</td>
<td>inf1</td>
<td>First diagonal element with zero value</td>
</tr>
<tr>
<td>2150</td>
<td>inf1</td>
<td>First diagonal element with zero value</td>
</tr>
</tbody>
</table>

Syntax

**Fortran**
```
CALL EINFO (icode[, inf1[, inf2]])
```

**C and C++**
```
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 247 on page 1296, 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: an integer; $icode = 0$ or an error code value indicated in Table 247 on page 1296

**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 $icode \neq 0$, then $inf1$ is the first information receiver, containing numerical information related to the computational error.

Returned as: an integer.

**inf2**

has the following meaning, where:

If $icode = 0$, this argument is not used in the computation.

If $icode \neq 0$, 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 1298.

Returned as: an integer.
Notes

1. If $icode$ is not 0 and is not one of the error codes specified in Table 247 on page 1296, 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, “Coding Your Program,” on page 133.
ERRSAV (ESSL ERRSAV Subroutine)

Purpose

The ERRSA 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 149.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL ERRSAV (ierno, tabent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td><code>errsav (ierno, tabent);</code></td>
</tr>
</tbody>
</table>

**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: an integer; *ierno* must be one of the error numbers in the option table. For a list of these numbers, see Table 45 on page 71.

**On Return**

*tabent*

is the storage area where the option table entry is stored.

Specified as: an area of storage of length 8-bytes.

Notes

Examples of how to use ERRSAV are provided in Chapter 4, “Coding Your Program,” on page 133.
ERRSET (ESSL ERRSET Subroutine)

Purpose

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 149.

Syntax

Fortran

CALL ERRSET (ierno, inoal, inomes, itrace, iusadr, irange)

C and C++

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: an integer; ierno must be one of the error numbers in the option table. For a list of these numbers, see Table 45 on page 71.

inoal

indicates the number of errors allowed before each execution is terminated, where:

- If inoal \leq 0, the specification is ignored, and the number-of-errors option is not changed.
- If inoal = 1, execution is terminated after one error.
- If $2 \leq inoal \leq 255$, 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: an integer, where:

- If iusadr = ENOTRM, then $2 \leq inoal \leq 255$.

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 $0 < inomes \leq 255$, then inomes specifies the number of messages to be printed.
If $inomes > 255$, an unlimited number of error messages is allowed.

Specified as: an integer.

**itrace**

This argument is ignored, but must be specified.

Specified as: an 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 $iusadr$ is zero, the option table is not altered.

If $iusadr$ is one, 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 32-bit integer in a 32-bit integer, 32-bit pointer environment, or as the name of a subroutine; $iusadr = 0$, 1, or ENOTRM.

Specified: as a 64-bit integer in either a 32-bit integer, 64-bit pointer environment or a 64-bit integer, 64-bit pointer environment, or as the name of a subroutine; $iusadr = 0_8$, $1_8$, 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 $irange \geq ierno$, 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: an integer.

**Notes**

1. Examples of how to use ERRSET are provided in Chapter 4, “Coding Your Program,” on page 133.

2. If you specify ENOTRM for $iusadr$, then $inoal$ must be in the following range: $2 \leq inoal \leq 255$. 


ERRSTR (ESSL ERRSTR Subroutine)

Purpose

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 149.

Syntax

<table>
<thead>
<tr>
<th>Fortran</th>
<th>CALL ERRSTR (ierno, tabent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>errstr (ierno, tabent);</td>
</tr>
</tbody>
</table>

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: an integer; **ierno** must be one of the error numbers in the option table. For a list of these numbers, see Table 45 on page 71.

**tabent**

is the storage area containing the table entry data.

Specified as: an area of storage of length 8-bytes.

Notes

Examples of how to use ERRSTR are provided in Chapter 4, “Coding Your Program,” on page 133.
**IESSL (Determine the Level of ESSL Installed)**

**Purpose**

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**

<table>
<thead>
<tr>
<th>Language</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>IESSL()</td>
</tr>
<tr>
<td>C and C++</td>
<td>iesl();</td>
</tr>
</tbody>
</table>

**On Return**

*Function value*

is the level of ESSL installed on your system. It is provided as an 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: an 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 an 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.

**Examples**

**Example 1**

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:

```fortran
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.
SETGPUS (Set the Number of GPUs and Identify Which GPUs ESSL Should Use)

**Purpose**

SETGPUS allows you to set the number and specify which GPUs ESSL should use.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td><code>SETGPUS (ngpus, ids)</code></td>
</tr>
<tr>
<td>C and C++</td>
<td><code>setgpus (ngpus, ids);</code></td>
</tr>
</tbody>
</table>

**On Entry**

- `ngpus` is the number of GPUs ESSL should use.  
  Specified as: an integer; `0 < ngpus ≤` number of CUDA devices.
- `ids` is the array of length `ngpus` containing the IDs for the GPUs ESSL should use.  
  Specified as: an integer; `0 ≤ ids_i <` (number of CUDA devices) for `i = 1, ngpus`.

**Function**

This subroutine allows you to set the number and specify which GPUs ESSL should use.

**Error conditions**

**Resource Errors**

1. The number of OpenMP Threads is less than `ngpus`. ESSL issues attention message 2538-2615 and uses the same number of GPUs as there are OpenMP threads.
2. Not all the CUDA devices specified by the `ids` array are in the same NVIDIA compute mode.

**Input-Argument Errors**

1. SETGPUS has been called either:
   - More than once
   - After the first call to any ESSL subroutine that is GPU enabled.
2. `ngpus ≤ 0` or `ngpus >` (number of CUDA devices).
3. `ids_i < 0` or `ids_i >` (number of CUDA devices) - 1 for `i = 1, ngpus`.
4. NVIDIA compute mode is PROHIBITED for GPUs identified in the `ids` array.
5. Environment variable ESSL_CUDA_HYBRID is not 'yes', 'no', or unset.
6. Environment variable ESSL_CUDA_PIN is not 'yes', 'no', 'pinned', or unset.

**Examples**

**Example**

This example shows setting 2 GPUs that ESSL should use. This call results in ESSL using GPUs 1 and 0 for CUDA applications.

Call Statement and Input:
CALL SETGPUS( 2, IDS)

IDS = (1,0)
STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)

Purpose

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 1308.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL STRIDE (n, incd, incr, dt, iopt)</td>
</tr>
<tr>
<td>C and C++</td>
<td>stride (n, incd, incr, dt, iopt);</td>
</tr>
</tbody>
</table>

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: an 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 Example 1—SCFT.
  Specified as: an 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: an integer; iopt = 0, 1, or 2.
**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: an 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 [Example 1—SCFT](#example1). 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 1031.

**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 [Example 1—SCFT](#example1). 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 1031.

**Error conditions**

**Computational Errors**

None

**Input-Argument Errors**

1. \( n \leq 0 \)
2. \( \text{incd} = 0 \)
3. \( \text{iop} \neq 0, 1, \text{or} 2 \)
4. \( \text{dt} \neq S, D, C, \text{or} Z \)

**Examples**

**Example 1—SCFT**

This example shows the use of the STRIDE subroutine in computing one-dimensional row transforms using the SCFT subroutine.

If \( \text{inc2} = 1 \), the input sequences are stored in the transposed form as rows of a two-dimensional array \( X(1:\text{INC1X},N) \). In this case, the STRIDE subroutine helps in determining a good value of \( \text{inc1x} \) for this array. The required minimum value of \( \text{inc1x} \) is \( m \), the number of Fourier transforms being computed. To find a good value of \( \text{inc1x} \), use STRIDE as follows:

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{N} & \text{INCD} & \text{INCR} & \text{DT} & \text{IOPT} \\
\hline
\end{array}
\]

CALL STRIDE( \( \text{N} \), \( m \), \( \text{INC1X} \), 'C', 0 )

Here, the arguments refer to the SCFT subroutine. In the following table, values of \( \text{inc1x} \) are given (as obtained from the STRIDE subroutine) for some combinations of \( n \) and \( m \) and for POWER3 with 64KB level 1 cache:

\[
\begin{array}{|c|c|c|}
\hline
\text{N} & \text{M} & \text{INC1X} \\
\hline
128 & 64 & 64 \\
240 & 32 & 32 \\
240 & 64 & 65 \\
256 & 256 & 264 \\
512 & 60 & 60 \\
1024 & 64 & 65 \\
\hline
\end{array}
\]

The above example also applies when the output sequences are stored in the transposed form (\( \text{inc2} = 1 \)). In that case, in the above example, \( \text{inc1x} \) is replaced by \( \text{inc1y} \).

In computing column transforms (\( \text{inc1} = \text{inc1y} = 1 \)), the values of \( \text{inc2} \) and \( \text{inc2y} \) 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 \( \text{inc2} = 1 \), the input sequences are stored in the transposed form as rows of a two-dimensional array \( X(1:\text{INC1X},N/2+1) \). 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:

```plaintext
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 POWER3 with 64KB level 1 cache:

<table>
<thead>
<tr>
<th>N</th>
<th>M</th>
<th>INC1X</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>240</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>240</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>256</td>
<td>256</td>
<td>64</td>
</tr>
<tr>
<td>512</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>1024</td>
<td>64</td>
<td>65</td>
</tr>
</tbody>
</table>

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:

```plaintext
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 POWER3 with 64KB level 1 cache:

<table>
<thead>
<tr>
<th>N</th>
<th>M</th>
<th>INC1X</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>240</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>240</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>256</td>
<td>256</td>
<td>64</td>
</tr>
<tr>
<td>512</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>1024</td>
<td>64</td>
<td>65</td>
</tr>
</tbody>
</table>

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 \( \text{inc1y} = 1 \), the two-dimensional output array is stored in the normal form. In this case, the output array can be declared as \( Y(\text{INC2Y}, N2) \), where the required minimum value of \( \text{inc2y} \) is \( n1 \). The STRIDE subroutine helps in picking a good value of \( \text{inc2y} \). To find a good value of \( \text{inc2y} \), use STRIDE as follows:

\[
\begin{array}{cccc}
N & \text{INCD} & \text{INCR} & \text{DT} & \text{IOPT} \\
\hline
\text{CALL STRIDE} & \text{N2} & \text{N1} & \text{INC2Y} & \text{'}C' & 0 \\
\end{array}
\]

Here, the arguments refer to the SCFT2 subroutine. In the following table, values of \( \text{inc2y} \) are given (as obtained from the STRIDE subroutine) for some two-dimensional arrays with \( n1 = n2 \) and for POWER3 with 64KB level 1 cache:

<table>
<thead>
<tr>
<th>( N1 )</th>
<th>( N2 )</th>
<th>( \text{INC2Y} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>128</td>
<td>128</td>
<td>136</td>
</tr>
<tr>
<td>240</td>
<td>240</td>
<td>248</td>
</tr>
<tr>
<td>512</td>
<td>512</td>
<td>520</td>
</tr>
<tr>
<td>840</td>
<td>840</td>
<td>848</td>
</tr>
</tbody>
</table>

If the input array is stored in the normal form (\( \text{inc1x} = 1 \)), the value of \( \text{inc2x} \) is not important. However, if you want to use the same array for input and output, you should use \( \text{inc2x} = \text{inc2y} \).

If \( \text{inc2y} = 1 \), the two-dimensional output array is stored in the transposed form. In this case, the output array can be declared as \( Y(\text{INC1Y}, N1) \), where the required minimum value of \( \text{inc1y} \) is \( n2 \). The STRIDE subroutine helps in picking a good value of \( \text{inc1y} \). To find a good value of \( \text{inc1y} \), use STRIDE as follows:

\[
\begin{array}{cccc}
N & \text{INCD} & \text{INCR} & \text{DT} & \text{IOPT} \\
\hline
\text{CALL STRIDE} & \text{N1} & \text{N2} & \text{INC1Y} & \text{'}C' & 0 \\
\end{array}
\]

Here, the arguments refer to the SCFT2 subroutine. In the following table, values of \( \text{inc1y} \) are given (as obtained from the STRIDE subroutine) for some combinations of \( n1 \) and \( n2 \) and for POWER3 with 64K level 1 cache:

<table>
<thead>
<tr>
<th>( N1 )</th>
<th>( N2 )</th>
<th>( \text{INC1Y} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>128</td>
<td>128</td>
<td>136</td>
</tr>
<tr>
<td>256</td>
<td>240</td>
<td>240</td>
</tr>
<tr>
<td>512</td>
<td>512</td>
<td>520</td>
</tr>
<tr>
<td>840</td>
<td>840</td>
<td>848</td>
</tr>
</tbody>
</table>

If the input array is stored in the transposed form (\( \text{inc2x} = 1 \)), the value of \( \text{inc1x} \) is also important. The above example can be used to find a good value of \( \text{inc1x} \), by replacing \( \text{inc1y} \) with \( \text{inc1x} \). If both arrays are stored in the transposed form, a good value for \( \text{inc1y} \) is also a good value for \( \text{inc1x} \). 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 \( Y(\text{INC2Y}, N2) \), where the required minimum value of \( \text{inc2y} \) is \( n1/2 + 1 \). The STRIDE subroutine helps in picking a good value of \( \text{inc2y} \). To find a good value of \( \text{inc2y} \), use STRIDE as follows:
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 POWER3 with 64KB level 1 cache:

<table>
<thead>
<tr>
<th>N1</th>
<th>N2</th>
<th>INC2Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>240</td>
<td>240</td>
<td>121</td>
</tr>
<tr>
<td>420</td>
<td>420</td>
<td>211</td>
</tr>
<tr>
<td>512</td>
<td>512</td>
<td>257</td>
</tr>
<tr>
<td>840</td>
<td>840</td>
<td>421</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>513</td>
</tr>
<tr>
<td>2048</td>
<td>2048</td>
<td>1032</td>
</tr>
</tbody>
</table>

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:

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 POWER3 with 64KB level 1 cache:

<table>
<thead>
<tr>
<th>N1</th>
<th>N2</th>
<th>INC2Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>240</td>
<td>240</td>
<td>242</td>
</tr>
<tr>
<td>420</td>
<td>420</td>
<td>422</td>
</tr>
<tr>
<td>512</td>
<td>512</td>
<td>514</td>
</tr>
<tr>
<td>840</td>
<td>840</td>
<td>842</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>1026</td>
</tr>
<tr>
<td>2048</td>
<td>2048</td>
<td>2064</td>
</tr>
</tbody>
</table>

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.

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 \((n2)(inc2y)\).

\[
\begin{array}{cccc}
N & INC2Y & INC3Y \\
30 & 30 & 900 \\
32 & 32 & 1032 \\
64 & 64 & 4112 \\
120 & 120 & 14408 \\
128 & 136 & 17416 \\
240 & 240 & 57608 \\
256 & 264 & 67592 \\
420 & 420 & 176400 \\
\end{array}
\]

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 \(inc2x \geq inc2y\) and \(inc3x \geq inc3y\). 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 \(X(INC2X,INC3X/INC2X,N3)\).

\[
\begin{array}{cccc}
N1,N2,N3 & INC2Y & INC3Y \\
30 & 30 & 900 \\
32 & 32 & 1032 \\
64 & 64 & 4112 \\
120 & 120 & 14408 \\
128 & 136 & 17416 \\
240 & 240 & 57608 \\
256 & 264 & 67592 \\
420 & 420 & 176400 \\
\end{array}
\]

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 \(inc2x \geq inc2y\) and \(inc3x \geq inc3y\). 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 \(n1/2+1\).

\[
\begin{array}{cccc}
N & INC2Y & INC3Y \\
30 & 30 & 900 \\
32 & 32 & 1032 \\
64 & 64 & 4112 \\
120 & 120 & 14408 \\
128 & 136 & 17416 \\
240 & 240 & 57608 \\
256 & 264 & 67592 \\
420 & 420 & 176400 \\
\end{array}
\]

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 POWER3 with 64KB level 1 cache:

<table>
<thead>
<tr>
<th>( N1, N2, N3 )</th>
<th>( INC2Y )</th>
<th>( INC3Y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>16</td>
<td>488</td>
</tr>
<tr>
<td>32</td>
<td>17</td>
<td>552</td>
</tr>
<tr>
<td>64</td>
<td>33</td>
<td>2128</td>
</tr>
<tr>
<td>120</td>
<td>61</td>
<td>7320</td>
</tr>
<tr>
<td>128</td>
<td>65</td>
<td>8328</td>
</tr>
<tr>
<td>240</td>
<td>121</td>
<td>29064</td>
</tr>
<tr>
<td>256</td>
<td>129</td>
<td>33032</td>
</tr>
<tr>
<td>420</td>
<td>211</td>
<td>88620</td>
</tr>
</tbody>
</table>

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 \( inc2x \geq 2(inc2y) \) and \( inc3x \geq 2(inc3y) \). 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 \( n1+2 \).

\[
\text{CALL STRIDE}( N2, N1 + 2, INC2Y, 'S', 0 )
\]

Here, the arguments refer to the SCRFT3 subroutine. Next, you should find a good value for \( inc3y \). The minimum acceptable value for \( inc3y \) is \((n2)(inc2y)\).

\[
\text{CALL STRIDE}( N3, N2+INC2Y, INC3Y, 'S', 0 )
\]

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 POWER3 with 64KB level 1 cache:

<table>
<thead>
<tr>
<th>( N1, N2, N3 )</th>
<th>( INC2Y )</th>
<th>( INC3Y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>32</td>
<td>976</td>
</tr>
<tr>
<td>32</td>
<td>34</td>
<td>1104</td>
</tr>
</tbody>
</table>
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.

**Example 10—SCFTD, D = 1**

This example shows the use of the STRIDE subroutine in computing one-dimensional row transforms using the SCFTD subroutine.

If incmx = 1, the input sequences are stored in the transposed form as rows of a two-dimensional array X(INCX(1),N(1)). In this case, the STRIDE subroutine helps in determining a good value of incx1 for this array. The required minimum value of incx1 is m, the number of Fourier transforms being computed. To find a good value of incx1, use STRIDE as follows:

```fortran
CALL STRIDE( N(1), M, INCX(1), 'C', 0 )
```

Here, the arguments refer to the SCFTD subroutine. In the following table, values of incx1 are given (as obtained from the STRIDE subroutine) for some combinations of n1 and m and for POWER6 with 64KB level 1 cache:

<table>
<thead>
<tr>
<th>N</th>
<th>M</th>
<th>INCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>64</td>
<td>66</td>
</tr>
<tr>
<td>240</td>
<td>32</td>
<td>34</td>
</tr>
<tr>
<td>240</td>
<td>64</td>
<td>66</td>
</tr>
<tr>
<td>256</td>
<td>256</td>
<td>264</td>
</tr>
<tr>
<td>512</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>1024</td>
<td>64</td>
<td>66</td>
</tr>
</tbody>
</table>

The above example also applies when the output sequences are stored in the transposed form (incmy = 1). In that case, in the above example, incx1 is replaced by incy1.

In computing column transforms (incx1 = incy1 = 1), the values of incmx and incmy are not very important. For these, any value over the required minimum of n1 can be used.

**Example 11—SCFTD, D = 2**

This example shows the use of the STRIDE subroutine in computing two-dimensional transforms using the SCFTD subroutine with m = 1.

If incy1 = 1, the two-dimensional output array is stored in the normal form. In this case, the output array can be declared as Y(INCY(2),N(2)), where the required minimum value of incy2 is n1. The STRIDE subroutine helps in picking a good value of incy2. To find a good value of incy2, use STRIDE as follows:

```fortran
CALL STRIDE( N(2), N(1), INCY(2), 'C', 0 )
```

Here, the arguments refer to the SCFTD subroutine. In the following table, values of incy2 are given (as obtained from the STRIDE subroutine) for some two-dimensional arrays with n1 = n2 and for POWER6 with 64KB level 1 cache:
If the input array is stored in the normal form \((incx_2 = 1)\), the value of \(incx_2\) is not important. However, if you want to use the same array for input and output, you should use \(incx_2 = incy_2\).

If \(incy_2 = 1\), the two-dimensional output array is stored in the transposed form. In this case, the output array can be declared as \(Y(incy_1, N(1))\), where the required minimum value of \(incy_1\) is \(n_1\). The STRIDE subroutine helps in picking a good value of \(incy_1\). To find a good value of \(incy_1\), use STRIDE as follows:

\[
\begin{array}{cccc}
| N | INC | INC | DT | IOPT |
\end{array}
\]

\[
\text{CALL STRIDE}(N(1), N(2), INCY(1), 'C', 0)
\]

Here, the arguments refer to the SCFTD subroutine. In the following table, values of \(incy_1\) are given (as obtained from the STRIDE subroutine) for some combinations of \(n_1\) and \(n_2\) and for POWER6 with 64KB level 1 cache:

\[
\begin{array}{ccc}
N(1) & N(2) & INCY(1) \\
64 & 64 & 64 \\
128 & 128 & 136 \\
240 & 240 & 240 \\
512 & 512 & 520 \\
840 & 840 & 840 \\
\end{array}
\]

If the input array is stored in the transposed form \((incx_1 = 1)\), the value of \(incx_1\) is also important. The above example can be used to find a good value of \(incx_1\), by replacing \(incy_1\) with \(incx_1\). If both arrays are stored in the transposed form, a good value for \(incy_1\) is also a good value for \(incx_1\). In that situation, the two arrays can also be made equivalent.

Example 12--SCFTD, \(D = 3\)

This example shows the use of the STRIDE subroutine in computing three-dimensional transforms using the SCFTD subroutine with \(m = 1\).

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 \(incy_2\) and \(incy_3\). This requires two calls to the STRIDE subroutine as shown below. First, you should find a good value for \(incy_2\). The minimum acceptable value for \(incy_2\) is \(n_1\).

\[
\begin{array}{cccc}
| N | INC | INC | DT | IOPT |
\end{array}
\]

\[
\text{CALL STRIDE}(N(2), N(1), INCY(2), 'C', 0)
\]

Here, the arguments refer to the SCFTD subroutine. Next, you should find a good value for \(incy_3\). The minimum acceptable value for \(incy_3\) is \((n_2)(incy_2)\) assuming \(incy_1 = 1\).

\[
\begin{array}{cccc}
| N | INC | INC | DT | IOPT |
\end{array}
\]

\[
\text{CALL STRIDE}(N(3), N(2)*INCY(2), INCY(3), 'C', 0)
\]

If \(incy_3\) turns out to be a multiple of \(incy_2\), then \(Y\) can be declared a three-dimensional array as \(Y(INCY(2), INCY(3)/INCY(2), N(3))\). For large problems, this may not happen. In that case, you can declare the \(Y\) array as a two-dimensional array \(Y(0:INCY(3)-1, 0:N(3)-1)\) or a one-dimensional array.
Y(0:INCY(3)*N(3)-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 \( (k1+k2*incy_2,k3) \)
- For the one-dimensional array, location \( (k1+k2*incy_2+k3*incy_3) \)

In the following table, values of \( incy_2 \) and \( incy_3 \) are given (as obtained from the STRIDE subroutine) for some three-dimensional arrays with \( n_1 = n_2 = n_3 \) and for POWER6 with 64KB level 1 cache:

<table>
<thead>
<tr>
<th>N1, N2, N3</th>
<th>INCY(2)</th>
<th>INCY(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>30</td>
<td>900</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>1032</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>4104</td>
</tr>
<tr>
<td>120</td>
<td>120</td>
<td>14400</td>
</tr>
<tr>
<td>128</td>
<td>136</td>
<td>17416</td>
</tr>
<tr>
<td>240</td>
<td>240</td>
<td>57608</td>
</tr>
<tr>
<td>256</td>
<td>264</td>
<td>67592</td>
</tr>
<tr>
<td>420</td>
<td>420</td>
<td>176400</td>
</tr>
</tbody>
</table>

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 \( incx_2 \geq incy_2 \) and \( incx_3 \geq incy_3 \). A simple thing to do is to use \( incx_2 = incy_2 \) and make \( incx_3 \) a multiple of \( incx_2 \) not smaller than \( incy_3 \). Then \( X \) can be declared as a three-dimensional array \( X(INCX(2),INCY(3)/INCX(2),N(3)) \).

**Example 13--SRCFTD, D = 1**

This example shows the use of the STRIDE subroutine in computing one-dimensional row transforms using the SRCFTD subroutine.

If \( incmx \) equal to 1, the input sequences are stored in the transposed form as rows of a two-dimensional array \( X(INCX(1),N(1)) \). In this case, the STRIDE subroutine helps in determining a good value of \( incx_1 \) for this array. The required minimum value of \( incx_1 \) is \( m \), the number of Fourier transforms being computed. To find a good value of \( incx_1 \), use STRIDE as follows:

\[
\begin{align*}
\text{CALL STRIDE}( & N(1), M, \text{INCX}(1), 'S', 0 )
\end{align*}
\]

Here, the arguments refer to the SRCFTD subroutine. In the following table, values of \( incx_1 \) are given (as obtained from the STRIDE subroutine) for some combinations of \( n_1 \) and \( m \) and for POWER6 with 64KB level 1 cache:

<table>
<thead>
<tr>
<th>N(1)</th>
<th>M</th>
<th>INCX(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>240</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>240</td>
<td>64</td>
<td>68</td>
</tr>
<tr>
<td>256</td>
<td>256</td>
<td>272</td>
</tr>
<tr>
<td>512</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>1024</td>
<td>64</td>
<td>64</td>
</tr>
</tbody>
</table>

If \( incmy \) equal to 1, the output sequences are stored in the transposed form as rows of a two-dimensional array \( Y(INCY(1),N(1)/2+1) \). In this case, the STRIDE subroutine helps in determining a good value of \( incy_1 \) for this array. The required minimum value of \( incy_1 \) is \( m \), the number of Fourier transforms being computed. To find a good value of \( incy_1 \), use STRIDE as follows:

\[
\begin{align*}
\text{CALL STRIDE}( & N(1)/2+1, M, \text{INCY}(1), 'C', 0 )
\end{align*}
\]
Here, the arguments refer to the SRCFTD subroutine. In the following table, values of $\text{incy}_1$ are given (as obtained from the STRIDE subroutine) for some combinations of $n_1$ and $m$ and for POWER6 with 64KB level 1 cache:

<table>
<thead>
<tr>
<th>$N(1)$</th>
<th>$M$</th>
<th>$\text{INCY}(1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>64</td>
<td>66</td>
</tr>
<tr>
<td>240</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>240</td>
<td>64</td>
<td>66</td>
</tr>
<tr>
<td>256</td>
<td>256</td>
<td>264</td>
</tr>
<tr>
<td>512</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>1024</td>
<td>64</td>
<td>66</td>
</tr>
</tbody>
</table>

In computing column transforms ($\text{incx}_1$ equal to $\text{incy}_1$ equal to 1), the values of $\text{incm}_x$ and $\text{incm}_y$ are not very important. For these, any value over the required minimum can be used.

**Example 14--SRCFTD, D = 2**

This example shows the use of the STRIDE subroutine in computing two-dimensional transforms using the SRCFTD subroutine with $m$ equal to 1.

If $\text{incy}_1$ equal to 1, the two-dimensional output array is stored in the normal form. In this case, the output array can be declared as $Y(\text{INCY}(2),N(2))$, where the required minimum value of $\text{incy}_2$ is $n_1/2+1$. The STRIDE subroutine helps in picking a good value of $\text{incy}_2$. To find a good value of $\text{incy}_2$, use STRIDE as follows:

```
N INCD INCR DT IOPT
CALL STRIDE( N(2) , N(1)/2+1 , INCY(2) , 'C' , 0 )
```

Here, the arguments refer to the SRCFTD subroutine. In the following table, values of $\text{incy}_2$ are given (as obtained from the STRIDE subroutine) for some two-dimensional arrays with $n_1$ equal to $n_2$ and for POWER6 with 64KB level 1 cache:

<table>
<thead>
<tr>
<th>$N(1)$</th>
<th>$N(2)$</th>
<th>$\text{INCY}(2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>240</td>
<td>240</td>
<td>122</td>
</tr>
<tr>
<td>420</td>
<td>420</td>
<td>212</td>
</tr>
<tr>
<td>512</td>
<td>512</td>
<td>258</td>
</tr>
<tr>
<td>840</td>
<td>840</td>
<td>422</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>514</td>
</tr>
<tr>
<td>2048</td>
<td>2048</td>
<td>1026</td>
</tr>
</tbody>
</table>

If the input array is stored in the normal form ($\text{incx}_1$ equal to 1), the value of $\text{incx}_2$ is not important. However, if you want to use the same array for input and output, you should use $\text{incx}_2$ equal to $2(\text{incy}_2)$.

If $\text{incy}_2$ equal to 1, the two-dimensional output array is stored in the transposed form. In this case, the output array can be declared as $Y(\text{INCY}(1),N(1)/2+1)$, where the required minimum value of $\text{incy}_1$ is $n_2$. The STRIDE subroutine helps in picking a good value of $\text{incy}_1$. To find a good value of $\text{incy}_1$, use STRIDE as follows:

```
N INCD INCR DT IOPT
CALL STRIDE( N(1)/2+1 , N(2) , INCY(1) , 'C' , 0 )
```

Here, the arguments refer to the SRCFTD subroutine. In the following table, values of $\text{incy}_1$ are given (as obtained from the STRIDE subroutine) for some combinations of $n_1$ and $n_2$ and for POWER6 with 64KB level 1 cache:

<table>
<thead>
<tr>
<th>$N(1)$</th>
<th>$N(2)$</th>
<th>$\text{INCY}(1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>240</td>
<td>240</td>
<td>240</td>
</tr>
<tr>
<td>420</td>
<td>420</td>
<td>420</td>
</tr>
</tbody>
</table>
Example 15--SRCFTD, D = 3

This example shows the use of the STRIDE subroutine in computing three-dimensional transforms using the SRCFTD subroutine with m equal to 1.

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 incy2 and incy3. This requires two calls to the STRIDE subroutine as shown below. First, you should find a good value for incy2. The minimum acceptable value for incy2 is n_1/2+1.

\[
\begin{array}{cccc}
N & INCD & INCR & DT & IOPT \\
1 & 30 & 16 & 480 & \\
2 & 32 & 18 & 576 & \\
3 & 64 & 34 & 2176 & \\
4 & 120 & 62 & 7440 & \\
5 & 128 & 66 & 8456 & \\
6 & 240 & 122 & 29280 & \\
7 & 256 & 130 & 33288 & \\
8 & 420 & 212 & 89040 & \\
\end{array}
\]

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 incx_2 equal to 2(incy_2) and incx_3 equal to 2(incy_3).

Example 16--SCRFTD, D = 1

This example shows the use of the STRIDE subroutine in computing one-dimensional row transforms using the SCRFTD subroutine.

If incmx equal to 1, the input sequences are stored in the transposed form as rows of a two-dimensional array X(INCX(1),N(1)). In this case, the STRIDE subroutine helps in determining a good value of incx_1 for this array.
required minimum value of incx₁ is \( m \), the number of Fourier transforms being computed. To find a good value of incx₁, use STRIDE as follows:

\[
\begin{array}{c|c|c|c|c}
N & \text{INCD} & \text{INCR} & \text{DT} & \text{IOPT} \\
\hline
\text{CALL STRIDE(} & N(1)/2+1 & , & M & , \text{INCX(1)} & , \text{'}C\text{'}, \text{0}) \\
\end{array}
\]

Here, the arguments refer to the SCRFTD subroutine. In the following table, values of incx₁ are given (as obtained from the STRIDE subroutine) for some combinations of \( n₁ \) and \( m \) and for POWER6 with 64KB level 1 cache:

<table>
<thead>
<tr>
<th>( N(1) )</th>
<th>( M )</th>
<th>( \text{INCX(1)} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>64</td>
<td>66</td>
</tr>
<tr>
<td>240</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>240</td>
<td>64</td>
<td>66</td>
</tr>
<tr>
<td>256</td>
<td>256</td>
<td>264</td>
</tr>
<tr>
<td>512</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>1024</td>
<td>64</td>
<td>66</td>
</tr>
</tbody>
</table>

If incmy equal to 1, the output sequences are stored in the transposed form as rows of a two-dimensional array \( Y(\text{INCY(1)}, N(1)) \). In this case, the STRIDE subroutine helps in determining a good value of incy₁ for this array. The required minimum value of incy₁ is \( m \), the number of Fourier transforms being computed. To find a good value of incy₁, use STRIDE as follows:

\[
\begin{array}{c|c|c|c|c}
N & \text{INCD} & \text{INCR} & \text{DT} & \text{IOPT} \\
\hline
\text{CALL STRIDE(} & N(1) & , & \text{INCY(1)} & , \text{'}S\text{'}, \text{0}) \\
\end{array}
\]

Here, the arguments refer to the SCRFTD subroutine. In the following table, values of incy₁ are given (as obtained from the STRIDE subroutine) for some combinations of \( n₁ \) and \( m \) and for POWER6 with 64KB level 1 cache:

<table>
<thead>
<tr>
<th>( N(1) )</th>
<th>( M )</th>
<th>( \text{INCY(1)} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>240</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>240</td>
<td>64</td>
<td>68</td>
</tr>
<tr>
<td>256</td>
<td>256</td>
<td>272</td>
</tr>
<tr>
<td>512</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>1024</td>
<td>64</td>
<td>64</td>
</tr>
</tbody>
</table>

In computing column transforms (incx₁ equal to incy₁ equal to 1), the values of incmx and incmy are not very important. For these, any value over the required minimum can be used.

**Example 17--SCRFTD, D = 2**

This example shows the use of the STRIDE subroutine in computing two-dimensional transforms using the SCRFTD subroutine with \( m \) equal to 1.

If incy₁ equal to 1, the two-dimensional output array is stored in the normal form. In this case, the output array can be declared as \( Y(\text{INCY(2)}, N(2)) \), where the required minimum value of incy₂ is \( n₁+2 \). The STRIDE subroutine helps in picking a good value of incy₂. To find a good value of incy₂, use STRIDE as follows:

\[
\begin{array}{c|c|c|c|c}
N & \text{INCD} & \text{INCR} & \text{DT} & \text{IOPT} \\
\hline
\text{CALL STRIDE(} & N(2) & , & \text{N(1)+2} & , \text{INCY(2)} & , \text{'}S\text{'}, \text{0}) \\
\end{array}
\]

Here, the arguments refer to the SCRFTD subroutine. In the following table, values of incy₂ are given (as obtained from the STRIDE subroutine) for some two-dimensional arrays with \( n₁ \) equal to \( n₂ \) and for POWER6 with 64KB level 1 cache:
If the input array is stored in the normal form \((incx_1\) equal to 1), the value of \(incx_2\) is not important. However, if you want to use the same array for input and output, you should use \(incy_2\) equal to \(2incx_2\).

If \(incy_2\) equal to 1, the two-dimensional output array is stored in the transposed form. In this case, the output array can be declared as \(Y(incy_1, N(1)+2)\), where the required minimum value of \(incy_1\) is \(n_2\). The STRIDE subroutine helps in picking a good value of \(incy_1\). To find a good value of \(incy_1\), use STRIDE as follows:

\[
\text{CALL STRIDE}( N(1)+2, N(2), incy_1, 'S', 0 )
\]

Here, the arguments refer to the SCRFTD subroutine. In the following table, values of \(incy_1\) are given (as obtained from the STRIDE subroutine) for some combinations of \(n_1\) and \(n_2\) and for POWER6 with 64KB level 1 cache:

<table>
<thead>
<tr>
<th>(N(1))</th>
<th>(N(2))</th>
<th>(INCY(1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>240</td>
<td>240</td>
<td>244</td>
</tr>
<tr>
<td>420</td>
<td>420</td>
<td>424</td>
</tr>
<tr>
<td>512</td>
<td>512</td>
<td>516</td>
</tr>
<tr>
<td>840</td>
<td>840</td>
<td>844</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>1028</td>
</tr>
<tr>
<td>2048</td>
<td>2048</td>
<td>2052</td>
</tr>
</tbody>
</table>

Example 18--SCRFTD, \(D = 3\)

This example shows the use of the STRIDE subroutine in computing three-dimensional transforms using the SCRFTD subroutine with \(m\) equal to 1.

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 \(incy_2\) and \(incy_3\). This requires two calls to the STRIDE subroutine as shown below. First, you should find a good value for \(incy_2\). The minimum acceptable value for \(incy_2\) is \(n_1+2\).

\[
\text{CALL STRIDE}( N(1)+2, N(2), incy_2, 'S', 0 )
\]

Here, the arguments refer to the SCRFTD subroutine. Next, you should find a good value for \(incy_3\). The minimum acceptable value for \(incy_3\) is \((n_2)(incy_2)\) assuming \(incy_1\) equal to 1.

\[
\text{CALL STRIDE}( N(3), N(2)*incy_2, incy_3, 'S', 0 )
\]

If \(incy_3\) turns out to be a multiple of \(incy_2\), then \(Y\) can be declared as a three-dimensional array as \(Y(incy_2, incy_3/incy_2, N(3))\). For large problems, this may not happen. In that case, you can declare the \(Y\) array as a two-dimensional array \(Y(0:incy_3-1, 0:N(3)-1)\) or a one-dimensional array \(Y(0:incy_3*N(3)-1)\). 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 \((k1+k2*incy_2,k3)\)
• For the one-dimensional array, location \((k_1 + k_2 \times \text{incy}_2 + k_3 \times \text{incy}_3)\)

In the following table, values of \(\text{incy}_2\) and \(\text{incy}_3\) are given (as obtained from the STRIDE subroutine) for some three-dimensional arrays with \(n_1\) equal to \(n_2\) equal to \(n_3\) and for POWER6 with 64KB level 1 cache:

<table>
<thead>
<tr>
<th>N(1), N(2), N(3)</th>
<th>INCY(2)</th>
<th>INCY(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>32</td>
<td>960</td>
</tr>
<tr>
<td>32</td>
<td>36</td>
<td>1152</td>
</tr>
<tr>
<td>64</td>
<td>68</td>
<td>4352</td>
</tr>
<tr>
<td>120</td>
<td>124</td>
<td>14880</td>
</tr>
<tr>
<td>128</td>
<td>132</td>
<td>16912</td>
</tr>
<tr>
<td>240</td>
<td>244</td>
<td>58560</td>
</tr>
<tr>
<td>256</td>
<td>260</td>
<td>66576</td>
</tr>
<tr>
<td>420</td>
<td>424</td>
<td>178080</td>
</tr>
</tbody>
</table>

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 \(\text{incy}_2\) equal to \(2 \times (\text{incx}_2)\) and \(\text{incy}_3\) equal to \(2 \times (\text{incx}_3)\).
DSRSM (Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode)

**Purpose**

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**

<table>
<thead>
<tr>
<th>Language</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL DSRSM (iopt, ar, ja, ia, m, nz, ac, ka, lda)</td>
</tr>
<tr>
<td>C and C++</td>
<td>dsrsm (iopt, ar, ja, ia, m, nz, ac, ka, lda);</td>
</tr>
</tbody>
</table>

**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: an 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 integers; \( 1 \leq (JA \text{ elements}) \leq n \). 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 integers; \( IA(i+1) \geq IA(i) \) for \( i = 1, m+1 \).

- **m**
  
  is the number of rows in sparse matrix \( A \). Specified as: an integer; \( m \geq 0 \).

- **nz**
  
  is the number of columns in output arrays \( AC \) and \( KA \) that are available for use.
  
  Specified as: an 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: an integer; \( 0 < lda \leq m \).

**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: an 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 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 117 and "Storage-by-Rows" on page 122.

### 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 [87 on page 1368].

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.
Examples

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:

$$
\begin{bmatrix}
11.0 & 0.0 & 0.0 & 14.0 \\
0.0 & 22.0 & 0.0 & 24.0 \\
0.0 & 0.0 & 33.0 & 34.0 \\
0.0 & 0.0 & 0.0 & 44.0 \\
\end{bmatrix}
$$

Because there is a maximum of only two nonzero elements in each row of $A$, and argument $\text{nz}$ is specified as 5, columns 3 through 5 of arrays $AC$ and $KA$ are not used.

Call Statement and Input:

```fortran
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:

$\text{NZ} = 2$

$AC = \begin{bmatrix}
11.0 & 14.0 & . & . \\
22.0 & 24.0 & . & . \\
33.0 & 34.0 & . & . \\
44.0 & 0.0 & . & . \\
\end{bmatrix}$

$KA = \begin{bmatrix}
1 & 4 & . & . \\
2 & 4 & . & . \\
3 & 4 & . & . \\
4 & 4 & . & . \\
\end{bmatrix}$

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:

$$
\begin{bmatrix}
11.0 & 0.0 & 0.0 & 14.0 \\
0.0 & 22.0 & 0.0 & 24.0 \\
0.0 & 0.0 & 33.0 & 34.0 \\
14.0 & 24.0 & 34.0 & 44.0 \\
\end{bmatrix}
$$

Because there is a maximum of only four nonzero elements in each row of $A$, and argument $\text{nz}$ is specified as 6, columns 5 and 6 of arrays $AC$ and $KA$ are not used.

Call Statement and Input:

```fortran
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:

\[ \text{NZ} = 4 \]

\[
\begin{bmatrix}
11.0 & 14.0 & 0.0 & 0.0 & . & . \\
22.0 & 24.0 & 0.0 & 0.0 & . & . \\
33.0 & 34.0 & 0.0 & 0.0 & . & . \\
44.0 & 24.0 & 34.0 & 14.0 & . & .
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 4 & 4 & 4 & . & . \\
2 & 4 & 4 & 4 & . & . \\
3 & 4 & 4 & 4 & . & . \\
4 & 2 & 3 & 1 & . & .
\end{bmatrix}
\]
DGKTRN (For a General Sparse Matrix, Convert Between Diagonal-Out and Profile-In Skyline Storage Mode)

**Purpose**

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**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL DGKTRN ((n, au, nu, idu, al, nl, idl, itran, aux, naux))</td>
</tr>
<tr>
<td>C and C++</td>
<td>dgktrn ((n, au, nu, idu, al, nl, idl, itran, aux, naux));</td>
</tr>
</tbody>
</table>

**On Entry**

\( n \) is the order of general sparse matrix \( A \). Specified as: an 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 \( \text{ITRAN}(1) = 0 \), \( A \) is stored in diagonal-out skyline storage mode.
- If \( \text{ITRAN}(1) = 1 \), \( A \) 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: an integer; \( nu \geq 0 \) and \( nu \geq (\text{IDU}(n+1)-1) \).

\( idu \) 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 integers.

\( al \) is the array, referred to as \( AL \), containing the lower triangular part of general sparse matrix \( A \), stored as follows, where:

- If \( \text{ITRAN}(1) = 0 \), \( A \) is stored in diagonal-out skyline storage mode.
- If \( \text{ITRAN}(1) = 1 \), \( A \) 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: an integer; \( nl \geq 0 \) and \( nl \geq (\text{IDL}(n+1)-1) \).

\( idl \) 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 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 1329), 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 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: an 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, it must have one of the following values:

  For 32-bit integer arguments
  \[ naux \geq 2n \]

  For 64-bit integer arguments
  \[ naux \geq 4n \]

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.

\textit{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 integers.

\textit{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.

\textbf{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.

\textit{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 integers.

\textbf{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:

<table>
<thead>
<tr>
<th>Most Recent Computation Performed by DGKFS/DGKFSP</th>
<th>Direction Used by DGKFS/DGKFSP</th>
<th>Direction to Specify in ITRAN(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factor and Solve</td>
<td>Negative</td>
<td>Positive (ITRAN(3) = 1)</td>
</tr>
<tr>
<td>Factor Only</td>
<td>Positive</td>
<td>Negative (ITRAN(3) = -1)</td>
</tr>
<tr>
<td>Solve Only</td>
<td>Negative</td>
<td>Positive (ITRAN(3) = 1)</td>
</tr>
</tbody>
</table>

3. For a description of how sparse matrices are stored in skyline storage mode, see “Profile-In Skyline Storage Mode” on page 126 and “Diagonal-Out Skyline Storage Mode” on page 124.

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 51.

\textbf{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. \( IDU(i+1) \leq IDU(i) \) 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. \( IDL(i+1) \leq IDL(i) \) 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. \( ITRAN(1) \neq 0 \) or 1
13. \( ITRAN(2) \neq 0 \) or 1
14. \( ITRAN(3) \neq -1 \) or 1
15. Error 2015 is recoverable or naux\neq0, and naux is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

Examples

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:

\[
\begin{bmatrix}
11.0 & 12.0 & 13.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
21.0 & 22.0 & 23.0 & 24.0 & 25.0 & 0.0 & 0.0 & 0.0 & 29.0 \\
31.0 & 32.0 & 33.0 & 34.0 & 35.0 & 0.0 & 37.0 & 0.0 & 39.0 \\
41.0 & 42.0 & 43.0 & 44.0 & 45.0 & 46.0 & 47.0 & 0.0 & 49.0 \\
0.0 & 0.0 & 0.0 & 54.0 & 55.0 & 56.0 & 57.0 & 58.0 & 59.0 \\
0.0 & 0.0 & 0.0 & 62.0 & 63.0 & 64.0 & 65.0 & 66.0 & 67.0 & 68.0 & 69.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 74.0 & 75.0 & 76.0 & 77.0 & 78.0 & 79.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 84.0 & 85.0 & 86.0 & 87.0 & 88.0 & 89.0 \\
91.0 & 92.0 & 93.0 & 94.0 & 95.0 & 96.0 & 97.0 & 98.0 & 99.0 \\
\end{bmatrix}
\]

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:

<table>
<thead>
<tr>
<th>N</th>
<th>AU</th>
<th>NU</th>
<th>IDU</th>
<th>AL</th>
<th>NL</th>
<th>IDL</th>
<th>ITRAN</th>
<th>AUX</th>
<th>NAUX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALL DGKTRN( 9, AU, 33, IDU, AL, 35, IDL, ITRAN, AUX, 18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

AU = (11.0, 12.0, 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, 28.0, 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 = (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)

ITRAN = (1, 0, -1)

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:

<table>
<thead>
<tr>
<th>N</th>
<th>AU</th>
<th>NU</th>
<th>IDU</th>
<th>AL</th>
<th>NL</th>
<th>IDL</th>
<th>ITRAN</th>
<th>AUX</th>
<th>NAUX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALL DGKTRN( 9, AU, 33, IDU, AL, 35, IDL, ITRAN, AUX, 18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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)

Purpose

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, idia, itran, aux, naux)

C and C++

dsktrn (n, a, na, idia, itran, aux, naux);

On Entry

n is the order of symmetric sparse matrix A. Specified as: an 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 ITREN(1) = 0, A is stored in diagonal-out skyline storage mode.
- If ITREN(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: an integer; $na \geq 0$ and $na \geq (IDIA(n+1)-1)$.

idia is the array, referred to as IDIA, 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 integers.

itran is an array of parameters, ITREN(i), where:
- ITRN(1) indicates the input storage mode used for matrix A. This determines the arrangement of data in arrays A and IDIA on input, where:
  - If ITREN(1) = 0, diagonal-out skyline storage mode is used.
  - If ITREN(1) = 1, profile-in skyline storage mode is used.
- ITRN(2) indicates the output storage mode used for matrix A. This determines the arrangement of data in arrays A and IDIA on output, where:
  - If ITRN(2) = 0, diagonal-out skyline storage mode is used.
  - If ITRN(2) = 1, profile-in skyline storage mode is used.
- ITRN(3) indicates the direction of sweep that ESSL uses through the matrix A, allowing you to optimize performance (see “Notes” on page 1333), where:
  - If ITRN(3) = 1, matrix A is transformed in the positive direction, starting in row or column 1 and ending in row or column n.
  - If ITRN(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 integers, where:

\[
\begin{align*}
    \text{ITRAN}(1) &= 0 \text{ or } 1 \\
    \text{ITRAN}(2) &= 0 \text{ or } 1 \\
    \text{ITRAN}(3) &= -1 \text{ or } 1
\end{align*}
\]

\textit{aux} has the following meaning:

If \textit{naux} = 0 and error 2015 is unrecoverable, \textit{aux} is ignored. Otherwise, it is the storage work area used by this subroutine. Its size is specified by \textit{naux}.

Specified as: an area of storage, containing \textit{naux} long-precision real numbers.

\textit{naux} is the size of the work area specified by \textit{aux}—that is, the number of elements in \textit{aux}.

Specified as: an integer, where:

If \textit{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

\textbf{For 32-bit integer arguments}

\textit{naux} \geq n

\textbf{For 64-bit integer arguments}

\textit{naux} \geq 2n

\textbf{On Return}

\textit{a} is the array, referred to as \textit{A}, containing the upper triangular part of symmetric sparse matrix \textit{A}, stored as follows, where:

If \textit{ITRAN}(2) = 0, \textit{A} is stored in diagonal-out skyline storage mode.

If \textit{ITRAN}(2) = 1, \textit{A} is stored in profile-in skyline storage mode.

Returned as: a one-dimensional array of (at least) length \textit{na}, containing long-precision real numbers.

\textit{idia}\textbf{g} is the array, referred to as \textit{IDIAG}, containing the relative positions of the diagonal elements of matrix \textit{A} in output array \textit{A}. Returned as: a one-dimensional array of (at least) length \textit{n+1}, containing integers.

\textbf{Notes}

1. Your various arrays must have no common elements; otherwise, results are unpredictable.

2. The \textit{ITRAN}(3) argument allows you to specify the direction of travel through matrix \textit{A} that ESSL takes during the transformation. By properly specifying \textit{ITRAN}(3), you can optimize the performance of the transformation, which is especially beneficial when transforming large matrices.

The direction specified by \textit{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 126 and “Diagonal-Out Skyline Storage Mode” on page 124.

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 51.

**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. \( IDIAG(i+1) \leq IDIAG(i) \) 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. \( ITRAN(1) \neq 0 \) or \( 1 \)
8. \( ITRAN(2) \neq 0 \) or \( 1 \)
9. \( ITRAN(3) \neq -1 \) or \( 1 \)
10. \( naux \) Error 2015 is recoverable or \( naux \neq 0 \), and is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.

**Examples**

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:

\[
\begin{bmatrix}
11.0 & 12.0 & 13.0 & 14.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
12.0 & 22.0 & 23.0 & 24.0 & 25.0 & 26.0 & 0.0 & 28.0 & 0.0 \\
13.0 & 23.0 & 33.0 & 34.0 & 35.0 & 36.0 & 0.0 & 38.0 & 0.0 \\
14.0 & 24.0 & 34.0 & 44.0 & 45.0 & 46.0 & 0.0 & 48.0 & 0.0 \\
0.0 & 25.0 & 35.0 & 45.0 & 55.0 & 56.0 & 57.0 & 58.0 & 0.0 \\
0.0 & 26.0 & 36.0 & 46.0 & 56.0 & 66.0 & 67.0 & 68.0 & 69.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 57.0 & 67.0 & 77.0 & 78.0 & 79.0 \\
0.0 & 28.0 & 38.0 & 48.0 & 58.0 & 68.0 & 78.0 & 88.0 & 89.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 69.0 & 79.0 & 89.0 & 99.0
\end{bmatrix}
\]

Assuming that DSKFS last performed a factorization on matrix \( A \), the direction of the transformation is negative; that is, \( \text{ITRAN}(3) \) is -1. This provides the best performance here.

Call Statement and Input:

\[
\begin{align*}
N & \quad A & \quad NA & \quad IDIAG & \quad ITRAN & \quad AUX & \quad NAUX \\
\text{CALL DSKTRN(} & \quad 9 & , & A & , & 33 & , & IDIAG & , & ITRAN & , & AUX & , & 9 \text{)} \\
A & = (11.0, 22.0, 12.0, 23.0, 13.0, 44.0, 34.0, 24.0, \\
& \quad 14.0, 55.0, 45.0, 35.0, 25.0, 66.0, 56.0, 46.0, 36.0, \\
& \quad 26.0, 77.0, 67.0, 57.0, 88.0, 78.0, 68.0, 58.0, 48.0, \\
& \quad 38.0, 28.0, 99.0, 89.0, 79.0, 69.0) \\
IDIAG & = (1, 2, 4, 7, 11, 15, 20, 23, 30, 34) \\
ITRAN & = (0, 1, -1)
\end{align*}
\]

Output:

\[
\begin{align*}
A & = (11.0, 12.0, 22.0, 13.0, 23.0, 33.0, 14.0, 24.0, 34.0, \\
& \quad 44.0, 25.0, 35.0, 45.0, 55.0, 26.0, 36.0, 46.0, 56.0, \\
& \quad 66.0, 57.0, 67.0, 77.0, 28.0, 38.0, 48.0, 58.0, 68.0, \\
& \quad 78.0, 88.0, 69.0, 79.0, 89.0, 99.0) \\
IDIAG & = (1, 3, 6, 10, 14, 19, 22, 29, 33, 34)
\end{align*}
\]

**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, \( \text{ITRAN}(3) \) is 1. This provides the best performance here.

Call Statement and Input:

\[
\begin{align*}
N & \quad A & \quad NA & \quad IDIAG & \quad ITRAN & \quad AUX & \quad NAUX \\
\text{CALL DSKTRN(} & \quad 9 & , & A & , & 33 & , & IDIAG & , & ITRAN & , & AUX & , & 9 \text{)} \\
A & = \text{(same as output } A \text{ in Example 1)} \\
IDIAG & = \text{(same as output } IDIAG \text{ in Example 1)} \\
ITRAN & = (1, 0, 1)
\end{align*}
\]

Output:

\[
\begin{align*}
A & = \text{(same as input } A \text{ in Example 1)} \\
IDIAG & = \text{(same as input } IDIAG \text{ in Example 1)}
\end{align*}
\]
Part 3. Appendixes
Appendix A. Basic Linear Algebra Subprograms (BLAS) and Complex BLAS (CBLAS)

This appendix lists the ESSL subprograms corresponding to a subprogram in the standard set of BLAS and CBLAS.

### Level 1 BLAS

<table>
<thead>
<tr>
<th>Descriptive Name</th>
<th>Short-Precision Subprogram</th>
<th>Long-Precision Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>“ISAMAX, IDAMAX, ICAMAX, and IZAMAX (Position of the First or Last Occurrence of the Vector Element Having the Largest Magnitude)” on page 233</td>
<td>ISAMAX</td>
<td>IDAMAX</td>
</tr>
<tr>
<td></td>
<td>ICAMAX</td>
<td>IZAMAX</td>
</tr>
<tr>
<td></td>
<td>cblas_isamax</td>
<td>cblas_idamax</td>
</tr>
<tr>
<td></td>
<td>cblas_icamax</td>
<td>cblas_izamax</td>
</tr>
<tr>
<td>“SASUM, DASUM, SCASUM, and DZASUM (Sum of the Magnitudes of the Elements in a Vector)” on page 245</td>
<td>SASUM</td>
<td>DASUM</td>
</tr>
<tr>
<td></td>
<td>SCASUM</td>
<td>DZASUM</td>
</tr>
<tr>
<td></td>
<td>cblas_sasum</td>
<td>cblas_dasum</td>
</tr>
<tr>
<td></td>
<td>cblas_scasum</td>
<td>cblas_dcasum</td>
</tr>
<tr>
<td>“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 248</td>
<td>SAXPY</td>
<td>DAXPY</td>
</tr>
<tr>
<td></td>
<td>CAXPY</td>
<td>ZAXPY</td>
</tr>
<tr>
<td></td>
<td>cblas_saxpy</td>
<td>cblas_zaxpy</td>
</tr>
<tr>
<td></td>
<td>cblas_caxpy</td>
<td></td>
</tr>
<tr>
<td>“SCOPY, DCOPY, CCOPY, and ZCOPY (Copy a Vector)” on page 251</td>
<td>SCOPY</td>
<td>DCOPY</td>
</tr>
<tr>
<td></td>
<td>CCOPY</td>
<td>ZCOPY</td>
</tr>
<tr>
<td></td>
<td>cblas_scopy</td>
<td>cblas_dcopy</td>
</tr>
<tr>
<td></td>
<td>cblas_ccopy</td>
<td>cblas_zcopy</td>
</tr>
<tr>
<td>“SDOT, DDOT, CDOTU, ZDOTU, CDOTC, and ZDOTC (Dot Product of Two Vectors)” on page 254</td>
<td>SDOT</td>
<td>DDOT</td>
</tr>
<tr>
<td></td>
<td>CDOTU</td>
<td>ZDOTU</td>
</tr>
<tr>
<td></td>
<td>cblas_sdot</td>
<td>cblas_ddot</td>
</tr>
<tr>
<td></td>
<td>cblas_cdotu_sub</td>
<td>cblas_dzdot</td>
</tr>
<tr>
<td></td>
<td>cblas_cdotc_sub</td>
<td>cblas_dzdotc_sub</td>
</tr>
<tr>
<td>“SNRM2, DNRM2, SCNRM2, and DZNRM2 (Euclidean Length of a Vector with Scaling of Input to Avoid Destructive Underflow and Overflow)” on page 268</td>
<td>SNRM2</td>
<td>DNRM2</td>
</tr>
<tr>
<td></td>
<td>SCNRM2</td>
<td>DZNRM2</td>
</tr>
<tr>
<td></td>
<td>cblas_snrm2</td>
<td>cblas_dnrn2</td>
</tr>
<tr>
<td></td>
<td>cblas_scnrn2</td>
<td>cblas_dznrm2</td>
</tr>
<tr>
<td>“SROTG, DROTG, CROTG, and ZROTG (Construct a Givens Plane Rotation)” on page 274</td>
<td>SROTG</td>
<td>DROTG</td>
</tr>
<tr>
<td></td>
<td>CROTG</td>
<td>ZROTG</td>
</tr>
<tr>
<td></td>
<td>cblas_srotg</td>
<td>cblas_drotg</td>
</tr>
<tr>
<td></td>
<td>cblas_crotg</td>
<td>cblas_zrotg</td>
</tr>
<tr>
<td>“SROT, DROT, CROT, ZROT, CSROT, and ZDROT (Apply a Plane Rotation)” on page 280</td>
<td>SROT</td>
<td>DROT</td>
</tr>
<tr>
<td></td>
<td>CROT</td>
<td>ZROT</td>
</tr>
<tr>
<td></td>
<td>CSROT</td>
<td>ZDROT</td>
</tr>
<tr>
<td></td>
<td>cblas_srot</td>
<td>cblas_drot</td>
</tr>
<tr>
<td></td>
<td>cblas_crot</td>
<td>cblas_zrot</td>
</tr>
<tr>
<td></td>
<td>cblas_csrot</td>
<td>cblas_zdrot</td>
</tr>
<tr>
<td>“SROTMG and DROTMG (Construct a modified Givens Transformation)” on page 284</td>
<td>SROTMG</td>
<td>DROTMG</td>
</tr>
<tr>
<td></td>
<td>cblas_srotng</td>
<td>cblas_drotng</td>
</tr>
<tr>
<td>“SROTM and DROTM (Apply a modified Givens Transformation)” on page 287</td>
<td>SROTM</td>
<td>DROTM</td>
</tr>
<tr>
<td></td>
<td>cblas_srotm</td>
<td>cblas_drotn</td>
</tr>
</tbody>
</table>
### Table 248. Level 1 BLAS Included in ESSL (continued)

<table>
<thead>
<tr>
<th>Descriptive Name</th>
<th>Short-Precision Subprogram</th>
<th>Long-Precision Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>“SSCAL, DSCAL, CSCAL, ZSCAL, CSSCAL, and ZDSCAL (Multiply a Vector X by a Scalar and Store in the Vector X)” on page 290</td>
<td>SSCAL, CSCAL, CSSCAL, cblas_sscal, cblas-csscal</td>
<td>DSCAL, ZSCAL, ZDSCAL, cblas_dscal, cblas_zscal</td>
</tr>
<tr>
<td>“SSWAP, DSWAP, CSWAP, and ZSWAP (Interchange the Elements of Two Vectors)” on page 293</td>
<td>SSWAP, CSWAP, cblas_sswap, cblas_cswap</td>
<td>DSWAP, ZSWAP, cblas_dswap, cblas_zswap</td>
</tr>
</tbody>
</table>

### Level 2 BLAS

### Table 249. Level 2 BLAS Included in ESSL

<table>
<thead>
<tr>
<th>Descriptive Name</th>
<th>Short-Precision Subprogram</th>
<th>Long-Precision Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>“SGEMV, DGEMV, CGEMV, ZGEMV, SGEMX, DGEMX, SGEMTX, and DGEMTX (Matrix-Vector Product for a General Matrix, Its Transpose, or Its Conjugate Transpose)” on page 334</td>
<td>SGEMV, CGEMV, cblas_sgemv, cblas_cgemv</td>
<td>DGEMV, ZGEMV, cblas_dgemv, cblas_zgemv</td>
</tr>
<tr>
<td>“SGER, DGER, CGERU, ZGERU, CGERC, and ZGERC (Rank-One Update of a General Matrix)” on page 345</td>
<td>SGER, CGERU, CGERC, cblas_sger, cblas_cger, cblas_cgerc</td>
<td>DGER, ZGERU, ZGERC, cblas_dsger, cblas_zgeru, cblas_zgerc</td>
</tr>
<tr>
<td>“SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV, SSYMV, and DSLMX (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Matrix)” on page 353</td>
<td>SSPMV, CHPMV, SSYMV, CHEMV, cblas_sspmv, cblas_chpmv, cblas_ssymv, cblas_chemv</td>
<td>DSPMV, ZHPMV, DSYMV, ZHEMV, cblas_dspmv, cblas_zhpmv, cblas_dsymv, cblas_zhemv</td>
</tr>
<tr>
<td>“SSPR, DSPR, CHPR, ZHPR, SSYR, DSYR, CHER, ZHER, SSL1, and DSL1 (Rank-One Update of a Real Symmetric or Complex Hermitian Matrix)” on page 362</td>
<td>SSPR, CHPR, SSYR, CHER, cblas_sspr, cblas_chpr, cblas_ssymr, cblas_cher</td>
<td>DSPR, ZHPR, DSYR, ZHER, cblas_dspr, cblas_zhpr, cblas_dsymr, cblas_zher</td>
</tr>
<tr>
<td>“SSPR2, DSPR2, CHPR2, ZHPR2, SSYR2, DSYR2, CHER2, ZHER2, SSL2, and DSL2 (Rank-Two Update of a Real Symmetric or Complex Hermitian Matrix)” on page 370</td>
<td>SSPR2, CHPR2, SSYR2, CHER2, cblas_sspr2, cblas_chpr2, cblas_ssymr2, cblas_cher2</td>
<td>DSPR2, ZHPR2, DSYR2, ZHER2, cblas_dspr2, cblas_zhpr2, cblas_dsymr2, cblas_zher2</td>
</tr>
</tbody>
</table>
### Table 249. Level 2 BLAS Included in ESSL (continued)

<table>
<thead>
<tr>
<th>Descriptive Name</th>
<th>Short-Precision Subprogram</th>
<th>Long-Precision Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>“SGBMV, DGBMV, CGBMV, and ZGBMV (Matrix-Vector Product for a General Band Matrix, Its Transpose, or Its Conjugate Transpose)” on page 379</td>
<td>SGBMV</td>
<td>DGBMV</td>
</tr>
<tr>
<td></td>
<td>CGBMV</td>
<td>ZGBMV</td>
</tr>
<tr>
<td></td>
<td>cblas_sgbmv</td>
<td>cblas_dgbmv</td>
</tr>
<tr>
<td></td>
<td>cblas_cgbmv</td>
<td>cblas_zgbmv</td>
</tr>
<tr>
<td>“SSBMV, DSBMV, CHBMV, and ZHBMV (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Band Matrix)” on page 386</td>
<td>SSBMV</td>
<td>DSBMV</td>
</tr>
<tr>
<td></td>
<td>CHBMV</td>
<td>ZHBMV</td>
</tr>
<tr>
<td></td>
<td>cblas_ssbmv</td>
<td>cblas_dsbmv</td>
</tr>
<tr>
<td></td>
<td>cblas_chbmv</td>
<td>cblas_zhbmv</td>
</tr>
<tr>
<td>“STRMV, DTRMV, CTRMV, ZTRMV, STPMV, DTPMV, CTPMV, and ZTPMV (Matrix-Vector Product for a Triangular Matrix, Its Transpose, or Its Conjugate Transpose)” on page 391</td>
<td>STPMV</td>
<td>DTPMV</td>
</tr>
<tr>
<td></td>
<td>CTPMV</td>
<td>ZTPMV</td>
</tr>
<tr>
<td></td>
<td>STRMV</td>
<td>DTRMV</td>
</tr>
<tr>
<td></td>
<td>CTRMV</td>
<td>ZTRMV</td>
</tr>
<tr>
<td></td>
<td>cblas_stpmv</td>
<td>cblas_dtpmv</td>
</tr>
<tr>
<td></td>
<td>cblas_ctpmv</td>
<td>cblas_ztpmv</td>
</tr>
<tr>
<td></td>
<td>cblas_strmv</td>
<td>cblas_dtrmv</td>
</tr>
<tr>
<td></td>
<td>cblas_ctrmv</td>
<td>cblas_ztrmv</td>
</tr>
<tr>
<td>“STRSV, DTRSV, CTRSV, ZTRSV, STPSV, DTPSV, CTPSV, and ZTPSV (Solution of a Triangular System of Equations with a Single Right-Hand Side)” on page 398</td>
<td>STPSV</td>
<td>DTPSV</td>
</tr>
<tr>
<td></td>
<td>CTPSV</td>
<td>ZTPSV</td>
</tr>
<tr>
<td></td>
<td>STRSV</td>
<td>DTRSV</td>
</tr>
<tr>
<td></td>
<td>CTRSV</td>
<td>ZTRSV</td>
</tr>
<tr>
<td></td>
<td>cblas_stpsv</td>
<td>cblas_dtpsv</td>
</tr>
<tr>
<td></td>
<td>cblas_ctpsv</td>
<td>cblas_ztpsv</td>
</tr>
<tr>
<td></td>
<td>cblas_strsv</td>
<td>cblas_dtrsv</td>
</tr>
<tr>
<td></td>
<td>cblas_ctrsv</td>
<td>cblas_ztrsv</td>
</tr>
</tbody>
</table>
### Level 3 BLAS

### Table 250. Level 3 BLAS Included in ESSL

<table>
<thead>
<tr>
<th>Descriptive Name</th>
<th>Short-Precision Subprogram</th>
<th>Long-Precision Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>“SGEMM, DGEMM, CGEMM, and ZGEMM (Combined Matrix Multiplication and Addition for General Matrices, Their Transposes, or Conjugate Transposes)” on page 461</td>
<td>SGEMM</td>
<td>DGEMM</td>
</tr>
<tr>
<td></td>
<td>CGEMM</td>
<td>ZGEMM</td>
</tr>
<tr>
<td></td>
<td>cblas_sgemm</td>
<td>cblas_dgemm</td>
</tr>
<tr>
<td></td>
<td>cblas_cgemm</td>
<td>cblas_zgemm</td>
</tr>
<tr>
<td>“SSYMM, DSYMM, CSYMM, ZSYMM, CHEMM, and ZHEMM (Matrix-Matrix Product Where One Matrix is Real or Complex Symmetric or Complex Hermitian)” on page 470</td>
<td>SSYMM</td>
<td>DSYMM</td>
</tr>
<tr>
<td></td>
<td>CSYMM</td>
<td>ZSYMM</td>
</tr>
<tr>
<td></td>
<td>CHEMM</td>
<td>ZHEMM</td>
</tr>
<tr>
<td></td>
<td>cblas_ssymm</td>
<td>cblas_dsymm</td>
</tr>
<tr>
<td></td>
<td>cblas_csymm</td>
<td>cblas_zsymm</td>
</tr>
<tr>
<td></td>
<td>cblas_chemm</td>
<td>cblas_zhemm</td>
</tr>
<tr>
<td>“STRMM, DTRMM, CTRMM, and ZTRMM (Triangular Matrix-Matrix Product)” on page 478</td>
<td>STRMM</td>
<td>DTRMM</td>
</tr>
<tr>
<td></td>
<td>CTRMM</td>
<td>ZTRMM</td>
</tr>
<tr>
<td></td>
<td>cblas_strmm</td>
<td>cblas_dtrmm</td>
</tr>
<tr>
<td></td>
<td>cblas_ctrm</td>
<td>cblas_ztrmm</td>
</tr>
</tbody>
</table>
### Table 250. Level 3 BLAS Included in ESSL (continued)

<table>
<thead>
<tr>
<th>Descriptive Name</th>
<th>Short-Precision Subprogram</th>
<th>Long-Precision Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>“STRSM, DTRSM, CTRSM, and ZTRSM (Solution of Triangular Systems of Equations with Multiple Right-Hand Sides)” on page 486</td>
<td>STRSM</td>
<td>DTRSM</td>
</tr>
<tr>
<td></td>
<td>CTRSM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cblas_strsm</td>
<td>cblas_dtrsm</td>
</tr>
<tr>
<td></td>
<td>cblas_ctrsm</td>
<td>cblas_ztrsm</td>
</tr>
<tr>
<td>“SSYRK, DSYRK, CSYRK, ZSYRK, CHERK, and ZHERK (Rank-K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix)” on page 494</td>
<td>SSYRK</td>
<td>DSYRK</td>
</tr>
<tr>
<td></td>
<td>CSYRK</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CHERK</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cblas_ssyrk</td>
<td>cblas_dsyrk</td>
</tr>
<tr>
<td></td>
<td>cblas_csyrk</td>
<td>cblas_zsyrk</td>
</tr>
<tr>
<td></td>
<td>cblas_cherk</td>
<td>cblas_zherk</td>
</tr>
<tr>
<td>“SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, and ZHER2K (Rank-2K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix)” on page 501</td>
<td>SSYR2K</td>
<td>DSYR2K</td>
</tr>
<tr>
<td></td>
<td>CSYR2K</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CHER2K</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cblas_ssyr2k</td>
<td>cblas_dsyr2k</td>
</tr>
<tr>
<td></td>
<td>cblas_csyr2k</td>
<td>cblas_zsyr2k</td>
</tr>
<tr>
<td></td>
<td>cblas_cher2k</td>
<td>cblas_zher2k</td>
</tr>
</tbody>
</table>

**Note:** xerbla is not used by ESSL but is included in the ESSL libraries for use in other programming environments (for example, Python).

### CBLAS Subroutines

Note the following about the CBLAS subroutines:

- The definitions of the CBLAS data types and prototypes are included in the ESSL header files. In the beginning of your program, before you call any of the CBLAS subroutines, you must code the following statement for the ESSL header file:

  ```c
  #include <essl.h>
  ```

- CBLAS enumerated types have changed to type definitions for ESSL Version 5 Release 5 and changes are required if you used enumerated types. See “Migrating Programs from ESSL for Linux on Power Version 5 Release 4 to Version 5 Release 5” on page 201 for the required changes to your existing programs.

- For information about CBLAS calling sequences, see “Syntax” on page xxvi and [10 on page 1364](#).
## Appendix B. LAPACK and LAPACKE

The following table lists the ESSL subroutines corresponding to subroutines in the standard set of LAPACK and LAPACKE.

<table>
<thead>
<tr>
<th>Short-Precision Subprogram</th>
<th>Long-Precision Subprogram</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGESV</td>
<td>DGESV</td>
<td>&quot;SGESV, DGESV, CGESV, ZGESV (General Matrix Factorization and Multiple Right-Hand Side Solve)&quot; on page 550</td>
</tr>
<tr>
<td>CGESV</td>
<td>ZGESV</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgesv</td>
<td>LAPACKE_dgesv</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgesv</td>
<td>LAPACKE_zgesv</td>
<td></td>
</tr>
<tr>
<td>SGETF</td>
<td>DGETRF</td>
<td>&quot;SGETRF, DGETRF, CGETRF and ZGETRF (General Matrix Factorization)&quot; on page 534</td>
</tr>
<tr>
<td>CGETF</td>
<td>ZGETRF</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgetrf</td>
<td>LAPACKE_dgetrf</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgetrf</td>
<td>LAPACKE_zgetrf</td>
<td></td>
</tr>
<tr>
<td>SGETRS</td>
<td>DGETRS</td>
<td>&quot;SGETRS, DGETRS, CGETRS, and ZGETRS (General Matrix Multiple Right-Hand Side Solve)&quot; on page 539</td>
</tr>
<tr>
<td>CGETRS</td>
<td>ZGETRS</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgetrs</td>
<td>LAPACKE_dgetrs</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgetrs</td>
<td>LAPACKE_zgetrs</td>
<td></td>
</tr>
<tr>
<td>SGECON</td>
<td>DGECON</td>
<td>&quot;SGECON, DGECON, CGECON, and ZGECON (Estimate the Reciprocal of the Condition Number of a General Matrix)&quot; on page 556</td>
</tr>
<tr>
<td>CGECON</td>
<td>ZGECON</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgecon</td>
<td>LAPACKE_dgecon</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgecon</td>
<td>LAPACKE_zgecon</td>
<td></td>
</tr>
<tr>
<td>SGETRI</td>
<td>DGETRI</td>
<td>&quot;SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD (General Matrix Inverse, Condition Number Reciprocal, and Determinant)&quot; on page 565</td>
</tr>
<tr>
<td>CGETRI</td>
<td>ZGETRI</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgetri</td>
<td>LAPACKE_dgetri</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgetri</td>
<td>LAPACKE_zgetri</td>
<td></td>
</tr>
<tr>
<td>SLANGE</td>
<td>DLANGE</td>
<td>&quot;SLANGE, DLANGE, CLANGE, and ZLANGE (General Matrix Norm)&quot; on page 572</td>
</tr>
<tr>
<td>CLANGE</td>
<td>ZLANGE</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_slange</td>
<td>LAPACKE_dlange</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_clange</td>
<td>LAPACKE_zlange</td>
<td></td>
</tr>
<tr>
<td>SPPSV</td>
<td>DPPSV</td>
<td>&quot;SPPSV, DPSSV, CPPSV, and ZPPSV (Positive Definite Real Symmetric and Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)&quot; on page 575</td>
</tr>
<tr>
<td>CPPSV</td>
<td>ZPPSV</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sppsv</td>
<td>LAPACKE_dpssv</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cppsv</td>
<td>LAPACKE_zppsv</td>
<td></td>
</tr>
<tr>
<td>SPOSV</td>
<td>DPOSV</td>
<td>&quot;SPOSV, DPOSV, CPSOV, and ZPOSV (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)&quot; on page 581</td>
</tr>
<tr>
<td>CPSOV</td>
<td>ZPOSV</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sposv</td>
<td>LAPACKE_dposv</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cposv</td>
<td>LAPACKE_zposv</td>
<td></td>
</tr>
<tr>
<td>SPOTRF</td>
<td>DPOTRF</td>
<td>&quot;SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPFF, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)&quot; on page 587</td>
</tr>
<tr>
<td>CPOTRF</td>
<td>ZPOTRF</td>
<td></td>
</tr>
<tr>
<td>SPPTRF</td>
<td>DPPTRF</td>
<td></td>
</tr>
<tr>
<td>CPPTRF</td>
<td>ZPPTRF</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spotrpf</td>
<td>LAPACKE_dpotrf</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpotrf</td>
<td>LAPACKE_zpotrf</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spptrf</td>
<td>LAPACKE_dpptrf</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpptrf</td>
<td>LAPACKE_zpptrf</td>
<td></td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 1986, 2016
## Table 251. LAPACK and LAPACKE subroutines included in ESSL (continued)

<table>
<thead>
<tr>
<th>Short-Precision Subprogram</th>
<th>Long-Precision Subprogram</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPOTRS</td>
<td>DPOTRS</td>
<td>&quot;SPOTRS, DPOTRS, CPOTRS, ZPOTRS, SSPTR, DPPTR, CPPTTR, and ZZPPTR (Positive Definite Real Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)&quot; on page 600</td>
</tr>
<tr>
<td>CPOTRS</td>
<td>ZPOTRS</td>
<td></td>
</tr>
<tr>
<td>SPPTRS</td>
<td>DPPTR</td>
<td></td>
</tr>
<tr>
<td>CPPTRS</td>
<td>ZZPPTR</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spotrs</td>
<td>LAPACKE_dpotrs</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpotrs</td>
<td>LAPACKE_zpotrs</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spptrs</td>
<td>LAPACKE_dpptrs</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpptrs</td>
<td>LAPACKE_zpptrs</td>
<td></td>
</tr>
<tr>
<td>SPOCON</td>
<td>DPOCON</td>
<td>&quot;SPOCON, DPOCON, CPPOCON, ZPOCON, SPOCON, DPOCON, DPPOC, and SPPOCD (Estimate the Reciprocal of the Condition Number of a Positive Definite Real Symmetric or Complex Hermitian Matrix)&quot; on page 611</td>
</tr>
<tr>
<td>CPOCON</td>
<td>ZPOCON</td>
<td></td>
</tr>
<tr>
<td>SPPOCON</td>
<td>DPPCON</td>
<td></td>
</tr>
<tr>
<td>CPPCON</td>
<td>ZZPPCON</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spocon</td>
<td>LAPACKE_dpocon</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpocon</td>
<td>LAPACKE_zpocon</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spcon</td>
<td>LAPACKE_dpcon</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cppcon</td>
<td>LAPACKE_zppcon</td>
<td></td>
</tr>
<tr>
<td>SPOTRI</td>
<td>DPOTRI</td>
<td>&quot;SPOTRI, DPOTRI, CPOTRI, ZPOTRI, SSPTR, DPPTR, CPPTTR, and ZZPPTR (Positive Definite Real Symmetric or Complex Hermitian Matrix Inverse, Condition Number Reciprocal, and Determinant)&quot; on page 626</td>
</tr>
<tr>
<td>CPOTRI</td>
<td>ZPOTRI</td>
<td></td>
</tr>
<tr>
<td>SPPTRI</td>
<td>DPPTR</td>
<td></td>
</tr>
<tr>
<td>CPPTRI</td>
<td>ZZPPTR</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spotri</td>
<td>LAPACKE_dpotri</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpotri</td>
<td>LAPACKE_zpotri</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spptri</td>
<td>LAPACKE_dpptri</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpptri</td>
<td>LAPACKE_zpptri</td>
<td></td>
</tr>
<tr>
<td>SLANSY</td>
<td>DLANSY</td>
<td>&quot;SLANSY, DLANSY, CLANHE, ZLANHE, SLANSP, DLANSP (Real Symmetric or Complex Hermitian Matrix Norm)&quot; on page 637</td>
</tr>
<tr>
<td>CLANHE</td>
<td>ZLANHE</td>
<td></td>
</tr>
<tr>
<td>SLANSP</td>
<td>DLANSP</td>
<td></td>
</tr>
<tr>
<td>CLANH</td>
<td>ZLANH</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_slansy</td>
<td>LAPACKE_dlanhs</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_clanhe</td>
<td>LAPACKE_zlanhe</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_slansp</td>
<td>LAPACKE_dlansp</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_clanhp</td>
<td>LAPACKE_zlanhp</td>
<td></td>
</tr>
<tr>
<td>SSYSV</td>
<td>DSYSV</td>
<td>&quot;SSYSV, DSYSV, CSYSV, ZSYSV, CHESV, ZHESV, SSSPSV, DSSPSV, CSPSV, ZSPSV, CHPSV, and ZHPSV (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)&quot; on page 642</td>
</tr>
<tr>
<td>CSYSV</td>
<td>ZSYSV</td>
<td></td>
</tr>
<tr>
<td>CHESV</td>
<td>ZHESV</td>
<td></td>
</tr>
<tr>
<td>SSSPSV</td>
<td>DSSPSV</td>
<td></td>
</tr>
<tr>
<td>CSPSV</td>
<td>ZSSPSV</td>
<td></td>
</tr>
<tr>
<td>CHPSV</td>
<td>ZCHPSV</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssysv</td>
<td>LAPACKE_dsysv</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_csysv</td>
<td>LAPACKE_zsysv</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chesv</td>
<td>LAPACKE_zhesv</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sspsv</td>
<td>LAPACKE_dspsv</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cspsv</td>
<td>LAPACKE_zspsv</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chspsv</td>
<td>LAPACKE_zhpsv</td>
<td></td>
</tr>
<tr>
<td>Short-Precision Subprogram</td>
<td>Long-Precision Subprogram</td>
<td>Descriptive Name and Location</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>SSYTF</td>
<td>DSYTF</td>
<td><strong>SSYTF, DSYTF, CSYTF, ZSYTF, CHETF, ZHETF, SSPTF, DSPTF, CSPTF, ZSPTF, and ZHPTRF (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Factorization)</strong> on page 651</td>
</tr>
<tr>
<td>CSYTF</td>
<td>ZSYTF</td>
<td></td>
</tr>
<tr>
<td>CHETF</td>
<td>ZHETF</td>
<td></td>
</tr>
<tr>
<td>SSPTRF</td>
<td>DSPTRF</td>
<td></td>
</tr>
<tr>
<td>CSPTRF</td>
<td>ZSPTRF</td>
<td></td>
</tr>
<tr>
<td>CHPTRF</td>
<td>ZHPTRF</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssytf</td>
<td>LAPACKE_dsytf</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_csytf</td>
<td>LAPACKE_zsytf</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chetf</td>
<td>LAPACKE_zhetf</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssptrf</td>
<td>LAPACKE_dsptf</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_csptrf</td>
<td>LAPACKE_zsptrf</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chptrf</td>
<td>LAPACKE_zhptrf</td>
<td></td>
</tr>
<tr>
<td>SSYTRF</td>
<td>DSYTF</td>
<td><strong>SSYTF, DSYTF, CSYTF, ZSYTF, CHETF, ZHETF, SSPTF, DSPTF, CSPTF, ZSPTF, and ZHPTRF (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)</strong> on page 660</td>
</tr>
<tr>
<td>CSYTRF</td>
<td>ZSYTF</td>
<td></td>
</tr>
<tr>
<td>CHETR</td>
<td>ZHETR</td>
<td></td>
</tr>
<tr>
<td>SSPTRS</td>
<td>DSPTRS</td>
<td></td>
</tr>
<tr>
<td>CSPTRS</td>
<td>ZSPTRS</td>
<td></td>
</tr>
<tr>
<td>CHPTRS</td>
<td>ZHPTRS</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssytrs</td>
<td>LAPACKE_dsytrs</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_csytrs</td>
<td>LAPACKE_zsytrs</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chetr</td>
<td>LAPACKE_zhetr</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssptrs</td>
<td>LAPACKE_dsptrs</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_csptrs</td>
<td>LAPACKE_zsptrs</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chptrs</td>
<td>LAPACKE_zhptrs</td>
<td></td>
</tr>
<tr>
<td>STRTRI</td>
<td>DTRTRI</td>
<td><strong>STRTRI, DTRTRI, CTRTRI, ZTRTRI, STPTRI, DTPTRI, CTPTRI, and ZTPTRI (Triangular Matrix Inverse)</strong> on page 682</td>
</tr>
<tr>
<td>STPTRI</td>
<td>DTPTRI</td>
<td></td>
</tr>
<tr>
<td>CTRTRI</td>
<td>ZTRTRI</td>
<td></td>
</tr>
<tr>
<td>CTPTRI</td>
<td>ZTPTRI</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_strtri</td>
<td>LAPACKE_dtrtri</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_stptri</td>
<td>LAPACKE_dptri</td>
<td></td>
</tr>
<tr>
<td>LAPACKECtrtr</td>
<td>LAPACKE_ztrtri</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ctptri</td>
<td>LAPACKE_zptri</td>
<td></td>
</tr>
<tr>
<td>SLANTR</td>
<td>DLANTR</td>
<td><strong>SLANTR, DLANTR, CLANTR, ZLANTR, SLANTP, DLANTP, CLANTP, and ZLANTP (Trapezoidal or Triangular Matrix Norm)</strong> on page 690</td>
</tr>
<tr>
<td>CLANTR</td>
<td>ZLANTR</td>
<td></td>
</tr>
<tr>
<td>SLANTP</td>
<td>DLANTP</td>
<td></td>
</tr>
<tr>
<td>CLANTP</td>
<td>ZLANTP</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_slantr</td>
<td>LAPACKE_dlantr</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_clantr</td>
<td>LAPACKE_zlantr</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_slantrp</td>
<td>LAPACKE_dlantrp</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_clantrp</td>
<td>LAPACKE_zlantrp</td>
<td></td>
</tr>
<tr>
<td>SGBSV</td>
<td>DGBSV</td>
<td><strong>SGBSV, DGBSV, CGBSV, and ZGBSV (General Band Matrix Factorization and Multiple Right-Hand Side Solve)</strong> on page 698</td>
</tr>
<tr>
<td>CGBSV</td>
<td>ZGBSV</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgbsv</td>
<td>LAPACKE_dgbsv</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgbsv</td>
<td>LAPACKE_zgbsv</td>
<td></td>
</tr>
<tr>
<td>SGBTRF</td>
<td>DGBTRF</td>
<td><strong>SGBTRF, DGBTRF, CGBTRF, and ZGBTRF (General Band Matrix Factorization)</strong> on page 702</td>
</tr>
<tr>
<td>CGBTRF</td>
<td>ZGBTRF</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgbtrfs</td>
<td>LAPACKE_dgbtrfs</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgbtrfs</td>
<td>LAPACKE_zgbtrfs</td>
<td></td>
</tr>
<tr>
<td>SGBTRS</td>
<td>DGBTRS</td>
<td><strong>SGBTRS, DGBTRS, CGBTRS, and ZGBTRS (General Band Matrix Multiple Right-Hand Side Solve)</strong> on page 706</td>
</tr>
<tr>
<td>CGBTRS</td>
<td>ZGBTRS</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgbtrs</td>
<td>LAPACKE_dgbtrs</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgbtrs</td>
<td>LAPACKE_zgbtrs</td>
<td></td>
</tr>
<tr>
<td>Short-Precision Subprogram</td>
<td>Long-Precision Subprogram</td>
<td>Descriptive Name and Location</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>SPBSV</td>
<td>DPBSV</td>
<td>“SPBSV, DPBSV, CPBSV, and ZPBSV (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization and Multiple Right-Hand Side Solve)” on page 715</td>
</tr>
<tr>
<td>CPBSV</td>
<td>ZPBSV</td>
<td>“CPBSV, ZPBSV, CPBSV, and ZPBSV (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization)” on page 721</td>
</tr>
<tr>
<td>LAPACKE_spbsv</td>
<td>LAPACKE_dpbsv</td>
<td>“LAPACKE_spbsv, LAPACKE_dpbsv, and ZPBSV (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization)” on page 722</td>
</tr>
<tr>
<td>LAPACKE_cpbsv</td>
<td>LAPACKE_zpbsv</td>
<td>“LAPACKE_cpbsv, LAPACKE_zpbsv, and ZPBSV (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization)” on page 723</td>
</tr>
<tr>
<td>SPBTRF</td>
<td>DPBTRF</td>
<td>“SPBTRF, DPBTRF, CPBTRF, and ZPBTRF (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization)” on page 721</td>
</tr>
<tr>
<td>CPBTRF</td>
<td>ZPBTRF</td>
<td>“CPBTRF, ZPBTRF, CPBTRF, and ZPBTRF (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Multiple Right-Hand Side Solve)” on page 726</td>
</tr>
<tr>
<td>LAPACKE_spbtrf</td>
<td>LAPACKE_dpbtrf</td>
<td>“LAPACKE_spbtrf, LAPACKE_dpbtrf, and ZPBTRF (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Multiple Right-Hand Side Solve)” on page 726</td>
</tr>
<tr>
<td>LAPACKE_cpbtrf</td>
<td>LAPACKE_zpbtrf</td>
<td>“LAPACKE_cpbtrf, LAPACKE_zpbtrf, and ZPBTRF (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Multiple Right-Hand Side Solve)” on page 726</td>
</tr>
<tr>
<td>SGTSV</td>
<td>DGTSV</td>
<td>“SGTSV, DGTSV, CGTSV, and ZGTSV (General Tridiagonal Matrix Factorization and Multiple Right-Hand Side Solve)” on page 731</td>
</tr>
<tr>
<td>CGTSV</td>
<td>ZGTSV</td>
<td>“CGTSV, ZGTSV, CGTSV, and ZGTSV (General Tridiagonal Matrix Factorization)” on page 735</td>
</tr>
<tr>
<td>LAPACKE_sgtsv</td>
<td>LAPACKE_dgtsv</td>
<td>“LAPACKE_sgtsv, LAPACKE_dgtsv, and ZGTSV (General Tridiagonal Matrix Factorization)” on page 735</td>
</tr>
<tr>
<td>LAPACKE_cgtsv</td>
<td>LAPACKE_zgtsv</td>
<td>“LAPACKE_cgtsv, LAPACKE_zgtsv, and ZGTSV (General Tridiagonal Matrix Factorization)” on page 735</td>
</tr>
<tr>
<td>SGTTRF</td>
<td>DGTTTRF</td>
<td>“SGTTRF, DGTTTRF, CGTTRF, and ZGTTTRF (General Tridiagonal Matrix Factorization)” on page 735</td>
</tr>
<tr>
<td>CGTTRF</td>
<td>ZGTTTRF</td>
<td>“CGTTRF, ZGTTTRF, CGTTRF, and ZGTTTRF (General Tridiagonal Matrix Factorization)” on page 735</td>
</tr>
<tr>
<td>LAPACKE_sgttrf</td>
<td>LAPACKE_dgttrf</td>
<td>“LAPACKE_sgttrf, LAPACKE_dgttrf, and ZGTTTRF (General Tridiagonal Matrix Factorization)” on page 735</td>
</tr>
<tr>
<td>LAPACKE_cgttrf</td>
<td>LAPACKE_zgttrf</td>
<td>“LAPACKE_cgttrf, LAPACKE_zgttrf, and ZGTTTRF (General Tridiagonal Matrix Factorization)” on page 735</td>
</tr>
<tr>
<td>SPTSV</td>
<td>DPTSV</td>
<td>“SPTSV, DPTSV, CPTSV, and ZPTSV (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization and Multiple Right-Hand Side Solve)” on page 746</td>
</tr>
<tr>
<td>CPTSV</td>
<td>ZPTSV</td>
<td>“CPTSV, ZPTSV, CPTSV, and ZPTSV (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization)” on page 751</td>
</tr>
<tr>
<td>LAPACKE_sptsv</td>
<td>LAPACKE_dptsv</td>
<td>“LAPACKE_sptsv, LAPACKE_dptsv, and ZPTSV (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization)” on page 751</td>
</tr>
<tr>
<td>LAPACKE_cptsv</td>
<td>LAPACKE_zptsv</td>
<td>“LAPACKE_cptsv, LAPACKE_zptsv, and ZPTSV (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization)” on page 751</td>
</tr>
<tr>
<td>SPTTRF</td>
<td>DPTTRF</td>
<td>“SPTTRF, DPTTRF, CPTTRF, and ZPTTRF (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization)” on page 751</td>
</tr>
<tr>
<td>CPTTRF</td>
<td>ZPTTRF</td>
<td>“CPTTRF, ZPTTRF, CPTTRF, and ZPTTRF (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization)” on page 751</td>
</tr>
<tr>
<td>LAPACKE_spttrf</td>
<td>LAPACKE_dpttrf</td>
<td>“LAPACKE_spttrf, LAPACKE_dpttrf, and ZPTTRF (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization)” on page 751</td>
</tr>
<tr>
<td>LAPACKE_cpttrf</td>
<td>LAPACKE_zpttrf</td>
<td>“LAPACKE_cpttrf, LAPACKE_zpttrf, and ZPTTRF (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization)” on page 751</td>
</tr>
<tr>
<td>SGEVSVD</td>
<td>DGESVD</td>
<td>“SGEV, DGESV, CGEV, ZGESV, DGESV, ZGESV, CGEV, ZGESV, DGESV, and ZGESV (Singular Value Decomposition for a General Matrix)” on page 881</td>
</tr>
<tr>
<td>CGEVSV</td>
<td>ZGESVD</td>
<td>“CGEV, ZGESV, CGEV, ZGESV, DGESV, ZGESV, CGEV, ZGESV, DGESV, and ZGESV (Singular Value Decomposition for a General Matrix)” on page 881</td>
</tr>
<tr>
<td>SGESDD</td>
<td>DGESDD</td>
<td>“SGEV, DGESV, CGEV, ZGESV, DGESV, ZGESV, CGEV, ZGESV, DGESV, and ZGESV (Singular Value Decomposition for a General Matrix)” on page 881</td>
</tr>
<tr>
<td>CGESDD</td>
<td>ZGESDD</td>
<td>“CGEV, ZGESV, CGEV, ZGESV, DGESV, ZGESV, CGEV, ZGESV, DGESV, and ZGESV (Singular Value Decomposition for a General Matrix)” on page 881</td>
</tr>
<tr>
<td>LAPACKE_sgesvd</td>
<td>LAPACKE_dgesvd</td>
<td>“LAPACKE_sgesvd, LAPACKE_dgesvd, and ZGESV (Singular Value Decomposition for a General Matrix)” on page 881</td>
</tr>
<tr>
<td>LAPACKE_cgesvd</td>
<td>LAPACKE_zgesvd</td>
<td>“LAPACKE_cgesvd, LAPACKE_zgesvd, and ZGESV (Singular Value Decomposition for a General Matrix)” on page 881</td>
</tr>
<tr>
<td>LAPACKE_sgesdd</td>
<td>LAPACKE_dgesdd</td>
<td>“LAPACKE_sgesdd, LAPACKE_dgesdd, and ZGESV (Singular Value Decomposition for a General Matrix)” on page 881</td>
</tr>
<tr>
<td>LAPACKE_cgesdd</td>
<td>LAPACKE_zgesdd</td>
<td>“LAPACKE_cgesdd, LAPACKE_zgesdd, and ZGESV (Singular Value Decomposition for a General Matrix)” on page 881</td>
</tr>
<tr>
<td>SGEQRF</td>
<td>DGGEQRF</td>
<td>“SGEQRF, DGGEQRF, CGEQR, and ZGEQRF (General Matrix QR Factorization)” on page 895</td>
</tr>
<tr>
<td>CGGEQRF</td>
<td>ZGEQRF</td>
<td>“CGGEQRF, ZGEQRF, CGEQR, and ZGEQRF (General Matrix QR Factorization)” on page 895</td>
</tr>
<tr>
<td>LAPACKE_sgeqrf</td>
<td>LAPACKE_dgeqrf</td>
<td>“LAPACKE_sgeqrf, LAPACKE_dgeqrf, and ZGEQRF (General Matrix QR Factorization)” on page 895</td>
</tr>
<tr>
<td>LAPACKE_cgeqrf</td>
<td>LAPACKE_zgeqrf</td>
<td>“LAPACKE_cgeqrf, LAPACKE_zgeqrf, and ZGEQRF (General Matrix QR Factorization)” on page 895</td>
</tr>
<tr>
<td>SGELS</td>
<td>DGELS</td>
<td>“SGELS, DGELS, CGELS, and ZGELS (Linear Least Squares Solution for a General Matrix)” on page 901</td>
</tr>
<tr>
<td>CGELS</td>
<td>ZGELS</td>
<td>“CGELS, ZGELS, CGELS, and ZGELS (Linear Least Squares Solution for a General Matrix)” on page 901</td>
</tr>
<tr>
<td>LAPACKE_sgels</td>
<td>LAPACKE_dgels</td>
<td>“LAPACKE_sgels, LAPACKE_dgels, and ZGELS (Linear Least Squares Solution for a General Matrix)” on page 901</td>
</tr>
<tr>
<td>LAPACKE_cgels</td>
<td>LAPACKE_zgels</td>
<td>“LAPACKE_cgels, LAPACKE_zgels, and ZGELS (Linear Least Squares Solution for a General Matrix)” on page 901</td>
</tr>
</tbody>
</table>
Table 251. LAPACK and LAPACKE subroutines included in ESSL (continued)

<table>
<thead>
<tr>
<th>Short-Precision Subprogram</th>
<th>Long-Precision Subprogram</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-Precision Subprogram</td>
<td>Long-Precision Subprogram</td>
<td>Descriptive Name and Location</td>
</tr>
<tr>
<td>Short-Precision Subprogram</td>
<td>Long-Precision Subprogram</td>
<td>Descriptive Name and Location</td>
</tr>
<tr>
<td>Short-Precision Subprogram</td>
<td>Long-Precision Subprogram</td>
<td>Descriptive Name and Location</td>
</tr>
<tr>
<td>Short-Precision Subprogram</td>
<td>Long-Precision Subprogram</td>
<td>Descriptive Name and Location</td>
</tr>
<tr>
<td>Short-Precision Subprogram</td>
<td>Long-Precision Subprogram</td>
<td>Descriptive Name and Location</td>
</tr>
<tr>
<td>Short-Precision Subprogram</td>
<td>Long-Precision Subprogram</td>
<td>Descriptive Name and Location</td>
</tr>
</tbody>
</table>

LAPACK Subroutines

Note the following about the LAPACKE subroutines:

- ESSL includes only the high level C Interface to LAPACK.
- The definitions of the LAPACKE data types and prototypes are included in the ESSL header files. In the beginning of your program, before you call any of the LAPACKE subroutines, you must code the following statement for the ESSL header file:
  
  ```
  #include <essl.h>
  ```

- The optional LAPACKE NaN checking is not supported.

- ESSL supports only C99 complex data types for C programs and C99 or STL complex data types for C++ programs. There is no support provided for custom complex data types. The default for ESSL is to use C99 complex data types for C programs and STL complex data types for C++ programs. If you wish to use C99 data types for C++ programs code the `#include` statement for the C99 complex floating point types (`#include <complex.h>`) in your program prior to coding the `#include` statement for the ESSL header file.

- For information about LAPACKE calling sequences, see "Syntax" on page xxvi.
Appendix C. FFTW Version 3.1.2 to ESSL Wrapper Libraries

This appendix lists the FFTW Version 3.1.2 wrappers that can be used for calling functions from the ESSL libraries.

Documentation for FFTW Version 3.1.2 can be found at the following URL:

http://www.fftw.org

Additional information about the FFTW Wrapper libraries can be found in the following files:

AIX  /usr/lpp/essl.rte.common/FFTW3/README
Linux  /opt/ibmmath/essl/version.release/FFTW3/README

C and Fortran Wrappers

The following tables list the available C and Fortran wrappers.

**Table 252. List of available C and Fortran wrappers.**

<table>
<thead>
<tr>
<th>Category</th>
<th>C Wrapper</th>
<th>Fortran Wrapper</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plan usage</td>
<td><code>fftw_execute</code></td>
<td><code>DFFTW_EXECUTE</code></td>
</tr>
<tr>
<td></td>
<td><code>fftuf_execute</code></td>
<td><code>SFFTW_EXECUTE</code></td>
</tr>
<tr>
<td></td>
<td><code>fftw_destroy_plan</code></td>
<td><code>DFFTW_DESTROY_PLAN</code></td>
</tr>
<tr>
<td></td>
<td><code>fftuf_destroy_plan</code></td>
<td><code>SFFTW_DESTROY_PLAN</code></td>
</tr>
<tr>
<td></td>
<td><code>fftw_cleanup</code></td>
<td><code>DFFTW_CLEANUP</code></td>
</tr>
<tr>
<td></td>
<td><code>fftuf_cleanup</code></td>
<td><code>SFFTW_CLEANUP</code></td>
</tr>
<tr>
<td>Basic interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Complex DFTs)</td>
<td><code>fftw_plan_dft_1d</code></td>
<td><code>DFFTW_PLAN_DFT_1D</code></td>
</tr>
<tr>
<td></td>
<td><code>fftw_plan_dft_1d</code></td>
<td><code>SFFTW_PLAN_DFT_1D</code></td>
</tr>
<tr>
<td></td>
<td><code>fftw_plan_dft_2d</code></td>
<td><code>DFFTW_PLAN_DFT_2D</code></td>
</tr>
<tr>
<td></td>
<td><code>fftw_plan_dft_2d</code></td>
<td><code>SFFTW_PLAN_DFT_2D</code></td>
</tr>
<tr>
<td></td>
<td><code>fftw_plan_dft_3d</code></td>
<td><code>DFFTW_PLAN_DFT_3D</code></td>
</tr>
<tr>
<td></td>
<td><code>fftw_plan_dft_3d</code></td>
<td><code>SFFTW_PLAN_DFT_3D</code></td>
</tr>
<tr>
<td></td>
<td><code>fftw_plan_dft</code></td>
<td><code>DFFTW_PLAN_DFT</code></td>
</tr>
<tr>
<td></td>
<td><code>fftw_plan_dft</code></td>
<td><code>SFFTW_PLAN_DFT</code></td>
</tr>
<tr>
<td>Basic interface</td>
<td><code>fftw_plan_dft_r2c_1d</code></td>
<td><code>DFFTW_PLAN_DFT_R2C_1D</code></td>
</tr>
<tr>
<td>(Real-data DFTs)</td>
<td><code>fftw_plan_dft_r2c_1d</code></td>
<td><code>SFFTW_PLAN_DFT_R2C_1D</code></td>
</tr>
<tr>
<td></td>
<td><code>fftw_plan_dft_r2c_2d</code></td>
<td><code>DFFTW_PLAN_DFT_R2C_2D</code></td>
</tr>
<tr>
<td></td>
<td><code>fftw_plan_dft_r2c_2d</code></td>
<td><code>SFFTW_PLAN_DFT_R2C_2D</code></td>
</tr>
<tr>
<td></td>
<td><code>fftw_plan_dft_r2c_3d</code></td>
<td><code>DFFTW_PLAN_DFT_R2C_3D</code></td>
</tr>
<tr>
<td></td>
<td><code>fftw_plan_dft_r2c_3d</code></td>
<td><code>SFFTW_PLAN_DFT_R2C_3D</code></td>
</tr>
<tr>
<td></td>
<td><code>fftw_plan_dft_r2c</code></td>
<td><code>DFFTW_PLAN_DFT_R2C</code></td>
</tr>
<tr>
<td></td>
<td><code>fftw_plan_dft_r2c</code></td>
<td><code>SFFTW_PLAN_DFT_R2C</code></td>
</tr>
<tr>
<td></td>
<td><code>fftw_plan_dft_c2r_1d</code></td>
<td><code>DFFTW_PLAN_DFT_C2R_1D</code></td>
</tr>
<tr>
<td></td>
<td><code>fftw_plan_dft_c2r_1d</code></td>
<td><code>SFFTW_PLAN_DFT_C2R_1D</code></td>
</tr>
<tr>
<td></td>
<td><code>fftw_plan_dft_c2r_2d</code></td>
<td><code>DFFTW_PLAN_DFT_C2R_2D</code></td>
</tr>
<tr>
<td></td>
<td><code>fftw_plan_dft_c2r_2d</code></td>
<td><code>SFFTW_PLAN_DFT_C2R_2D</code></td>
</tr>
<tr>
<td></td>
<td><code>fftw_plan_dft_c2r_3d</code></td>
<td><code>DFFTW_PLAN_DFT_C2R_3D</code></td>
</tr>
<tr>
<td></td>
<td><code>fftw_plan_dft_c2r_3d</code></td>
<td><code>SFFTW_PLAN_DFT_C2R_3D</code></td>
</tr>
<tr>
<td></td>
<td><code>fftw_plan_dft_c2r</code></td>
<td><code>DFFTW_PLAN_DFT_C2R</code></td>
</tr>
<tr>
<td></td>
<td><code>fftw_plan_dft_c2r</code></td>
<td><code>SFFTW_PLAN_DFT_C2R</code></td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 1986, 2016

1349
Table 252. List of available C and Fortran wrappers (continued).

<table>
<thead>
<tr>
<th>Category</th>
<th>C Wrapper</th>
<th>Fortran Wrapper</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advanced interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Complex DFTs)</td>
<td>fftw_plan_many_dft</td>
<td>DFFTW_PLAN_MANY_DFT</td>
</tr>
<tr>
<td></td>
<td>fftwf_plan_many_dft</td>
<td>SFFTW_PLAN_MANY_DFT</td>
</tr>
<tr>
<td>Advanced interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Real-data DFTs)</td>
<td>fftw_plan_many_dft_r2c</td>
<td>DFFTW_PLAN_MANY_DFT_R2C</td>
</tr>
<tr>
<td></td>
<td>fftwf_plan_many_dft_r2c</td>
<td>SFFTW_PLAN_MANY_DFT_R2C</td>
</tr>
<tr>
<td></td>
<td>fftw_plan_many_dft_c2r</td>
<td>DFFTW_PLAN_MANY_DFT_C2R</td>
</tr>
<tr>
<td></td>
<td>fftwf_plan_many_dft_c2r</td>
<td>SFFTW_PLAN_MANY_DFT_C2R</td>
</tr>
<tr>
<td>Guru interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Complex DFTs)</td>
<td>fftw_plan_guru_dft</td>
<td>DFFTW_PLAN_GURU_DFT</td>
</tr>
<tr>
<td></td>
<td>fftwf_plan_guru_dft</td>
<td>SFFTW_PLAN_GURU_DFT</td>
</tr>
<tr>
<td></td>
<td>fftw_execute_dft</td>
<td>DFFTW_EXECUTE_DFT</td>
</tr>
<tr>
<td></td>
<td>fftwf_execute_dft</td>
<td>SFFTW_EXECUTE_DFT</td>
</tr>
<tr>
<td>Guru interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Real-data DFTs)</td>
<td>fftw_plan_guru_dft_r2c</td>
<td>DFFTW_PLAN_GURU_DFT_R2C</td>
</tr>
<tr>
<td></td>
<td>fftwf_plan_guru_dft_r2c</td>
<td>SFFTW_PLAN_GURU_DFT_R2C</td>
</tr>
<tr>
<td></td>
<td>fftw_plan_guru_dft_c2r</td>
<td>DFFTW_PLAN_GURU_DFT_C2R</td>
</tr>
<tr>
<td></td>
<td>fftwf_plan_guru_dft_c2r</td>
<td>SFFTW_PLAN_GURU_DFT_C2R</td>
</tr>
<tr>
<td></td>
<td>fftw_execute_dft_r2c</td>
<td>DFFTW_EXECUTE_DFT_R2C</td>
</tr>
<tr>
<td></td>
<td>fftwf_execute_dft_c2r</td>
<td>SFFTW_EXECUTE_DFT_C2R</td>
</tr>
<tr>
<td>Memory allocation</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fftw_malloc</td>
<td>(Not applicable)</td>
</tr>
<tr>
<td></td>
<td>fftwf_malloc</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fftw_free</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fftwf_free</td>
<td></td>
</tr>
</tbody>
</table>

Use of the ESSL FFTW Wrapper library has the following restrictions:
- No functions for real-to-real transforms are provided.
- No wrappers for the FFTW Wisdom functions are provided.
- The following flags are treated as equivalent: FFTW_ESTIMATE, FFTW_MEASURE, FFTW_PATIENT, FFTW_EXHAUSTIVE
- The FFTW Wrapper libraries use the same method to specify SMP parallelism as does ESSL instead of using the fftw_threads_init, fftw_plan_with_threads and fftw_cleanup_threads functions. These functions are not provided as part of the FFTW Wrapper libraries.

**Using the FFTW Wrapper libraries**

Applications using the FFTW Wrapper library can be linked with either the ESSL Serial Library or the ESSL SMP library in the following environments:
- 32-bit integer, 32-bit pointer (AIX only)
- 32-bit integer, 64-bit pointer

For additional information about how to use the FFTW Wrapper libraries, see Chapter 5, “Processing Your Program,” on page 185.
Building the FFTW Wrapper libraries on AIX

The C and Fortran wrappers are provided as source code to be compiled using the IBM C/C++ Compiler. The source code, header files and makefiles can be found in the /usr/lpp/essl.rte.common/FFTW3 directory.

To build and install the FFTW Wrapper libraries:
1. Change to a writable directory with approximately 512kb of free space.
2. Do one of the following:
   • Enter:

   cp /usr/lpp/essl.rte.common/FFTW3/src/Makefile
   make install
   —or—
   • Enter:

   make -f /usr/lpp/essl.rte.common/FFTW3/src/Makefile
   install

Building the FFTW Wrapper libraries on Linux

The C and Fortran wrappers are provided as source code to be compiled using the IBM C/C++ Compiler or the gcc compiler. If ESSL was installed in the default location, the source code, header files and makefiles can be found in the /opt/ibmmath/essl/version.release/FFTW3 directory.

Note: For little endian mode only, by default the FFTW Wrapper libraries are installed in /usr/local/lib64. If you wish to install them to /usr/local/lib instead, you can change the value of LIBSUBDIR in Makefile or Makefile.gcc.

To build and install the FFTW Wrapper libraries using IBM XL C:
1. Change to a writable directory with approximately 512kb of free space.
2. Do one of the following:
   • Enter:

   cp /opt/ibmmath/essl/version.release/FFTW3/src/Makefile.gcc
   make install
   —or—
   • Enter:

   make -f /opt/ibmmath/essl/version.release/FFTW3/src/Makefile_gcc
   install

To build and install the FFTW Wrapper libraries using gcc:
1. Change to a writable directory with approximately 512kb of free space.
2. Do one of the following:
   • Enter:

   cp /opt/ibmmath/essl/version.release/FFTW3/src/Makefile.gcc
   make -f ./Makefile_gcc
   install
   —or—
Enter:

```
make -f /opt/ibmmath/essl/version.release/FFTW3/src/Makefile.gcc install
```
Appendix D. Using ESSL with netlib-java and Python

The ESSL libraries provide the BLAS and CBLAS subroutines which can be used by some other programming languages such as:

- netlib-java
- Python (Numpy and Scipy)

This appendix shows how to use the ESSL libraries with these programming languages in RHEL7 (little endian mode), assuming you installed the ESSL packages with a symbolic link in the /usr directory. If you installed the ESSL package without creating the symbolic link in the /usr directory, you must set the correct path for the ESSL runtime libraries, as shown in the example below:

```
/opt/ibmmath/essl/5.5/lib64/libessl.so
```

Setting the BLAS runtime library for netlib-java and Python

Do one of the following to set up the BLAS runtime library for netlib-java and Python:

- To run with the 32-bit integer, 64-bit pointer environment ESSL Serial library as root set, the symbolic link as follows:
  ```
  ln -s /usr/lib64/libessl.so /usr/lib64/libblas.so.3
  ```
- To run with the 32-bit integer, 64-bit pointer environment ESSL SMP library as root, set the symbolic link as follows:
  ```
  ln -s /usr/lib64/libesslsm.so /usr/lib64/libblas.so.3
  ```
- To run with the 32-bit integer, 64-bit pointer environment ESSL SMP CUDA library, as root set the symbolic link as follows:
  ```
  ln -s /usr/lib64/libesslsmcuda.so /usr/lib64/libblas.so.3
  ```

Set the options as suggested in the “Using the ESSL SMP CUDA Library” on page 43 to get the best performance.

Apache Spark and netlib-java Example

netlib-java is a JAVA API used by Apache Spark for linear algebra subroutines such as BLAS, LAPACK, and ARPACK. Use the instructions listed at the following URL to enable the Native Optimized BLAS/CBLAS library for netlib-java:


You can then run the following sample Java application using ESSL:
import com.github.fommil.netlib.BLAS;
public class ddot_example {
    public static void main(String[] args) {
        System.out.println(BLAS.getInstance().getClass().getName());
        double[] x = {1.0, 2.0, -3.0, 4.0, 5.0};
        double[] y = {9.0, 8.0, 7.0, -6.0, 5.0};

        int n;
        int nx = x.length;
        int ny = y.length;

        if (nx != ny) {
            System.err.println("Size of array X is not the same as size of array Y.\nAborting...");
            System.exit(-1);
        }

        n = nx;
        System.out.println("Answer = " + BLAS.getInstance().ddot(n, x, 1, y, 1));
    }
}

Figure 16. ddot_example.java

Compile and run the example code as follows:
java -cp <PATH_TO_netlib-java Jar files> ddot_example.java
java -cp <PATH_TO_netlib-java Jar files>: ddot_example

Figure 17 shows the output:

$ java -cp /mnt/workspace/essl_netlib_example.jar:.. ddot_example
INFO: successfully loaded /tmp/jniloader4688291511028652171netlib-native_system-linux-ppc64le.so
com.github.fommil.netlib.NativeSystemBLAS
Answer = 5.0

Figure 17. Output from ddot_example.java

Scala and netlib-java Example

MLlib uses the linear algebra package Breeze (see http://www.scalanlp.org/),
which depends on netlib-java (see http://github.com/fommil/netlib-java for
optimized numerical processing. Once you have enabled the system native BLAS
library such as ESSL for netlib-java, you can run your application using ESSL, as
shown in the example below:
import com.github.fommil.netlib.BLAS.{getInstance => blas}
println(blas.getClass().getName())

var x = Array(1.0, 2.0, -3.0, 4.0, 5.0)
var y = Array(9.0, 8.0, 7.0, -6.0, 5.0)

var nx = x.length
var ny = y.length
if (nx != ny) {
    println("Size of array X is not the same as size of array Y.\naborting...")
    System.exit(1)
}
var n = nx

val res = blas.ddot(n, x, 1, y, 1)
println("Answer = "+ res)
System.exit(0)

Figure 18. ddot_example.scala

Figure 19 shows the output:

16/10/13 03:25:46 INFO JniLoader: successfully loaded
/tmp/jniloader5596681969452523481netlib-native_system-linux-ppc64le.so
com.github.fommil.netlib.NativeSystemBLAS
x: Array[Double] = Array(1.0, 2.0, -3.0, 4.0, 5.0)
y: Array[Double] = Array(9.0, 8.0, 7.0, -6.0, 5.0)
nx: Int = 5
ny: Int = 5
n: Int = 5
res: Double = 5.0
Answer = 5.0

Figure 19. Output from ddot_example.scala

Python numpy Example

numpy is an open-source add-on module to Python that provides common mathematical and numerical subroutines in pre-compiled, fast functions. After setting the symbolic link for the BLAS library, you can also run the python application using ESSL, as shown in the example below.
```python
#!/usr/bin/python
import numpy as np
x=np.array([1.0, 2.0, -3.0, 4.0, 5.0])
y=np.array([9.0, 8.0, 7.0, -6.0, 5.0])
print np.dot(x, y)
```

Figure 20. ddot_example.py

Figure 21 shows the output:

```
16/10/13 03:25:46 INFO JniLoader: successfully loaded /tmp/jniloader55966819694523481netlib-native_system-linux-ppc64le.so com.github.fommil.netlib.NativeSystemBLAS
x: Array[Double] = Array(1.0, 2.0, -3.0, 4.0, 5.0)
y: Array[Double] = Array(9.0, 8.0, 7.0, -6.0, 5.0)
x: Int = 5
y: Int = 5
n: Int = 5
res: Double = 5.0
Answer = 5.0
```

Figure 21. Output from ddot_example.py
Accessibility Features for ESSL

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility Features

The following list includes the major accessibility features in IBM ESSL. These features support:

- Keyboard-only operation.
- Interfaces that are commonly used by screen readers.
- Keys that are tactiley discernible and do not activate just by touching them.
- Industry-standard devices for ports and connectors.
- The attachment of alternative input and output devices.

IBM Knowledge Center and its related publications, are accessibility-enabled. The accessibility features of IBM Knowledge Center are described in the Accessibility topic at the following URL:

http://www.ibm.com/support/knowledgecenter/

IBM and Accessibility

See the IBM Human Ability and Accessibility Center for more information about the commitment that IBM has to accessibility:

http://www.ibm.com/able/
Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user’s responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

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

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.
IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

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
Intellectual Property Law
2455 South Road, P386
Poughkeepsie, New York 12601-5400
U.S.A.

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

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.
Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at [Copyright and trademark information](http://www.ibm.com/legal/copytrade.shtml).

Acrobat, Adobe, and the Adobe logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft is a trademark of Microsoft Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Software Update Protocol

IBM has provided modifications to this software. The resulting software is provided to you on an "AS IS" basis and WITHOUT A WARRANTY OF ANY KIND, WHETHER EXPRESS OR IMPLIED, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Programming Interfaces

This ESSL Guide and Reference manual is intended to help the customer do application programming. This manual documents General-use Programming Interface and Associated Guidance Information provided by ESSL.

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

References

Text books and articles covering the mathematical aspects of ESSL are listed here, 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.

To find ESSL publications available on the Internet, see "Where to Find Related Publications" on page xviii.

Each citation 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.


For more information, see:

http://www.netlib.org/lapack/index.html


   http://www.netlib.org/benchmark/performance.ps


   http://www.netlib.org/linpack/index.html


48. 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


You can download this document from:
http://www.netlib.org/lapack/lawns/lawn71.ps


You can view this document from:
http://www.netlib.org/benchmark/top500.html


58. EISPACK software library; National Energy Software Center, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439
59. Elmroth, E.; Gustavson, F. "Applying Recursion to Serial and Parallel QR Factorization Leads to Better Performance." To be Published. IBM J. Res. Develop. 44, No. 5.


You can download this document from:
http://www.netlib.org/lapack/lawns/lawn155.ps


Index

Special characters
<complex> or <complex.h> header file  
AIX  170  
Linux  
little endian mode  171

Numerics
3838 Array Processor  
3838 Array Processor  
general signal processing routines  1026

A  
abbreviations  for product names xx  
interpreting math and programming xxiv  
absolute value  maximum 233  
minimum 236  
notation xxiv  
sum of all absolute values 245  
accessibility 1357  
accuracy  
considerations for dense and banded linear algebraic equations 524  
considerations for eigensystem analysis 940  
considerations for Fourier transforms, convolutions, and correlations 1030  
considerations for interpolation 1221  
considerations for linear algebra subprograms 230  
considerations for matrix operations 431  
considerations for numerical quadrature 1243  
considerations for related computations 1034  
considerations for sorting and searching 1201  
error of computation 64  
of results 6, 63  
precisions 63  
what accuracy means 63  
where to find information on 63  
acronyms  
associated with programming values xxiv  
product names xx  
adding  
absolute values 245  
general matrices or their transposes 434  
vector x to vector y and store in vector z 296  
address notation xxiv  
advantages of ESSL 3  
AIX  
<complex> or <complex.h> header file 170  
C++ (C++ programming language)  
setting up complex and logical data 170  
publications 1363  
AIX, supported versions 8  
algebra 517  
Announcing ESSL brochure 1363  
applications in the industry 4  
architecture supported by ESSL for AIX on the workstations 8  

architecture supported by ESSL for Linux on the workstations 8  
arguments  
coding rules 50  
conventions used in the subroutine descriptions xxviii  
diagnosing ESSL input-argument errors 209  
font for ESSL calling xxi  
list of ESSL input-argument errors 212, 219  
passing in C programs 152  
passing in C++ programs 167  
array  
coding in C programs 155  
coding in C++ programs 172  
coding in Fortran programs 134  
conventions for xxiii  
definition of 48  
real and complex elements 134  
setting up data structures inside 75  
storage techniques overview 48  
array data  
storage and performance tradeoffs 65  
arow notation, what it means xxiv  
notation xxiv  
attention error messages, interpreting 211  
autocorrelation of one or more sequences 1172, 1176  
auxiliary working storage  
calculating 53  
dynamic allocation 52  
provided by the user 53  

B  
background books 1363  
band matrix  
definition of 100  
storage layout 101, 103, 106, 107, 110, 111, 114, 115  
band matrix subroutines, names of 517  
band width 100, 105  
banded linear algebraic equation subroutines 517  
SGBF and DGBF 761  
SGBS and DGBS 712, 765  
SGBSV, DGBSV, CGBSV, and ZGBSV 698  
SGBTF, DGBTBF, CGBTRF and ZGBTRF 702  
SGBTRS, DGBTRS, CGBTRS, and ZGBTRS 706  
SGTF and DGTF 775  
SGTNP, DGTNP, CGTNP, and ZGTNP 780  
SGTNPF, DGTNPF, CGTNPF, and ZGTNPF 783  
SGTNPS, DGTNPS, CGTNPS, and ZGTNPS 786  
SGTS and DGTS 778  
SGTSV, DGTTSV, CGTTSV, and ZGTSV 731  
SGTTRF, DGTTRF, CGTTRF, and ZGTTRF 735  
SGTTRS, DGTTRS, CGTTRS, and ZGTTRS 740  
SPBE, DPBE, SPBCHF, and DBPCHF 768  
SPBS, DPBS, SPBSCHS, and DPBSCHS 772  
SPBSV, DPBSV, CPBSV, and ZPBSV 715  
SPBTBF, DPBTBF, CPBTBF, and ZPBTBF 721  
SPBTRS, DPBTRS, CPBTRS, and ZPBTRS 726  
SPTF and DPTF 789  
SPTS and DPTS 791  
SPTSV, DPTSV, CPTSV, and ZPTSV 746
banded linear algebraic equation subroutines (continued)
SPTTRF, DPTTRF, CPTTRF, and ZPTTRF 751
SPTTRS, DPTTRS, CPTTRS, and ZPTTRS 755
STBSV, DTBSV, CTSBV, and ZTBSV 411
base program, processing your
under AIX 185
big endian mode
complex data on AIX 154
complex data on Linux 154
binary search 1213
BLAS (Basic Linear Algebra Subprograms) 225
ESL subprograms 225, 1339
migrating from 206
migrating to ESSL 31
BLAS-general-band storage mode 103
bold letters, usage of xxi
books 1363
C
C (C programming language)
coding programs 151
ESL header file 151, 154
function reference 151
handling errors in your program 158
how to code arrays 155
modifying procedures for using ESSL for AIX 187
modifying procedures for using ESSL for Linux
little endian 193
passing character arguments 152
program calling interface 151
setting up complex and logical data 154
C and C++
publications xviii
C++ (C++ programming language)
coding programs 166
ESL header file 167, 170
function reference 166
handling errors in your program 175
how to code arrays 172
modifying procedures for using ESSL for AIX 188
modifying procedures for using ESSL for Linux
little endian 196
programming interface 166
setting up complex and logical data 170
calculating auxiliary working storage 53
calculating transform lengths 58
CALL statement 133
calling sequence
for C programs 151
for C++ programs 166
for Fortran programs 133
specifying the arguments 50
subroutines versus functions 133, 151, 166
syntax description xxvi
cataloged procedures, ESSL 185
CAXPY 248
CAXPYI 326
CCOPY 251
CDOTC 254
CDOTCI 329
CDOTU 254
CDOTUI 329
calculating auxiliary working storage 53
Ceiling notation and meaning xxiv
CGBMV 379
CGBSV 698
CGBTRF 702
CGBTRS 706
CGEADD 434
CGECMI 509
CGECMO 512
CGECON 556
CGEEV 942
CGEEVX 942
CGEF 544
CGEMM 461
CGEMMS 455
CGEMUL 446
CGEMV 334
CGEQRF 895
CGERC 345
CGERU 345
CGES 547
CGESDD 881
CGESM 551
CGESUB 440
CGESV 530
CGESVD 881
CGETMI 509
CGETMO 512
CGETRF 534
CGETRI 565
CGETRS 539
CGGEV 991
CGGEVX 991
CGGHER 991
CGGHRZ 323
CGTNP 780
CGTNPF 783
CGTNPS 786
character data
conventions xxii, 48
characters, special usage of xxiv
CHBM 386
CHEEVD 978
CHEEVR 959
CHEEVR 1008
CHEMM 470
CHEMV 453
CHER 362
CHER2 370
CHER2K 501
CHERK 494
Choosing the ESSL library 31
Choosing the ESSL subroutine 31
CHPEVD 978
CHPEVX 959
CHPGVX 1008
CHP MV 353
CHPR 362
CHPR2 370
Citations 1363
CLANGE 572
CLANHE 637
CLANHP 637
CLANTP 690
CLANTR 690
CNORM 271
coding your program
arguments in ESSL calling sequences 50
CALL sequence for C programs 151
CALL sequence for C++ programs 166
CALL sequence for Fortran programs 133
coding your program (continued)
calls to ESSL in C programs 151
calls to ESSL in C++ programs 166
calls to ESSL in Fortran programs 133
data types used in your program 48
handling errors with ERRSET, EINFO, ERRSAV, ERRSTR, and return codes 140, 150, 175
restrictions for application programs 48
techniques that affect performance 65
column vector 75
comparison of accuracy for libraries 6
compilers, required by ESSL for AIX on the workstations 9
compilers, required by ESSL for Linux on the workstations 9
compiling your program
C programs 187, 193
C++ programs 188
little endian 196
Fortran programs 185, 192
under AIX 185
complex and real array elements 134
complex conjugate notation xxiv
complex data
AIX
big endian mode 154
conventions xxii, 48
Linux
big endian mode 154
little endian mode 155
setting up for C 154
setting up for C++ 170
complex Hermitian band matrix
definition of 108
storage layout 108
complex Hermitian matrix
definition of 90
storage layout 90
complex Hermitian Toeplitz matrix
definition of 92
complex Hermitian tridiagonal matrix 116
definition of 116
storage layout 116
complex matrix 81
complex vector 75
compressed-diagonal storage mode for sparse matrices 118
compressed-matrix storage mode for sparse matrices 117
compressed-vector, definition and storage mode 80
computational areas, overview 4
computational errors
diagnosing 210
list of messages for 217
overview 68
condition number, reciprocal of
general matrix 561, 565
positive definite complex Hermitian matrix 626
positive definite real symmetric matrix 620, 626
conjugate notation xxiv
conjugate transpose
of matrix operation results for multiply 449, 459, 465
conjugate transpose of a matrix 82
conjugate transpose of a vector 76
continuation, convention for numerical data xxii
conventions xxii
for messages 212
mathematical and programming notations xxiv
subroutine descriptions xxvi
convolution and correlation
autocorrelation of one or more sequences 1172
convolution and correlation (continued)
direct method
one sequence with another sequence 1151
with decimated output 1167
mixed radix Fourier method
autocorrelation of one or more sequences 1176
one sequence with one or more sequences 1157
one sequence with one or more sequences 1145
convolution and correlation subroutines
accuracy considerations 1030
performance and accuracy considerations 1032
performance considerations 1030
SACOR 1172
SACORF 1176
SCON and SCOR 1145
SCOND and SCORD 1151
SCONF and SCORF 1157
SDCON, DDCON, SDCOR, and DDCOR 1167
usage considerations 1027
copy a vector 251
correlation 1145
cosine notation xxiv
cosine transform 1085
courier font usage xxiv
CPBSV 715
CPBTRF 721
CPBTRS 726
CPOCON 611
CPOF 587
CPOSM 600
CPOSV 581
CPOTRF 587
CPOTRI 626
CPOTRS 600
CPPCON 611
CPPSV 575
CPPTRF 587
CPPTRI 626
CPPTRS 600
CPTSV 746
CPTTRF 751
CPTTRIS 755
CROT 280
CROTG 274
CSCAL 290
CSCTR 317
CSRAT 280
CSSCAL 290
CSWAP 293
CSYAX 308
CSYMM 470
CSYR2K 501
CSYRK 494
CTBMV 405
CTBSV 411
CTPMV 391
CTPSV 398
CTPTRI 682
CTRMM 478
CTRMMV 391
CTRSM 486
CTRSV 398
CTRTRI 682
cubic spline interpolating 1232, 1237
customer service, IBM 207
customer support, IBM 207
CVEA 296
dense and banded subroutines in the ESSL

DDOTI, DDCOR, DDCON, DCRFT2, DCFTD, DBSTRS, DBSSV, DBSSV, DBSRCH, DAXPYI, DCY, CVES, CVEM, SPPF, SPOTRI, DPOTRI, CPOTRI, ZPOTRI, SPOICD, DPOICD, SPOSM, DPOSM, CPOSM, ZPOSM, SPPTRS, DPPTRS, SGETRS, DGETRS, CGETRS, and ZGETRS

SGESV, SGES, DGES, CGES, and ZGES

SGEF, DGEE, CGEF, and ZGEF

SGEFC and DGEFCD

SGES, DGES, CGES, and ZGES

SGESM, DGESM, CGESM, and ZGESM

SGESV

SGETRF, DGETRF, CGETRF, and ZGETRF

SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD

SGETRS, DGETRS, CGETRS, and ZGETRS

SPOFC and DPOFC

SPOSM, DPOSM, CPOSM, ZPOSM, SPPTRS, DPPTRS, CPPTRS, ZPPTRS, SPOTRS, DPOTRS, CPOTRS, and ZPOTRS

SPOSV

SPOTRI, DPOTRI, CPOTRI, ZPOTRI, SPOICD, DPOICD, SPPTRI, DPPTPRI, CPPTPRI, ZPPTPRI, SPPICD, and DPPICD

SPPF, DPPE, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPOE, DPOE, CPOE, ZPOE, SPOTRF, DPOTRF, CPOTRF, ZPOTRF

SPPFCD and DPPFCD
Index 1375

DGETRF 534
DGETRI 565
DGETRS 539
DGGEV 991
DGGEVX 991
DGHMQ 1266
DGKFS 804
DGKTRN 1327
DGLGQ 1259
DGLNQ 1250
DGLNQ2 1253
DGRAQ 1262
DGSF 794
DGSS 800
DGTF 775
diagnosis procedures
attention error messages 211
comparative errors 210
ESSL messages, list of 211
in your program 207
in problem diagnosis procedures (symptom index) 209
input-argument errors 209
miscellaneous error messages 211
program exceptions 209
resource error messages 210
diagnostics 211
diagonal-out skyline storage mode 124
dimensions of arrays
storage layout 134
direct method
general skyline sparse matrix 804
general sparse matrix 794
symmetric skyline sparse matrix 821
direct sparse matrix solvers
usage considerations 524
disability 1357
distributions of Linux that support ESSL 8
DIZC 1186
DLANGE 572
DLANSP 637
DLANSY 637
DLANTP 690
DLANTR 690
DNAXPY 258
DNADOT 263
DNORM2 271
DNRANF 1286
DNRM2 268
DNRNG 1279
dot product
notation xxiv
of dense vectors 254
of sparse vectors 329
special (compute N times) 263
DPBCHF 768
DPBCHS 772
DPBF 768
DPBS 772
DPBSV 715
DPBTRF 721
DPBTRS 726
DPINT 1223
DPOCON 611
DPOF 587
DPOFCD 620
DPOICCD 626
DPOLY 1183
DPOS 600
DPOSV 581
DPOTRF 587
DPOTRI 626
DPOTRS 600
DPPOCON 611
DPPF 587
DPFFCD 620
DPPICD 626
DPPS 608
DPSSV 575
DPPTRF 587
DPPTRI 626
DPPTRS 600
DPPTF 789
DPDNTNQ 1247
DPDTS 791
DPDTSV 746
DPDTRF 751
DPDTRS 755
DQINT 1192
DRDFT 1069
DRDFT2 1108
DRDFT3 1130
DRDFTD 1044
DROT 280
DROTTG 274
DROTM 287
DROTMG 284
DSBMV 386
DCAL 290
DSCTR 317
DSDCG 858
DSDCGC 873
DSDMX 425
DSINF 1093
DSKFS 821
DSKTRN 1332
DSLAX 333
D SLR1 362
D SLR2 370
DSMCG 850
DSMCY 866
DSMXX 416
DSMTM 421
DSORT 1204
DSORTS 1209
DSORTX 1206
DSPEVD 978
DSPEVX 959
DSPEVX 353
DSPR 362
DSPR2 370
DSRIS 839
DSRSM 1323
DSRCH 1217
DSWAP 293
DSYEV 978
DSYEVX 959
DSYEVX 1008
efficiency of your program 65
eigensystem analysis subroutines
   performance and accuracy considerations 940
   SGEEV, DGEEV, CGEEV, and ZGEEV 942
   SGEEVX, DGEEVX, CGEEVX, and ZGEEVX 942
   SGEV, DGGEV, CGGEV, ZGGEV, SGGEVX, DGGEVX, 
      CGGEVX, and ZGGEVX 991
   SSEPVD, DSEPVD, CHPEVD, ZHEPVD, SSYEVD, 
      DSYEVD, CHEEVD, ZHEEVD 978
   SSEPVE, DSEPVE, CHPEVX, ZHEPVE, SSYEVX, DSYEVX, 
      CHEEVX, ZHEEVX 959
eigenvalues and eigenvectors 942
   complex Hermitian matrix 959, 978
   real symmetric matrix 959, 978
eigenvectors and eigenvalues 942
EINFO, ESSL error information-handler
   considerations when designing your program 68
   diagnosis procedures using 210
   subroutine description 1296
   using EINFO in C programs 158
   using EINFO in C++ programs 175
   using EINFO in Fortran programs 141
   element of a matrix notation xxiv
   element of a vector notation xxiv
   error conditions, conventions used in the subroutine 
      descriptions xxix
   error messages 211
   error option table default values 71
   error-handling subroutines
      EINFO 1296
      ERRSAV 1299
      ERRSET 1300
      ERRSTR 1302
errors
   attention error messages, interpreting 211
   attention messages, overview 70
   calculating auxiliary storage 51
   computational errors 68, 210
   EINFO subroutine description 1296
   extended error-handling subroutines 30
   handling errors in your C program 158
   handling errors in your C++ program 175
   handling errors in your Fortran program 140 
      how errors affect output 67
   informational error messages, interpreting 211
   input-argument errors 209
   input-argument errors, overview 67
   miscellaneous error messages, interpreting 211
      overview of 67
   program exceptions 67, 209
   resource error messages, interpreting 210
   resource errors, overview 70
   types of errors that you can encounter 67
   using ERRSAV and ERRSTR 73
   values returned for EINFO error codes 1296
   when to use ERRSET 71 
      where to find information on 67
ERRSAV
   in workstation environment 30
   subroutine description 1299
   using with large applications 73
ERRSET
   diagnosis procedures using 210
   ESSL default values for 71
   handling errors in C 158
   handling errors in C++ 175
   in workstation environment 30
   subroutine description 1300
   using EINFO in C programs 158
   using EINFO in C++ programs 175
   using EINFO in Fortran programs 141
   using ERRSET, EINFO, and return codes in C 158
   using ERRSET, EINFO, and return codes in C++ 175
   using ERRSET, EINFO, and return codes in Fortran 141
   when to use 68, 71
   when to use ERRSAV and ERRSTR with ERRSET 73
ERRSTR
   in workstation environment 30
   subroutine description 1302
   using with large applications 73
ESSL (Engineering and Scientific Subroutine Library)
   advantages of 3
   attention error messages, interpreting 211
   attention messages, overview 70
   coding your program 133
   computational areas, overview 4
   computational errors 68
   computational errors, diagnosing 210
   designing your program 31
   diagnosis procedures for ESSL errors 207
   eigensystem analysis subroutines 939
   error option table default values 71
   extended error-handling subroutines 30
   Fourier transform, convolutions and correlations, and
   related-computation subroutines 1025
   functional capability 4
   informational error messages, interpreting 211
   input-argument errors, diagnosing 209
   input-argument errors, overview 67
ESSL (Engineering and Scientific Subroutine Library)
(continued)
interpolation subroutines 1221
introduction to  3
languages supported  8, 9
linear algebra subprograms  225
linear algebraic equations subroutines  517
matrix operation subroutines  429
message conventions  212
messages, list of  211
migrating from one IBM hardware platform to
another  205
migrating programs  201
migrating to future releases or future hardware  205
miscellaneous error messages, interpreting  211
name xx
names with an underscore, interpreting  xix
number of subroutines in each area  4
numerical quadrature subroutines  1243
ordering publications 1363
overview  3
overview of the subroutines  4
packaging characteristics  10
parallel processing subroutines on the workstations  5
processing your program  185
program exceptions  67
program number for  1363
publications overview 1363
random number generation subroutines  1269
reference information conventions  xxvi
related publications 1363
resource error messages, interpreting  210
resource errors, overview  70
setting up your data structures  75
sorting and searching subroutines  1201
usability of subroutines  3
utility subroutines  1293
when coding large applications  73
when to use ERRSET for ESSL errors  71
ESDL libraries
  Blue Gene  48
  shared  48
ESDL messages  211
ESDL SMP CUDA Library
  using  43
ESDL/370, migrating from  206
Euclidean length
  with no scaling of input  271
  with scaling of input  268
Euclidean norm notation  xxiv
examples of matrices  81
examples of vectors  75
examples, conventions used in the subroutine
descriptions  xxix
exponential function notation  xxiv
expressions, special usage of  xxiv
extended error-handling subroutines
  handling errors in C  158
  handling errors in C++  175
  handling errors in your Fortran program  140
  how they work  67, 73
  in ESSL and in Fortran, list of  30
  using them in diagnosing problems  209
extended-error-handling subroutines, using
  in your C program  158
  in your C++ program  175
  in your Fortran program  140
extreme eigenvalues and eigenvectors  942
  general matrix  942

F
factor and solve
  general band matrix with multiple right-hand sides  698
  general tridiagonal matrix  731
factoring
  complex Hermitian matrix  581
  general band matrix  702, 761
  general matrix  530, 534, 544, 561
  general skyline sparse matrix  804
  general sparse matrix  794
  general tridiagonal matrix  735, 775
  indefinite
    complex Hermitian matrix  642, 651
    complex symmetric matrix  642, 651
    real symmetric matrix  642, 651
  positive definite
    complex Hermitian band matrix  715, 721, 726
    complex Hermitian matrix  575, 587
    real symmetric band matrix  715, 721, 726
    real symmetric indefinite matrix  673, 678
    real symmetric matrix  575, 587, 620
    symmetric band matrix  768
    symmetric tridiagonal matrix  789
    real symmetric matrix  581
    symmetric skyline sparse matrix  821
    tridiagonal matrix  751, 780, 783
fast Fourier transform (FFT)  1032
FFT  1032
floor notation and meaning  xxiv
fonts used  xxi
formula for transform lengths, interpreting  59
formulas for auxiliary storage, interpreting  53
Fortran
  languages required by ESSL for AIX on the
    workstations  9
  languages required by ESSL for Linux on the
    workstations  9
  modifying procedures for using ESSL
    Linux (little endian mode)  192
  modifying procedures for using ESSL for AIX  185
Fortran considerations
  coding programs  133
  function reference  225
  handling errors in your program  141
  Fortran function reference  133
  Fortran program calling interface  133
Fourier transform  1032
  one dimension
    complex  1036, 1060
    complex-to-real  1052, 1077
    cosine transform  1085
    real-to-complex  1044, 1069
    sine transform  1093
  three dimensions
    complex  1123
    complex-to-real  1137
    real-to-complex  1130
  two dimensions
    complex  1101
    complex-to-real  1115
    real-to-complex  1108
  fourier transform subroutines
    SCFT and DCFT  1060
Fourier transform subroutines (continued)
SCFT2 and DCFT2 1101
SCFT3 and DCFT3 1123
SCFTD and DCFTD 1036
SCOSF and DCOSF 1085
SCRFT and DCRFT 1077
SCRFT2 and DCRFT2 1115
SCRFT3 and DCRFT3 1137
SCRFTD and DCRFTD 1052
SRCFT and DRCFT 1069
SRCFT2 and DRCFT2 1108
SRCFT3 and DRCFT3 1130
SRCFTD and DRCFTD 1044
SSINF and DSINF 1093
Fourier transform subroutines
accuracy considerations 1030
how they achieve high performance 1032
performance considerations 1030
terminology used for 1027
usage considerations 1027
Frobenius norm notation xxiv
full-matrix storage mode 116
full-vector, definition and storage mode 80
function
calling sequence in C programs 151
calling sequence in C++ programs 166
calling sequence in Fortran programs 133
function reference 225
functional capability of the ESSL subroutines 4
functional description, conventions used in the subroutine
descriptions xxviii
functions, ESSL 225
future migration considerations 205

G
Gather vector elements 320, 323
Gaussian quadrature methods
Gauss-Hermite Quadrature 1266
Gauss-Laguerre Quadrature 1259
Gauss-Legendre Quadrature 1250
Gauss-Rational Quadrature 1262
two-dimensional Gauss-Legendre Quadrature 1253
general matrix 517
general matrix subroutines, names of 517
general tridiagonal matrix
definition of 112
storage layout 112
general-band storage mode 101
generation of random numbers 1269
Givens plane rotation, constructing 274
greek letters notation xxiv
guide information 1
guidelines for handling problems 207

H
half band width 105
handling errors
in your C program 158
in your C++ program 175
in your Fortran program 140
hardware
required on the workstations 8
header file, ESSL, for C 151, 154
header file, ESSL, for C++ 167, 170
Hermitian band matrix
definition of 108
storage layout 108
Hermitian matrix
definition of 90
definition of, complex 92
storage layout 90
how to use this documentation xvii, xviii
Hypertext Markup Language, required products 10

I
i-th zero crossing 1026, 1186
IBM products, migrating from 206
IBM publications 1363
IBM Request for Enhancement (RFE) Community xix
IBSRCH 1213
ICAMAX 233
IDAMAX 233
IDAMIN 236
identify the GPUs ESSL should use 1305
identifying problems 209
IDMAX 239
IDMIN 242
IESSL 1303
indefinite complex Hermitian matrix
definition of 91
storage layout 91
indefinite symmetric matrix 89
definition of 89
storage layout 90
industry areas 4
infinity notation xxiv
informational error messages, interpreting 211
informational messages, for ESSL 211
initiate random number generators subroutine
INITRNG 1271
INITRNG 1271
input arguments, conventions used in the subroutine
descriptions xxviii
input data, conventions for 48
input-argument errors
diagnosing 209
list of messages for 212, 219
overview 67, 70
integral notation and meaning xxiv
integer data
coventions xxii, 48
integral notation xxiv
interchange elements of two vectors 293
interface, ESSL
for C programs 151
for C++ programs 166
for Fortran programs 133
interpolating
cubic spline 1232
local polynomial 1228
polynomial 1223
quadratic 1192
two-dimensional cubic spline 1237
interpolation subroutines
accuracy considerations 1221
performance considerations 1221
SCSIN2 and DCSIN2 1237
SCSINT and DCSINT 1232
SPINT and DPINT 1223
STPINT and DTPINT 1228
library (continued)
migrating from ESSL for Linux on Power Version 5 Release 1 to Version 5 Release 1 203
migrating from ESSL for Linux on Power Version 5 Release 2 to Version 5 Release 3 202
migrating from ESSL Version 4 Release 1 to Version 4 Release 2 204
migrating from ESSL Version 4 Release 2 to Version 4 Release 2.1 204
migrating from ESSL Version 4 Release 2.1 to Version 4 Release 2.2 204
migrating from ESSL Version 4 Release 3 to Version 4 Release 4 203
migrating from ESSL Version 4 Release 4 to Version 5 Release 1 203
migrating from LAPACK 206
migrating to ESSL Version 4 Release 3 204
overview 4
Licensed Program Specification, ESSL 1363
linear algebra 517
linear algebra subprograms 225
accuracy considerations 230
list of matrix-vector linear algebra subprograms 228
list of sparse matrix-vector linear algebra subprograms 229
list of sparse vector-linear algebra subprograms 227
list of vector-scalar linear algebra subprograms 225
performance considerations 230
usage considerations 230
linear algebraic equation subroutines
CGECON 556
CLANGE 572
CLANHE 637
CLANHP 637
CLANTP 690
CLANTR 690
CPOCON 611
CPPCON 611
DGECON 556
DLANGE 572
DLANSP 637
DLANSY 637
DLANTP 690
DLANTR 690
DPOCON 611
DPFCON 611
SLECON 556
SLANGE 572
SLANSY 637
SLANTP 690
SLANTR 690
SPOCON 611
SPFCON 611
ZGGECON 556
ZLANGE 572
ZLANHE 637
ZLANHP 637
ZLANTP 690
ZLANTR 690
ZPOCON 611
ZPPCON 611
linear algebraic equations
accuracy considerations 524
list of banded linear algebraic equation subroutines 520
list of dense linear algebraic equations 517

Index 1379

levels of ESSL, getting 1303
library

ESL SMP CUDA 43
migrating from a non-IBM 206
migrating from another IBM 206
migrating from ESSL for AIX 5.1 and ESSL for Linux on Power Version 5 Release 1.1 to Version 5 Release 2 202
migrating from ESSL for Linux on Power Version 5 Release 2 to Version 5 Release 3.1 201
migrating from ESSL for Linux on Power Version 5 Release 3 to Version 5 Release 3.1 201
migrating from ESSL for Linux on Power Version 5 Release 3.1 to Version 5 Release 3.2 201
migrating from ESSL for Linux on Power Version 5 Release 3.2 to Version 5 Release 4 201
migrating from ESSL for Linux on Power Version 5 Release 4 to Version 5 Release 5 201
interpolation subroutines (continued) usage considerations 1221
introduction to ESSL 3
inverse
general matrix 565
matrix notation xxiv
positive definite complex Hermitian matrix 626
positive definite real symmetric matrix 626
triangular matrix 682
ISAMAX 233
ISAMIN 236
ISMAX 239
ISMIN 242
ISORT 1204
ISORTS 1209
ISORTX 1206
ISSRCH 1217
italic font usage xxi
iterative linear system solver
general sparse matrix 839, 866, 873
sparse negative definite symmetric matrix 850, 858
sparse positive definite symmetric matrix 850, 858
symmetric sparse matrix 839
usage considerations 526
IZAMAX 233

L

l(2) norm
with no scaling of input 271
with scaling of input 268
languages supported by ESSL 8
languages supported by ESSL for AIX 9
LAPACK and LAPACKE
ESL subprograms 1343
LAPACK Blower-tridiagonal-packed storage mode 112
LAPACK tridiagonal matrices storage layout 112
LAPACK upper-tridiagonal-packed storage mode 112
LAPACK, migrating from 206
leading dimension for matrices
how it is used for matrices 83
how it is used in three dimensions 131
least squar solution 927
CGEQRF 895
DGEQRF 895
SGEQRF 895
ZGEQRF 895
letters, fonts of xxi
level of ESSL, getting 1303
library

ESL SMP CUDA 43
migrating from a non-IBM 206
migrating from another IBM 206
migrating from ESSL for AIX 5.1 and ESSL for Linux on Power Version 5 Release 1.1 to Version 5 Release 2 202
migrating from ESSL for Linux on Power Version 5 Release 2 to Version 5 Release 3.1 201
migrating from ESSL for Linux on Power Version 5 Release 3 to Version 5 Release 3.1 201
migrating from ESSL for Linux on Power Version 5 Release 3.1 to Version 5 Release 3.2 201
migrating from ESSL for Linux on Power Version 5 Release 3.2 to Version 5 Release 4 201
migrating from ESSL for Linux on Power Version 5 Release 4 to Version 5 Release 5 201

linear algebraic equations (continued)
list of linear least squares subroutines 523
list of sparse linear algebraic equation subroutines 522
overview 517
performance considerations 524
usage considerations 524
linear least squares solution
preparing for 881, 919
QR decomposition with column pivoting 932
QR factorization 901, 911
singular value decomposition 927
linear least squares subroutines 517
CGEQRF 895
DGELS 901
DGELSD 911
DGEQRF 895
SGELLS and DGELLS 932
SGELS 901
SGELSD 911
SGEQRF 895
SGESVD, DGESVD, CGESVD, ZGESVD, SGESDD, DGESDD, CGESDD, and ZGESDD 881
SGESVF and DGESVF 919
SGESVS and DGESVS 927
ZGELS 901
ZGELSD 911
ZGEEQRF 895
linking
C programs 187, 193
C++ programs 188
little endian 196
Fortran programs 185, 192
linking and loading your program
under AIX 185
Linux
little endian mode
<complex> or <complex.h> header file 171
Fortran program procedures 192
processing your program 191
Linux, supported distributions 8
little endian mode
complex data on Linux 155
logical data
conventions xxii, 48
setting up for C 154
setting up for C++ 170
long precision
accuracy statement 6
meaning of 63
lower band width 100
lower storage mode 85, 88
lower-band-packed storage mode 107
lower-packed storage mode 85
lower-storage-by-rows for symmetric sparse matrices 122
lower-trapezoidal storage mode 98, 99
lower-trapezoidal-packed storage mode 98
lower-triangular storage mode 94, 96
lower-triangular-band-packed storage mode 110, 111
lower-triangular-packed storage mode 94, 95
lower-tridiagonal storage mode 112, 113
lower-tridiagonal-packed storage mode 113
math and programming notations xxiv
math background publications 1363
mathematical expressions, conventions for xxiv
mathematical functions, overview 4
matrix
class matrix 100
complex Hermitian band matrix 108
complex Hermitian matrix 90, 92
complex Hermitian Toeplitz matrix 92
complex Hermitian tridiagonal matrix 116
conventions for xxii
description of 81
font for xxii
full or dense matrix 116
general tridiagonal matrix 112
indefinite complex Hermitian matrix 91
leading dimension for 83
negative definite complex Hermitian matrix 91
negative definite symmetric matrix 89
positive definite Complex hermitian band matrix 108
positive definite complex Hermitian matrix 91
positive definite complex Hermitian tridiagonal matrix 116
positive definite symmetric band matrix 107
positive definite symmetric matrix 89
positive definite symmetric tridiagonal matrix 115
sparse matrix 116
storage of 82
symmetric band matrix 105
symmetric matrix 85
symmetric tridiagonal matrix 114
Toeplitz matrix 91
trapezoidal matrices 96
triangular band matrices 109
triangular matrices 93
matrix operation subroutines
accuracy considerations 431
performance considerations 431
SGEADD, DGEEADD, CGEADD, and ZGEEADD 434
SGEMM, DGEMM, CGEMM, and ZGEMM 461
SGEMMS, DGEMMS, CGEMMS, and ZGEMMS 455
SGEMUL, DGEMUL, CGEMUL, and ZGEMUL 446
SGESUB, DGESUB, CGESUB, and ZGESUB 440
SGETMI, DGETMI, CGETMI, ZGETMI, CGECMI, and ZGETMI 509
SGETMO, DGETMO, CGETMO, CGECMO, and ZGETMO 512
SSYMM, DSYMM, CSSYMM, ZSMM, CHEMM, and ZHEMM 470
SSYR2K, DSYR2K, CSSYR2K, ZSYR2K, CHER2K, and ZHER2K 501
SSYRK, DSYRK, CSSYRK, ZSYRK, CHERK, and ZHERK 494
STRMM, DTRMM, CTRMM, and ZTRMM 478
usage considerations 430
matrix-matrix product
complex Hermitian matrix 470
complex symmetric matrix 470
general matrices, their transposes, or their conjugate transposes 461
real symmetric matrix 470
triangular matrix 478
matrix-vector linear algebra subprograms
SGBMV, DGBMV, CGBMV, and ZGBMV 379
SGEMX, DGEMX, CGEMTX, DGEMTX, SGEMV, DGEMV, CGEMV, and ZGEMV 334
SGER, DGER, CGERU, ZGERU, CGERU, and ZGERU 345
matrix-vector linear algebra subprograms (continued)
SSBMV, DSBMV, CHBMV, and ZHBMV 386
SSPMV, DSPMV, CHPMV, ZHPMV, SYMV, DSYMV,
CHEMV, ZHEMV, SSLMX and DSLMX 353
SSPR, DSPR, CHPR, ZHPR, SYR, DSYR, CHER, ZHER,
SSLR1, and DSSLR2 362
SSPR2, DSPR2, CHPR2, ZHPR2, SYR2, DSYR2, CHER2,
ZHER2, SSLR2, and DSSLR2 370
STBMV, DTMVM, CTBMV, and ZTBMV 405
STPMV, DTPMV, CTPMV, ZTPMV, STRMV, DTRMV,
CTRVM, and ZTRMV 391
matrix-vector product
complex Hermitian, band matrix 386
complex Hermitian matrix 353
general band matrix, its transpose, or its conjugate
transpose 379
general matrix, its transpose, or its conjugate
transpose 334
real symmetric band matrix 386
real symmetric matrix 353
sparse matrix 418
sparse matrix or its transpose 425
triangular band matrix, its transpose, or its conjugate
transpose 405
triangular matrix, its transpose, or its conjugate
transpose 391
max notation and meaning xxiv
maximum
absolute value 233
value 239
messages
ESSL and attention messages, interpreting 211
ESSL informational messages, interpreting 211
ESSL miscellaneous messages, interpreting 211
ESSL resource messages, interpreting 210
list of ESSL messages 212, 219
message conventions 212
migrating
from ESSL Version 4 Release 1 to Version 4 Release 2 204
from ESSL Version 4 Release 2 to Version 4 Release
2.1 204
from ESSL Version 4 Release 2.1 to Version 4 Release
2.2 204
from ESSL Version 4 Release 3 to Version 4 Release 4 203
from ESSL Version 4 Release 4 to Version 5 Release 1 203
from ESSL 370 206
from LAPACK 206
from non-IBM libraries 206
from one IBM hardware platform to another 205
from other IBM subroutine libraries 206
future migration considerations 205
migrating from ESSL for AIX 5.1 and ESSL for Linux on
Power Version 5 Release 1.1 to Version 5 Release 2 202
migrating from ESSL for Linux on Power Version 5 Release
2 to Version 5 Release 3.1 201
migrating from ESSL for Linux on Power Version 5 Release
3 to Version 5 Release 3 201
migrating from ESSL for Linux on Power Version 5 Release
3.1 to Version 5 Release 3.2 201
migrating from ESSL for Linux on Power Version 5 Release
3.2 to Version 5 Release 4 201
migrating from ESSL for Linux on Power Version 5 Release
4 to Version 5 Release 5 201
migrating from ESSL for Linux on Power Version 5 Release
1 to Version 5 Release 1 203
migrating from ESSL for Linux on Power Version 5 Release
2 to Version 5 Release 3 202
migrating (continued)
programs to ESSL 201
to ESSL Version 4 Release 3 204
min notation and meaning xxiv
minimum
absolute value 236
value 242
miscellaneous error messages, interpreting 211
mod notation and meaning xxiv
modification level of ESSL, getting 1303
modified Givens transformation 284
applying a 287
modifying
C programs, for using ESSL for AIX 187
C programs, for using ESSL for Linux little endian 193
C++ programs, for using ESSL for AIX 188
C++ programs, for using ESSL for Linux little endian 196
Fortran programs, for using ESSL 192
Fortran programs, for using ESSL for AIX 185
modulo notation xxiv
multiplying
compute SAXPY or DAXPY N times 258
general matrices using Strassen’s algorithm 455
general matrices, their transposes, or their conjugate
transposes 446
notation xxiv
sparse vector x by a scalar, add sparse vector y, and store
in vector y 326
vector x by a scalar and store in vector x 290
vector x by a scalar and store in vector y 308
vector x by a scalar, add to vector y, and store in vector
y 248
vector x by a scalar, add to vector y, and store in vector
z 312
vector x by vector y, and store in vector z 304
multithreaded
ESSL subroutines 31

N
name usage restrictions 48
names in ESSL with an underscore (_) prefix, how to
interpret xix
names of
products and acronyms xx
the eigensystem analysis subroutines 939
the Fourier transform, convolution and correlation, and
related-computation subroutines 1025
the interpolation subroutines 1221
the linear algebra subprograms 225
the linear algebraic equations subroutines 517
the matrix operations subroutines 429
the numerical quadrature subroutines 1243
the random number generation subroutines 1269
the sorting and searching subroutines 1201
the utility subroutines 1293
National Language Support 208
negative definite complex Hermitian matrix
definition of 91
negative definite complex Hermitian Toeplitz matrix
definition of 92
negative definite Hermitian matrix
storage layout 91
negative definite symmetric matrix
definition of 89

Index 1381
negative definite symmetric matrix (continued)
storage layout 89
negative definite symmetric Toeplitz matrix
definition of 91
negative stride, for vectors 79
netlib-java 1353
NLS, National Language Support 208
non-IBM library, migrating from 206
norm notation xxiv
normally distributed pseudo-random numbers, generate 1279
normally distributed random numbers, generate 1286
notations and conventions xxi
notes, conventions used in the subroutine descriptions xxviii
number of subroutines in each area 4
numbers 29
accuracy of computations 64
accuracy of computations, for ESSL 6
numerical quadrature
accuracy considerations 1243
performance considerations 1243
programming considerations for SUBF 1244
usage considerations 1243
numerical quadrature performed on a function
using Gauss-Hermite Quadrature 1266
using Gauss-Laguerre Quadrature 1259
using Gauss-Legendre Quadrature 1250
using Gauss-Rational Quadrature 1262
using two-dimensional Gauss-Legendre Quadrature 1253
on a set of points 1247
numerical quadrature subroutines
SGHMQ and DGHMQ 1266
SGLQQ and DGLQQ 1259
SGLNQ and DGLNQ 1250
SGLNQ2 and DGLNQ2 1253
SGRAQ and DGRAQ 1262
SPTNQ and DPTNQ 1247
P
packed band storage mode 101
packed-Hermitian-Toeplitz storage mode 93
packed-symmetric-Toeplitz storage mode 92
parallel processing
introduction to 5
performance
achieving better performance in your program 65
aspects of parallel processing on the workstations 5
coding techniques that affect performance 65
considerations for dense and banded linear algebraic equations 524
considerations for eigensystem analysis 940
considerations for Fourier transforms, convolutions, and correlations 1030
considerations for interpolation 1221
considerations for linear algebra subprograms 230
considerations for matrix operations 431
considerations for numerical quadrature 1243
considerations for related computations 1034
considerations for sorting and searching 1201
how the Fourier transforms achieve high performance 1032
information on ESSL run-time performance 67
tradeoffs for convolution and correlation subroutines 1032
where to find information on 67
pi notation xxiv
PL/I (Programming Language/I)
handling errors in your program 158, 175
plane rotation
applying a 280
constructing a Givens 274
planning your program 31
polynomial
evaluating 1183
interpolating 1223, 1228
positive definite Complex hermitian band matrix
definition of 108
storage layout 108
positive definite complex Hermitian matrix
definition of 91
positive definite complex Hermitian Toeplitz matrix
definition of 92
positive definite complex Hermitian tridiagonal matrix 116
definition of 116
storage layout 116
positive definite Hermitian matrix
storage layout 91
positive definite symmetric band matrix
definition of 107
storage layout 108
positive definite symmetric band matrix subroutines, names of 517
positive definite symmetric matrix
definition of 89
storage layout 89
positive definite symmetric matrix subroutines, names of 517
positive definite symmetric Toeplitz matrix
definition of 91
positive definite symmetric tridiagonal matrix 115
definition of 115
storage layout 115
positive stride, for vectors 78

O
one norm notation xxiv
online documentation
required Hypertext Markup Language products 10
operating systems that support ESSL 8
option table, default values for ESSL errors 71
order numbers of the publications 1363
ordering IBM publications 1363
output
accuracy on different processors 6
how errors affect output 67
output arguments, conventions used in the subroutine descriptions xxviii
overflow, avoiding 268
overview
of eigensystem analysis 939
of ESSL 3
of Fourier transforms, convolutions and correlations, and related computations 1025
of interpolation 1221
of linear algebra subprograms 225
of linear algebraic equations 517
of matrix operations 429
of numerical quadrature 1243
of random number generation 1269
of sorting and searching 1201
of the documentation 1363
overview (continued)
of utility subroutines 1293
program exceptions
  description of ESSL  67
program exceptions, diagnosing  209
program number for ESSL  1363
programming considerations for SUBF in numerical quadrature  1244
programming items, font for xx
programming products
  required by ESSL for Linux on the workstations  9
  required by ESSL for AIX on the workstations  9
programming publications  1363
pseudo-random number generation
  uniformly distributed  1276
pseudo-random number generation subroutines
  SURN and DURN  1276
PTF
  getting the most recent level applied  1303
publications
  list of ESSL  1363
  math background  1363
  related  1363
Python  1353

Q
QR decomposition with column pivoting  932
QR factorization  901, 911
quadratic interpolation  27, 1192

R
random number generation
  long period uniformly distributed  1289
  normally distributed  1279, 1286
  uniformly distributed  1283
  usage considerations  1269
random number generation subroutines
  SNRAND and DNRAND  1286
  SNRNG and DNRNG  1279
  SURAND and DURAND  1283
  SURXOR and DURXOR  1289
random number generators
  initiate  1271
random number generators, initiate  1271
rank-2k update
  complex Hermitian matrix  501
  complex symmetric matrix  501
  real symmetric matrix  501
rank-k update
  complex Hermitian matrix  494
  complex symmetric matrix  494
  real symmetric matrix  494
rank-one update
  complex Hermitian matrix  362
  general matrix  345
  real symmetric matrix  362
rank-two update
  complex Hermitian matrix  370
  real symmetric matrix  370
real and complex array elements  134
real data
  conventions xxii, 48
real general matrix eigensystem analysis subroutine  939
real symmetric matrix eigensystem analysis subroutine  939
reciprocal of the condition number
  general matrix  561, 565

precision, meaning of  63
precision, short and long  6
problems, handling  207
problems, IBM support for  207
processing your program
  requirements for ESSL for AIX on the workstations  8
  requirements for ESSL for Linux on the workstations  8
  steps involved in  185
  using parallel subroutines on the workstations  5
Processing your program
Linux
  little endian mode  191
processor-independent formulas for auxiliary storage, interpreting  53
product 248
  matrix-matrix
    complex Hermitian matrix  470
    complex symmetric matrix  470
    general matrices, their transposes, or their conjugate transposes  461
    real symmetric matrix  470
    triangular matrix  478
  matrix-vector
    complex Hermitian band matrix  386
    complex Hermitian matrix  353
    general band matrix, its transpose, or its conjugate transpose  379
    general matrix, its transpose, or its conjugate transpose  334
    real symmetric band matrix  386
    real symmetric matrix  353
    sparse matrix  418
    sparse matrix or its transpose  425
    triangular band matrix, its transpose, or its conjugate transpose  405
    triangular matrix, its transpose, or its conjugate transpose  391
product names, acronyms for xx
products, programming
  migrating from LAPACK  206
  migrating from other IBM  206
  required by ESSL for AIX on the workstations, programming  9
  required by ESSL for Linux on the workstations, programming  9
profile-in skyline storage mode  126
program
  attention messages, overview  70
  coding  133
  computational errors  68
  design  31
  errors  67
  handling errors in your C program  158
  handling errors in your C++ program  175
  handling errors in your Fortran program  140
  input-argument errors, overview  67
  interface for C programs  151
  interface for C++ programs  166
  interface for Fortran programs  133
  migrated to ESSL  201
  performance, achieving high  65
  processing your program  185
  resource errors, overview  70
  setting up your data structures  75
  types of data in your program  48
  when coding large applications  73
reciprocal of the condition number (continued)  626
positive definite complex Hermitian matrix  626
positive definite real symmetric matrix  620, 626
reference information
  math background texts and reports  1363
  organization of  223
  what is in each subroutine description and the conventions used xxvi
related publications  1363
related-computation subroutines
  accuracy considerations  1034
  CWLEV and ZWLEV  1196
  performance considerations  1034
  SIZC and DIZC  1186
  SPOLY and DPOLY  1183
  SQINT and DQINT  1192
  STREC and DTREC  1189
  SWLEV and DWLEV  1196
release of ESSL, getting  1303
reporting problems to IBM  207
required publications  1363
requirements
  auxiliary working storage  52, 53
  for ESSL for AIX workstation product  8
  for ESSL for Linux workstation product  8
  software products on the workstations  9
  transforms in storage, lengths of  58
  workstation hardware  8
resource error messages, interpreting  210
restrictions, ESSL coding  48
results
  accuracy on different processors  6
  how accuracy is affected by the nature of the computation  64
  in C programs  151
  in C++ programs  166
  in Fortran programs  133
  multiplication of NaN  64
results transposed and conjugate transposed for matrix multiplication  449, 459, 465
results transposed for matrix addition  436
results transposed for matrix subtraction  442
return code
  in C programs  158
  in C++ programs  175
  in Fortran programs  141
  using during diagnosis  210
rotation
  applying a plane  280
  constructing a Givens plane  274
routine names  48
row vector  75
run-time performance
  optimizing in your program  65
run-time problems, diagnosing
  attention error messages, interpreting  211
  computational errors  210
  informational error messages, interpreting  211
  input-argument errors  209
  miscellaneous error messages, interpreting  211
  resource error messages, interpreting  210
running your program
  C programs  187, 193
  C++ programs  188
  little endian  196
  Fortran programs  185, 192

S
SCOR  1172
SCorf  1176
SASUM  245
SAXPY  248
SAXPYI  326
SBSRCH  1213
scalar data
  conventions xxii, 48
  scalar items, font for xx
  scale argument used for Fourier transform subroutines  1031
  scaling, when to use  65
SCASUM  245
scatter vector elements  317
SCFT  1060
SCFT2  1101
SCFT3  1123
SCFTD  1036
SCNRM2  268
SCON  1145
SCOND  1151
SCONF  1157
SCOPY  251
SCOR  1145
SCORD  1151
SCORF  1157
SCOSF  1085
SCOSFT, no documentation provided for  1025
SCRFT  1077
SCRFT2  1115
SCRFT3  1137
SCRFTD  1082
SCSIN2  1237
SCSINT  1232
SDCON  1167
SDCOR  1167
SDOT  254
SDOTI  329
searching
  binary  1213
  sequential  1217
selecting an ESSL library  31
selecting an ESSL subroutine  31
sequences
  conventions for xxiii
  description of  128
  storage layout  128
  sequential search  1217
service, IBM  207
set the number of GPUs  1305
SETGPUS
  identify the GPUs ESSL should use  1305
  set the number of GPUs  1305
setting up
  AIX procedures  185
  setting up your data  48
SGBF  761
SGBMV  379
SGBS  712, 765
SGBSV  698
SGBTRF  702
SGBTRS  706
SGEADD  434
SGECON  556
SGEEV  942
SGEEVX  942
SGEF  544
SGEFCD 561
SGEICD 565
SGELLS 932
SGELS 901
SGELSD 91
SGEMM 461
SGEMMS 455
SGEMTX 334
SGEMUL 446
SGEMV 334
SGEQRF 895
SGER 345
SGES 547
SGESDD 881
SGESM 551
SGESUB 440
SGESV 530
SGESVD 881
SGEVX 991
SGLMQ 1266
SGLQ 1259
SGLNQ 1250
SGLNQ2 1253
SCRAQ 1262
SGTF 775
SGTHR 320
SGTHRZ 323
SGTNP 783
SGTNPS 786
SGTS 778
short precision
  accuracy statement 6
  meaning of 63
SIGN notation and meaning  xxiv
signal processing subroutines 1026
simple formulas for auxiliary storage, interpreting 53
sin notation  xxiv
sine transform 1093
singular value decomposition for a general matrix 881, 919, 927
SIZC 1186
size of array
  required for a vector 77
skyline solvers
  usage considerations 525
skyline storage mode for sparse matrices, diagonal-out 124
skyline storage mode for sparse matrices, profile-in 126
SL MATH (Subroutine Library–Mathematics)
  migrating from 206
SLANGE 572
SLANSP 637
SLANSY 637
SLANTP 690
SLANTR 690
SLSS (Subscription Library Services System) 1363
SMP
  ESSL Library, why use it 31

SMP (continued)
  ESSL multithreaded subroutines 31
  performance 6
SNAPXY 258
SNDOT 263
SNORM2 271
SRAND 1286
SNRM2 268
SNRG 1279
software products
  required by ESSL for Linux on the workstations 9
  required by ESSL for AIX on the workstations 9
solve
  indefinite
    complex Hermitian matrix 660
    complex symmetric matrix 660
    real symmetric matrix 660
solving
  general band matrix 712, 765
  general band matrix with multiple right-hand sides 706
  general matrix 556, 572, 611, 637, 690
  general matrix or its transpose 539, 547
  general skyline sparse matrix 804
  general sparse matrix or its transpose 800
  general tridiagonal matrix 740, 755, 778, 780, 786
iterative linear system solver
  general sparse matrix 839, 866, 873
  sparse negative definite symmetric matrix 850, 858
  sparse positive definite symmetric matrix 850, 858
  symmetric sparse matrix 839
multiple right-hand sides
  general matrix, its transpose, or its conjugate transpose 539, 551
  positive definite complex Hermitian matrix 600
  positive definite real symmetric matrix 600
  triangular matrix 486
  positive definite
    real symmetric matrix 608
    symmetric band matrix 772
    symmetric tridiagonal matrix 791
  symmetric skyline sparse matrix 821
  triangular band matrix 411
  triangular matrix 398
  tridiagonal matrix multiple right-hand side solve 746
sorting
  elements of a sequence 1204
  index 1206
  stable sort 1209
sorting and searching subroutines
  accuracy considerations 1201
  IBSRCH, SBSRCH, and DBSRCH 1213
  ISORT, SISORT, and DSORT 1204
  ISORTS, SISORTS, and DSORTS 1209
  ISORTX, SISORTX, and DSORTX 1206
  ISSRCH, SISRCH, and DSSRCH 1217
  performance considerations 1201
  usage considerations 1201
sparse linear algebraic equation subroutines 517
DGKFS 804
DGSF 794
DGSS 800
DSDCG 858
DSDGCG 873
DSKFS 821
DSMCG 850
DSMGCG 866

Index 1385
sparse linear algebraic equation subroutines (continued)
  DSRIS 839
sparse matrix subroutines
  direct solvers 524
  iterative linear system solvers 526
  performance and accuracy considerations 525, 526, 527
skyline solvers 525
sparse matrix-vector linear algebra subprograms
  DSDMX 425
  DSMMX 418
  DSMTM 421
sparse matrix, definition and storage modes 116
sparse vector-scalar linear algebra subprograms
  SAXPYI, DAXPYI, CAXPYI, and ZAXPYI 326
  SDOTI, DDOTI, CDOTUI, ZDOTUI, CDOTCI, and, ZDOTCI 329
  SGTHR, DGTHR, CGTHR, and ZGTHR 320
  SGTHRZ, DGTHRZ, CGTHRZ, and ZGTHRZ 323
  SSCTR, DSCTR, CSCTR, and ZSCTR 317
sparse vector, definition and storage modes 80
SPBCHF 768
SPBCHS 772
SPBF 768
SPBS 772
SPBSV 715
SPBTRF 721
SPBTRS 726
special usage
  of matrix addition 436
  of matrix multiplication 449, 459, 465
  of matrix subtraction 442
spectral norm notation xxiv
SPINT 1223
SPOCON 611
SPOF 587
SPOFCD 620
SPOICD 626
SPOLY 1183
SPOSM 600
SPOSV 581
SPOTRF 587
SPOTRI 626
SPOTRS 600
SPPCON 611
SPPF 587
SPPFCD 620
SPPICD 626
SPTS 608
SPPSV 575
SPPTRF 587
SPPTRI 626
SPPTRS 600
SPTF 789
SPTNQ 1247
SPTS 791
SPTSV 746
SPTTRF 751
SPTTRS 755
SQINT 1192
square root notation xxiv
SRCFT 1069
SRCFT2 1108
SRCFT3 1130
SRCFTD 1044
SROT 280
SROTG 274
SROTM 287

SROTMG 284
SBMV 386
SCAL 290
SCCTR 317
SSINF 1093
SSLMX 353
SSLR1 362
SSLR2 370
SORT 1204
SORTS 1209
SORTX 1206
SSP (Scientific Subroutine Package)
  migrating from 206
  SSPEVD 978
  SSPEVX 959
  SSPGVX 1008
  SSPMV 353
  SSR 362
  SSR2 370
  SSRCH 1217
  SSWAP 293
  SYEVD 978
  SYEVX 959
  SYMVX 1008
  SYMM 470
  SYMV 353
  SYR 362
  SYR2 370
  SYR2K 501
  SYRK 494
  stable sort 1209
  STBMV 405
  STBSV 411
stepping through storage, for matrices 82
stepping through storage, for vectors 78
storage
  array storage techniques overview 48
  auxiliary working storage requirements 52, 53
  compressed-diagonal storage mode for sparse matrices 118
  compressed-matrix storage mode for sparse matrices 117
  considerations when designing your program 48
  diagonal-out skyline storage mode for sparse matrices 124
  for matrices 82
  for vectors 77
  layout for a complex Hermitian band matrix 108
  layout for a complex Hermitian matrix 90
  layout for a complex Hermitian tridiagonal matrix 116
  layout for a general tridiagonal matrix 112
  layout for a indefinite complex Hermitian matrix 91
  layout for a indefinite symmetric matrix 90
  layout for a positive definite Hermitian matrix 91
  layout for a negative definite symmetric matrix 89
  layout for a positive definite complex Hermitian tridiagonal matrix 116
  layout for a positive definite Hermitian matrix 91
  layout for a positive definite symmetric matrix 89
  layout for a positive definite symmetric tridiagonal matrix 115
  layout for a sequence 128, 129, 130
  layout for a symmetric tridiagonal matrix 114
  layout for a Toeplitz matrix 92, 93
  layout for band matrices 101, 103
  layout for LAPACK tridiagonal matrices 112
  layout for positive definite symmetric band matrices 108
  layout for sparse matrices 116
  layout for sparse vectors 80

1386  ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
storage (continued)
  layout for symmetric band matrices 105
  layout for symmetric matrices 85
  layout for trapezoidal matrices 98
  layout for triangular band matrices 110, 111, 114, 115
  layout for triangular matrices 94
  layout for tridiagonal matrices 113
  list of subroutines using auxiliary storage 58
  list of subroutines using transforms 58
  of arrays in Fortran 134
  positive definite Complex hermitian band matrix 108
  profile-in skyline storage mode for sparse matrices 126
  storage-by-columns for sparse matrices 121
  storage-by-indices for sparse matrices 121
  storage-by-rows for sparse matrices 122
  tradeoffs for input 65
  transform length requirements 58
storage conversion subroutine
  general skyline sparse matrix 1327
  sparse matrix 1323
  symmetric skyline sparse matrix 1332
  storage-by-columns for sparse matrices 121
  storage-by-indices for sparse matrices 121
  storage-by-rows for sparse matrices 122
  STPINT 1228
  STPMV 391
  STPSV 398
  STPTRI 682
Strassen's algorithm, multiplying general matrices 455
STREC 1189
stride
  defining vectors in arrays 78
  how it is used in three dimensions 131
  negative 79
  optimizing for your Fourier transforms 1031
  positive 78
  subroutine for optimizing Fourier transforms 1307
  zero 79
STRIDE 1307
STRMM 478
STRMV 391
STRSM 486
STRSV 398
STTRI 682
structures, data (vectors and matrices) 75
subject code for ESSL documentation 1363
subprogram
  linear algebra 225
  meaning of xix, 225
subprogram, definition xx
subroutine
  calling sequence format for C programs 151
  calling sequence format for C++ programs 166
  calling sequence format for Fortran programs 133
  choose of 31
  conventions used in the description of xxvi
  overview of ESSL 4
subroutine, definition xx
subroutines, ESSL (continued)
  CGBSV 698
  CGBTRF 702
  CGBTRS 706
  CGEADD 434
  CGECMI 509
  CGECMO 512
  CGEEV 942
  CGEEVX 942
  CGEF 544
  CGEMM 461
  CGEEMMS 455
  CGEMUL 446
  CGEMV 334
  CGEQRF 895
  CGERC 345
  CGERU 345
  CGES 547
  CGESM 551
  CGESUB 440
  CGESV 530
  CGESVD 881
  CGETMI 509
  CGETMO 512
  CGETRF 534
  CGETRI 565
  CGETRS 539
  CGGEV 991
  CGGEVX 991
  CGTHR 320
  CGTHRZ 323
  CGTNP 780
  CGTNPF 783
  CGTNPS 786
  CGTSV 731
  CGTTRF 735
  CGTTRS 740
  CHBMV 386
  CHEEVD 978
  CHEEYX 959
  CHEGTX 108
  CHEMM 470
  CHEMV 353
  CHER 362
  CHER2 370
  CHER2K 501
  CHERK 494
  CHESV 642
  CHETRF 651
  CHETRS 660
  CHEPEVD 978
  CHEPEYX 959
  CHEPGVX 1008
  CHEPMV 353
  CHPR 362
  CHPR2 370
  CHPSV 642
  CHPTF 651
  CHPTRS 660
  CLANGE 572
  CLANHE 637
  CLANHP 637
  CLANTP 690
  CLANTR 690
  CNORM2 271
  CPBSV 715
  CPBTRF 721

Index 1387
subroutines, ESSL (continued)

CPBTRS 726
CPOF 587
CPOS M 600
CPOS V 581
CPOTRF 587
CPOTRI 626
CPOTRS 600
CPPSV 575
CPPTRF 587
CPPTRI 626
CPPTRS 600
CPOTRF 587
CPOTRI 626
CPOTRS 600
CPPSV 575
CPPTRF 587
CPPTRI 626
CPPTRS 600
CPTSV 746
CPTTRF 751
CPTTRS 755
CROT 280
CROT G 274
CSCAL 290
CSCTR 317
CSPSV 642
CSPTRF 651
CSPTRS 660
CSROT 280
CSSCAL 290
CSWAP 293
CSYAX 308
CSYMM 470
CSYR2K 501
CSYRK 494
CSYSV 642
CSYTRF 651
CSYTRS 660
CTBMV 405
CTBSV 411
CTPMV 391
CTPSV 398
CTPTRI 682
CTRM M 478
CTRM V 391
CTRSM 486
CTR SV 398
CTRTRI 682
CVEA 296
CVEM 304
CVES 300
CWLEV 1196
CYAX 308
CZAXPY 312
DASUM 245
DAXPY 248
DAXPYI 326
DBSRCH 1213
DBSV 667
DBSTRF 673
DBSTRS 678
DCFT 1060
DCFT2 1101
DCFT3 1123
DCFTD 1036
DCOPY 251
DCOSF 1085
DCRFT 1077
DCRFT2 1115
DCRFT3 1137
DCRFTD 1052
DCSIN2 1237
DCSINT 1232
DDCON 1167
DDCOR 1167
DDOT 254
DDOT I 329
DGBF 761
DGBMV 379
DGBS 712, 765
DGBSV 698
DGBTRF 702
DGBTRS 706
DGEADD 434
DGEEV 942
DGEEVX 942
DGES 544
DGEC FD 561
DGEC D 565
DGELLS 932
DGELS 901
DGELSD 911
DGEMM 461
DGEMMS 455
DGEMTX 334
DGEMUL 446
DGEMV 334
DGEMX 334
DGESQRF 895
DGERS 345
DGESM 551
DGESUB 440
DGESV 530
DGESVD 881
DGESVF 919
DGESVS 927
DGETMI 509
DGETO M 512
DGETF 534
DGETRI 565
DGETERS 539
DGGE V 991
DGGE VX 991
DGHRM Q 1266
DGKFS 804
DGKTRL 1327
DGKLQ 1299
DGKNQ 1250
DGKLN 1250
DGKLN2 1253
DGKRAQ 1262
DGFS 794
DGSS 800
DGTF 775
DGTHR 320
DGTHR Z 323
DGTNP 780
DGTNP F 783
DGTNP S 786
DGTS 778
DGTV 731
DGTRF 735
DGTRS 740
DIZC 1186
DLANGE 572
DLANSP 637
DLANSP 637
DLANTP 690
DLANTR 690

1388   ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference
### Subroutines, ESSL (continued)

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNAAXPY</td>
<td>258</td>
</tr>
<tr>
<td>DNDOT</td>
<td>263</td>
</tr>
<tr>
<td>DNORM2</td>
<td>271</td>
</tr>
<tr>
<td>DNRAND</td>
<td>1286</td>
</tr>
<tr>
<td>DNRM2</td>
<td>268</td>
</tr>
<tr>
<td>DNRNG</td>
<td>1279</td>
</tr>
<tr>
<td>DPBCHF</td>
<td>768</td>
</tr>
<tr>
<td>DPBCHS</td>
<td>772</td>
</tr>
<tr>
<td>DPBF</td>
<td>768</td>
</tr>
<tr>
<td>DPBS</td>
<td>772</td>
</tr>
<tr>
<td>DPBSV</td>
<td>715</td>
</tr>
<tr>
<td>DPBTRF</td>
<td>721</td>
</tr>
<tr>
<td>DPBTRS</td>
<td>726</td>
</tr>
<tr>
<td>DPINT</td>
<td>1223</td>
</tr>
<tr>
<td>DPOF</td>
<td>587</td>
</tr>
<tr>
<td>DPOFCD</td>
<td>620</td>
</tr>
<tr>
<td>DPOICD</td>
<td>626</td>
</tr>
<tr>
<td>DPOLY</td>
<td>1183</td>
</tr>
<tr>
<td>DPOSM</td>
<td>600</td>
</tr>
<tr>
<td>DPOSV</td>
<td>581</td>
</tr>
<tr>
<td>DPOTRF</td>
<td>587</td>
</tr>
<tr>
<td>DPOTRI</td>
<td>626</td>
</tr>
<tr>
<td>DPOTRS</td>
<td>600</td>
</tr>
<tr>
<td>DPPF</td>
<td>587</td>
</tr>
<tr>
<td>DPPFCD</td>
<td>620</td>
</tr>
<tr>
<td>DPPICD</td>
<td>626</td>
</tr>
<tr>
<td>DPPS</td>
<td>608</td>
</tr>
<tr>
<td>DPPSV</td>
<td>575</td>
</tr>
<tr>
<td>DPPTRF</td>
<td>587</td>
</tr>
<tr>
<td>DPPTRI</td>
<td>626</td>
</tr>
<tr>
<td>DPPTRS</td>
<td>600</td>
</tr>
<tr>
<td>DPTF</td>
<td>789</td>
</tr>
<tr>
<td>DPTNQ</td>
<td>1247</td>
</tr>
<tr>
<td>DPTS</td>
<td>791</td>
</tr>
<tr>
<td>DPTSV</td>
<td>746</td>
</tr>
<tr>
<td>DPTRTF</td>
<td>751</td>
</tr>
<tr>
<td>DPTRTRS</td>
<td>755</td>
</tr>
<tr>
<td>DQINT</td>
<td>1192</td>
</tr>
<tr>
<td>DRCFT</td>
<td>1069</td>
</tr>
<tr>
<td>DRCFT2</td>
<td>1108</td>
</tr>
<tr>
<td>DRCFT3</td>
<td>1130</td>
</tr>
<tr>
<td>DRCFTD</td>
<td>1044</td>
</tr>
<tr>
<td>DROT</td>
<td>280</td>
</tr>
<tr>
<td>DROTG</td>
<td>274</td>
</tr>
<tr>
<td>DROTM</td>
<td>287</td>
</tr>
<tr>
<td>DROTMG</td>
<td>284</td>
</tr>
<tr>
<td>DSBMV</td>
<td>386</td>
</tr>
<tr>
<td>DSCAL</td>
<td>290</td>
</tr>
<tr>
<td>DSCTR</td>
<td>317</td>
</tr>
<tr>
<td>DSDDCG</td>
<td>858</td>
</tr>
<tr>
<td>DSDDCG</td>
<td>873</td>
</tr>
<tr>
<td>DSDMX</td>
<td>425</td>
</tr>
<tr>
<td>DSINF</td>
<td>1093</td>
</tr>
<tr>
<td>DSKFS</td>
<td>821</td>
</tr>
<tr>
<td>DSKTRN</td>
<td>1332</td>
</tr>
<tr>
<td>DSLMX</td>
<td>353</td>
</tr>
<tr>
<td>DSLRI</td>
<td>362</td>
</tr>
<tr>
<td>DSLR2</td>
<td>370</td>
</tr>
<tr>
<td>DSMCC</td>
<td>850</td>
</tr>
<tr>
<td>DSMCCG</td>
<td>866</td>
</tr>
<tr>
<td>DSMMX</td>
<td>418</td>
</tr>
<tr>
<td>DSMTX</td>
<td>421</td>
</tr>
<tr>
<td>DSORT</td>
<td>1204</td>
</tr>
<tr>
<td>DSORTS</td>
<td>1209</td>
</tr>
<tr>
<td>DSORTX</td>
<td>1206</td>
</tr>
<tr>
<td>DSPEVD</td>
<td>978</td>
</tr>
<tr>
<td>DSPEVX</td>
<td>959</td>
</tr>
<tr>
<td>DspmV</td>
<td>353</td>
</tr>
<tr>
<td>DSPr</td>
<td>362</td>
</tr>
<tr>
<td>DSPr2</td>
<td>370</td>
</tr>
<tr>
<td>DSFSV</td>
<td>642</td>
</tr>
<tr>
<td>DSPTRF</td>
<td>651</td>
</tr>
<tr>
<td>DSPTRS</td>
<td>660</td>
</tr>
<tr>
<td>DSRIS</td>
<td>839</td>
</tr>
<tr>
<td>DSRSM</td>
<td>1323</td>
</tr>
<tr>
<td>DSSRCH</td>
<td>1217</td>
</tr>
<tr>
<td>DSWAP</td>
<td>293</td>
</tr>
<tr>
<td>DSYEVD</td>
<td>978</td>
</tr>
<tr>
<td>DSYEVX</td>
<td>959</td>
</tr>
<tr>
<td>DSYGVX</td>
<td>1008</td>
</tr>
<tr>
<td>DSYMv</td>
<td>470</td>
</tr>
<tr>
<td>DSYMv</td>
<td>353</td>
</tr>
<tr>
<td>DSYR</td>
<td>362</td>
</tr>
<tr>
<td>DSYR2</td>
<td>370</td>
</tr>
<tr>
<td>DSYRK</td>
<td>501</td>
</tr>
<tr>
<td>DSYRK</td>
<td>494</td>
</tr>
<tr>
<td>DSYSV</td>
<td>642</td>
</tr>
<tr>
<td>DSYTRF</td>
<td>651</td>
</tr>
<tr>
<td>DSYTRS</td>
<td>660</td>
</tr>
<tr>
<td>DTBMV</td>
<td>405</td>
</tr>
<tr>
<td>DTBSV</td>
<td>411</td>
</tr>
<tr>
<td>DTPINT</td>
<td>1228</td>
</tr>
<tr>
<td>DTPMV</td>
<td>391</td>
</tr>
<tr>
<td>DTFSV</td>
<td>398</td>
</tr>
<tr>
<td>DTFTPI</td>
<td>682</td>
</tr>
<tr>
<td>DTRMM</td>
<td>478</td>
</tr>
<tr>
<td>DTRMV</td>
<td>391</td>
</tr>
<tr>
<td>DTRSM</td>
<td>486</td>
</tr>
<tr>
<td>DTRSV</td>
<td>398</td>
</tr>
<tr>
<td>DTRTRI</td>
<td>682</td>
</tr>
<tr>
<td>DURAND</td>
<td>1283</td>
</tr>
<tr>
<td>DURNG</td>
<td>1276</td>
</tr>
<tr>
<td>DURXOR</td>
<td>1289</td>
</tr>
<tr>
<td>DVEA</td>
<td>296</td>
</tr>
<tr>
<td>DVEM</td>
<td>304</td>
</tr>
<tr>
<td>DVES</td>
<td>300</td>
</tr>
<tr>
<td>DWLEV</td>
<td>1196</td>
</tr>
<tr>
<td>DYAX</td>
<td>308</td>
</tr>
<tr>
<td>DZASUM</td>
<td>245</td>
</tr>
<tr>
<td>DZAXPY</td>
<td>312</td>
</tr>
<tr>
<td>DZNRM2</td>
<td>268</td>
</tr>
<tr>
<td>EINFO</td>
<td>1296</td>
</tr>
<tr>
<td>ERSAV</td>
<td>1299</td>
</tr>
<tr>
<td>ERRSET</td>
<td>1300</td>
</tr>
<tr>
<td>ERRSTR</td>
<td>1302</td>
</tr>
<tr>
<td>IBSRCH</td>
<td>1213</td>
</tr>
<tr>
<td>ICAMAX</td>
<td>233</td>
</tr>
<tr>
<td>IDMAX</td>
<td>233</td>
</tr>
<tr>
<td>IDAMIN</td>
<td>236</td>
</tr>
<tr>
<td>IDMAx</td>
<td>239</td>
</tr>
<tr>
<td>IDMIN</td>
<td>242</td>
</tr>
<tr>
<td>IESSL</td>
<td>1303</td>
</tr>
<tr>
<td>INITRNG</td>
<td>1271</td>
</tr>
<tr>
<td>ISAMAX</td>
<td>233</td>
</tr>
<tr>
<td>ISAMIN</td>
<td>236</td>
</tr>
<tr>
<td>ISMAX</td>
<td>239</td>
</tr>
<tr>
<td>ISMIN</td>
<td>242</td>
</tr>
<tr>
<td>ISORT</td>
<td>1204</td>
</tr>
<tr>
<td>ISORTS</td>
<td>1209</td>
</tr>
<tr>
<td>ISOORTX</td>
<td>1206</td>
</tr>
<tr>
<td>ISSRCH</td>
<td>1217</td>
</tr>
<tr>
<td>Subroutine</td>
<td>ESSL for AIX, 5.3, and ESSL for Linux on POWER, 5.5: Guide and Reference</td>
</tr>
<tr>
<td>-------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IZAMAX</td>
<td>233</td>
</tr>
<tr>
<td>SACOR</td>
<td>1172</td>
</tr>
<tr>
<td>SACORF</td>
<td>1176</td>
</tr>
<tr>
<td>SASUM</td>
<td>245</td>
</tr>
<tr>
<td>SAXPY</td>
<td>248</td>
</tr>
<tr>
<td>SAXPYI</td>
<td>326</td>
</tr>
<tr>
<td>SBSRCH</td>
<td>1213</td>
</tr>
<tr>
<td>SCASUM</td>
<td>245</td>
</tr>
<tr>
<td>SCFT</td>
<td>1060</td>
</tr>
<tr>
<td>SCFT2</td>
<td>1101</td>
</tr>
<tr>
<td>SCFT3</td>
<td>1123</td>
</tr>
<tr>
<td>SCFTD</td>
<td>1036</td>
</tr>
<tr>
<td>SCNRM2</td>
<td>268</td>
</tr>
<tr>
<td>SCON</td>
<td>1145</td>
</tr>
<tr>
<td>SCOND</td>
<td>1151</td>
</tr>
<tr>
<td>SCONF</td>
<td>1157</td>
</tr>
<tr>
<td>SCOPY</td>
<td>251</td>
</tr>
<tr>
<td>SCOR</td>
<td>1145</td>
</tr>
<tr>
<td>SCORD</td>
<td>1151</td>
</tr>
<tr>
<td>SCORF</td>
<td>1157</td>
</tr>
<tr>
<td>SCOSF</td>
<td>1085</td>
</tr>
<tr>
<td>SCRT</td>
<td>1077</td>
</tr>
<tr>
<td>SCRT2</td>
<td>1115</td>
</tr>
<tr>
<td>SCRT3</td>
<td>1137</td>
</tr>
<tr>
<td>SCRTFD</td>
<td>1052</td>
</tr>
<tr>
<td>LCSIN2</td>
<td>1237</td>
</tr>
<tr>
<td>LCSINT</td>
<td>1232</td>
</tr>
<tr>
<td>SDCON</td>
<td>1167</td>
</tr>
<tr>
<td>SDCOR</td>
<td>1167</td>
</tr>
<tr>
<td>SDOT</td>
<td>254</td>
</tr>
<tr>
<td>SDOTI</td>
<td>329</td>
</tr>
<tr>
<td>SETGPUS</td>
<td>1305</td>
</tr>
<tr>
<td>SGBF</td>
<td>761</td>
</tr>
<tr>
<td>SGBMV</td>
<td>379</td>
</tr>
<tr>
<td>SGBS</td>
<td>712, 765</td>
</tr>
<tr>
<td>SGBSV</td>
<td>698</td>
</tr>
<tr>
<td>SGBTRF</td>
<td>702</td>
</tr>
<tr>
<td>SGBTRS</td>
<td>706</td>
</tr>
<tr>
<td>SGEADD</td>
<td>434</td>
</tr>
<tr>
<td>SGEEV</td>
<td>942</td>
</tr>
<tr>
<td>SGEEVX</td>
<td>942</td>
</tr>
<tr>
<td>SGEDF</td>
<td>544</td>
</tr>
<tr>
<td>SGECFD</td>
<td>561</td>
</tr>
<tr>
<td>SGECDF</td>
<td>565</td>
</tr>
<tr>
<td>GEGS</td>
<td>932</td>
</tr>
<tr>
<td>GELS</td>
<td>901</td>
</tr>
<tr>
<td>GELSDF</td>
<td>911</td>
</tr>
<tr>
<td>SGEINT</td>
<td>461</td>
</tr>
<tr>
<td>SGMEM</td>
<td>455</td>
</tr>
<tr>
<td>SGMEMS</td>
<td>334</td>
</tr>
<tr>
<td>SGMUL</td>
<td>446</td>
</tr>
<tr>
<td>SGMV</td>
<td>334</td>
</tr>
<tr>
<td>SGMX</td>
<td>334</td>
</tr>
<tr>
<td>SGEQRF</td>
<td>895</td>
</tr>
<tr>
<td>SGERS</td>
<td>345</td>
</tr>
<tr>
<td>SGES</td>
<td>547</td>
</tr>
<tr>
<td>SGESM</td>
<td>551</td>
</tr>
<tr>
<td>SGESUB</td>
<td>440</td>
</tr>
<tr>
<td>SGESV</td>
<td>530</td>
</tr>
<tr>
<td>SGEVSVD</td>
<td>881</td>
</tr>
<tr>
<td>SGEVSF</td>
<td>919</td>
</tr>
<tr>
<td>SGEVSVS</td>
<td>927</td>
</tr>
<tr>
<td>GETMI</td>
<td>509</td>
</tr>
<tr>
<td>GETMO</td>
<td>512</td>
</tr>
<tr>
<td>SGETRF</td>
<td>534</td>
</tr>
<tr>
<td>SGETRI</td>
<td>565</td>
</tr>
<tr>
<td>Subroutine</td>
<td>Page</td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
</tr>
<tr>
<td>SROT</td>
<td>280</td>
</tr>
<tr>
<td>SROTG</td>
<td>274</td>
</tr>
<tr>
<td>SROTM</td>
<td>287</td>
</tr>
<tr>
<td>SROTMG</td>
<td>284</td>
</tr>
<tr>
<td>SSBMV</td>
<td>386</td>
</tr>
<tr>
<td>SSCAL</td>
<td>290</td>
</tr>
<tr>
<td>SSCTR</td>
<td>317</td>
</tr>
<tr>
<td>SSINF</td>
<td>1093</td>
</tr>
<tr>
<td>SSLMX</td>
<td>353</td>
</tr>
<tr>
<td>SSLR1</td>
<td>362</td>
</tr>
<tr>
<td>SSLR2</td>
<td>370</td>
</tr>
<tr>
<td>SSORT</td>
<td>1204</td>
</tr>
<tr>
<td>SSORTX</td>
<td>1206</td>
</tr>
<tr>
<td>SSORTS</td>
<td>1209</td>
</tr>
<tr>
<td>SSPEVD</td>
<td>978</td>
</tr>
<tr>
<td>SSPEVX</td>
<td>959</td>
</tr>
<tr>
<td>SSPGVX</td>
<td>1008</td>
</tr>
<tr>
<td>SSPMV</td>
<td>353</td>
</tr>
<tr>
<td>SPR</td>
<td>362</td>
</tr>
<tr>
<td>SPPR2</td>
<td>370</td>
</tr>
<tr>
<td>SPPSV</td>
<td>642</td>
</tr>
<tr>
<td>SPTTRF</td>
<td>651</td>
</tr>
<tr>
<td>SSPTRS</td>
<td>660</td>
</tr>
<tr>
<td>SSSRCH</td>
<td>1217</td>
</tr>
<tr>
<td>SWAP</td>
<td>293</td>
</tr>
<tr>
<td>SYEVD</td>
<td>978</td>
</tr>
<tr>
<td>SYEYVX</td>
<td>959</td>
</tr>
<tr>
<td>SYGVX</td>
<td>1008</td>
</tr>
<tr>
<td>SYMM</td>
<td>470</td>
</tr>
<tr>
<td>SYMV</td>
<td>353</td>
</tr>
<tr>
<td>SYR</td>
<td>362</td>
</tr>
<tr>
<td>SYR2</td>
<td>370</td>
</tr>
<tr>
<td>SYR2K</td>
<td>501</td>
</tr>
<tr>
<td>SYRK</td>
<td>494</td>
</tr>
<tr>
<td>SYSV</td>
<td>642</td>
</tr>
<tr>
<td>SYTTRF</td>
<td>651</td>
</tr>
<tr>
<td>SYTRS</td>
<td>660</td>
</tr>
<tr>
<td>STBMV</td>
<td>405</td>
</tr>
<tr>
<td>STBSV</td>
<td>411</td>
</tr>
<tr>
<td>STPINT</td>
<td>1228</td>
</tr>
<tr>
<td>STPMV</td>
<td>391</td>
</tr>
<tr>
<td>STPSV</td>
<td>398</td>
</tr>
<tr>
<td>STPTRI</td>
<td>682</td>
</tr>
<tr>
<td>STREC</td>
<td>1189</td>
</tr>
<tr>
<td>STRIDE</td>
<td>1307</td>
</tr>
<tr>
<td>STRMM</td>
<td>478</td>
</tr>
<tr>
<td>STRMV</td>
<td>391</td>
</tr>
<tr>
<td>STRSM</td>
<td>486</td>
</tr>
<tr>
<td>STRSV</td>
<td>398</td>
</tr>
<tr>
<td>STRTRI</td>
<td>682</td>
</tr>
<tr>
<td>SURAND</td>
<td>1283</td>
</tr>
<tr>
<td>SURNG</td>
<td>1276</td>
</tr>
<tr>
<td>SURXOR</td>
<td>1289</td>
</tr>
<tr>
<td>SVEA</td>
<td>296</td>
</tr>
<tr>
<td>SVEM</td>
<td>304</td>
</tr>
<tr>
<td>SVES</td>
<td>300</td>
</tr>
<tr>
<td>SWLEV</td>
<td>1196</td>
</tr>
<tr>
<td>SYAX</td>
<td>308</td>
</tr>
<tr>
<td>SZAXPY</td>
<td>312</td>
</tr>
<tr>
<td>ZAXPY</td>
<td>248</td>
</tr>
<tr>
<td>ZAXPYI</td>
<td>326</td>
</tr>
<tr>
<td>ZCOPY</td>
<td>251</td>
</tr>
<tr>
<td>ZDOTC</td>
<td>254</td>
</tr>
<tr>
<td>ZDOTCI</td>
<td>329</td>
</tr>
<tr>
<td>ZDOTU</td>
<td>254</td>
</tr>
<tr>
<td>ZDOTUI</td>
<td>329</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZDROT</td>
<td>280</td>
</tr>
<tr>
<td>ZDSCAL</td>
<td>290</td>
</tr>
<tr>
<td>ZDYAX</td>
<td>308</td>
</tr>
<tr>
<td>ZGBMV</td>
<td>379</td>
</tr>
<tr>
<td>ZGBSV</td>
<td>698</td>
</tr>
<tr>
<td>ZGBTRF</td>
<td>702</td>
</tr>
<tr>
<td>ZGBTRS</td>
<td>706</td>
</tr>
<tr>
<td>ZGEADD</td>
<td>434</td>
</tr>
<tr>
<td>ZGECMI</td>
<td>509</td>
</tr>
<tr>
<td>ZGECMO</td>
<td>512</td>
</tr>
<tr>
<td>ZGEEV</td>
<td>942</td>
</tr>
<tr>
<td>ZGEEVX</td>
<td>942</td>
</tr>
<tr>
<td>ZGEF</td>
<td>544</td>
</tr>
<tr>
<td>ZGELS</td>
<td>901</td>
</tr>
<tr>
<td>ZGELSD</td>
<td>911</td>
</tr>
<tr>
<td>ZGEMM</td>
<td>461</td>
</tr>
<tr>
<td>ZGEMMS</td>
<td>455</td>
</tr>
<tr>
<td>ZGEMUL</td>
<td>446</td>
</tr>
<tr>
<td>ZGEMV</td>
<td>334</td>
</tr>
<tr>
<td>ZGEOQRF</td>
<td>895</td>
</tr>
<tr>
<td>ZGERC</td>
<td>345</td>
</tr>
<tr>
<td>ZGERU</td>
<td>345</td>
</tr>
<tr>
<td>ZGES</td>
<td>547</td>
</tr>
<tr>
<td>ZGESM</td>
<td>551</td>
</tr>
<tr>
<td>ZGESUB</td>
<td>440</td>
</tr>
<tr>
<td>ZGESV</td>
<td>530</td>
</tr>
<tr>
<td>ZGESVD</td>
<td>881</td>
</tr>
<tr>
<td>ZGETMI</td>
<td>509</td>
</tr>
<tr>
<td>ZGETMO</td>
<td>512</td>
</tr>
<tr>
<td>ZGETRF</td>
<td>534</td>
</tr>
<tr>
<td>ZGETRS</td>
<td>539</td>
</tr>
<tr>
<td>ZGGEV</td>
<td>991</td>
</tr>
<tr>
<td>ZGGEVX</td>
<td>991</td>
</tr>
<tr>
<td>ZGTHR</td>
<td>320</td>
</tr>
<tr>
<td>ZGTHRZ</td>
<td>323</td>
</tr>
<tr>
<td>ZGTNP</td>
<td>780</td>
</tr>
<tr>
<td>ZGTNPF</td>
<td>783</td>
</tr>
<tr>
<td>ZGTNPS</td>
<td>786</td>
</tr>
<tr>
<td>ZGTSV</td>
<td>731</td>
</tr>
<tr>
<td>ZGTTRF</td>
<td>735</td>
</tr>
<tr>
<td>ZGTTRS</td>
<td>740</td>
</tr>
<tr>
<td>ZHBMV</td>
<td>386</td>
</tr>
<tr>
<td>ZHEEVD</td>
<td>978</td>
</tr>
<tr>
<td>ZHEEYX</td>
<td>959</td>
</tr>
<tr>
<td>ZHEGVX</td>
<td>1008</td>
</tr>
<tr>
<td>ZHEMM</td>
<td>470</td>
</tr>
<tr>
<td>ZHEMV</td>
<td>353</td>
</tr>
<tr>
<td>ZHER</td>
<td>362</td>
</tr>
<tr>
<td>ZHER2</td>
<td>370</td>
</tr>
<tr>
<td>ZHER2K</td>
<td>501</td>
</tr>
<tr>
<td>ZHERK</td>
<td>494</td>
</tr>
<tr>
<td>ZHESV</td>
<td>642</td>
</tr>
<tr>
<td>ZHETRF</td>
<td>651</td>
</tr>
<tr>
<td>ZHETRS</td>
<td>660</td>
</tr>
<tr>
<td>ZHPEVD</td>
<td>978</td>
</tr>
<tr>
<td>ZHPEVX</td>
<td>959</td>
</tr>
<tr>
<td>ZHPGVX</td>
<td>1008</td>
</tr>
<tr>
<td>ZHPMV</td>
<td>353</td>
</tr>
<tr>
<td>ZHPRY</td>
<td>362</td>
</tr>
<tr>
<td>ZHPR2</td>
<td>370</td>
</tr>
<tr>
<td>ZHPSV</td>
<td>642</td>
</tr>
<tr>
<td>ZHPTRF</td>
<td>651</td>
</tr>
<tr>
<td>ZHPTRS</td>
<td>660</td>
</tr>
<tr>
<td>ZLANGE</td>
<td>572</td>
</tr>
<tr>
<td>ZLANHE</td>
<td>637</td>
</tr>
<tr>
<td>Subroutine</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>------</td>
</tr>
<tr>
<td>ZLANHP</td>
<td>637</td>
</tr>
<tr>
<td>ZLANTR</td>
<td>690</td>
</tr>
<tr>
<td>ZNORM2</td>
<td>271</td>
</tr>
<tr>
<td>ZPBV</td>
<td>715</td>
</tr>
<tr>
<td>ZPBLT</td>
<td>721</td>
</tr>
<tr>
<td>ZPBLT</td>
<td>726</td>
</tr>
<tr>
<td>ZPOF</td>
<td>587</td>
</tr>
<tr>
<td>ZPOST</td>
<td>600</td>
</tr>
<tr>
<td>ZDTRF</td>
<td>587</td>
</tr>
<tr>
<td>ZDTRRS</td>
<td>626</td>
</tr>
<tr>
<td>ZDTRRS</td>
<td>600</td>
</tr>
<tr>
<td>ZDTRF</td>
<td>626</td>
</tr>
<tr>
<td>ZDTRF</td>
<td>600</td>
</tr>
<tr>
<td>ZDTPS</td>
<td>746</td>
</tr>
<tr>
<td>ZDTPRF</td>
<td>751</td>
</tr>
<tr>
<td>ZDTPRS</td>
<td>755</td>
</tr>
<tr>
<td>ZDROT</td>
<td>280</td>
</tr>
<tr>
<td>ZDROTG</td>
<td>274</td>
</tr>
<tr>
<td>ZSCAL</td>
<td>290</td>
</tr>
<tr>
<td>ZSCTR</td>
<td>317</td>
</tr>
<tr>
<td>ZSPSV</td>
<td>642</td>
</tr>
<tr>
<td>ZSPTRF</td>
<td>651</td>
</tr>
<tr>
<td>ZSPTRF</td>
<td>660</td>
</tr>
<tr>
<td>ZSWAP</td>
<td>293</td>
</tr>
<tr>
<td>ZSYMM</td>
<td>470</td>
</tr>
<tr>
<td>ZSYRK</td>
<td>494</td>
</tr>
<tr>
<td>ZSYSV</td>
<td>642</td>
</tr>
<tr>
<td>ZSYT</td>
<td>651</td>
</tr>
<tr>
<td>ZSYTRS</td>
<td>660</td>
</tr>
<tr>
<td>ZTBMV</td>
<td>405</td>
</tr>
<tr>
<td>ZTPS</td>
<td>391</td>
</tr>
<tr>
<td>ZTPSV</td>
<td>398</td>
</tr>
<tr>
<td>ZTPTRI</td>
<td>682</td>
</tr>
<tr>
<td>ZTRMM</td>
<td>478</td>
</tr>
<tr>
<td>ZTRMV</td>
<td>391</td>
</tr>
<tr>
<td>ZTRSM</td>
<td>486</td>
</tr>
<tr>
<td>ZTRSV</td>
<td>398</td>
</tr>
<tr>
<td>ZTRTRS</td>
<td>682</td>
</tr>
<tr>
<td>ZVEM</td>
<td>296</td>
</tr>
<tr>
<td>ZVEM</td>
<td>304</td>
</tr>
<tr>
<td>ZVES</td>
<td>300</td>
</tr>
<tr>
<td>ZWLEV</td>
<td>1196</td>
</tr>
<tr>
<td>ZYAX</td>
<td>308</td>
</tr>
<tr>
<td>ZZXAXPY</td>
<td>312</td>
</tr>
</tbody>
</table>

**Strings**

subroutine, ESSL (continued)

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZLANT</td>
<td>690</td>
</tr>
<tr>
<td>ZLIN</td>
<td>690</td>
</tr>
<tr>
<td>ZNORM2</td>
<td>271</td>
</tr>
<tr>
<td>ZPBV</td>
<td>715</td>
</tr>
<tr>
<td>ZPBLT</td>
<td>721</td>
</tr>
<tr>
<td>ZPBLT</td>
<td>726</td>
</tr>
<tr>
<td>ZPOF</td>
<td>587</td>
</tr>
<tr>
<td>ZPOST</td>
<td>600</td>
</tr>
<tr>
<td>ZDTRF</td>
<td>587</td>
</tr>
<tr>
<td>ZDTRRS</td>
<td>626</td>
</tr>
<tr>
<td>ZDTRRS</td>
<td>600</td>
</tr>
<tr>
<td>ZDTRF</td>
<td>626</td>
</tr>
<tr>
<td>ZDTRF</td>
<td>600</td>
</tr>
<tr>
<td>ZDTPS</td>
<td>746</td>
</tr>
<tr>
<td>ZDTPRF</td>
<td>751</td>
</tr>
<tr>
<td>ZDTPRS</td>
<td>755</td>
</tr>
<tr>
<td>ZDROT</td>
<td>280</td>
</tr>
<tr>
<td>ZDROTG</td>
<td>274</td>
</tr>
<tr>
<td>ZSCAL</td>
<td>290</td>
</tr>
<tr>
<td>ZSCTR</td>
<td>317</td>
</tr>
<tr>
<td>ZSPSV</td>
<td>642</td>
</tr>
<tr>
<td>ZSPTRF</td>
<td>651</td>
</tr>
<tr>
<td>ZSPTRF</td>
<td>660</td>
</tr>
<tr>
<td>ZSWAP</td>
<td>293</td>
</tr>
<tr>
<td>ZSYMM</td>
<td>470</td>
</tr>
<tr>
<td>ZSYRK</td>
<td>494</td>
</tr>
<tr>
<td>ZSYSV</td>
<td>642</td>
</tr>
<tr>
<td>ZSYT</td>
<td>651</td>
</tr>
<tr>
<td>ZSYTRS</td>
<td>660</td>
</tr>
<tr>
<td>ZTBMV</td>
<td>405</td>
</tr>
<tr>
<td>ZTPS</td>
<td>391</td>
</tr>
<tr>
<td>ZTPSV</td>
<td>398</td>
</tr>
<tr>
<td>ZTPTRI</td>
<td>682</td>
</tr>
<tr>
<td>ZTRMM</td>
<td>478</td>
</tr>
<tr>
<td>ZTRMV</td>
<td>391</td>
</tr>
<tr>
<td>ZTRSM</td>
<td>486</td>
</tr>
<tr>
<td>ZTRSV</td>
<td>398</td>
</tr>
<tr>
<td>ZTRTRS</td>
<td>682</td>
</tr>
<tr>
<td>ZVEM</td>
<td>296</td>
</tr>
<tr>
<td>ZVEM</td>
<td>304</td>
</tr>
<tr>
<td>ZVES</td>
<td>300</td>
</tr>
<tr>
<td>ZWLEV</td>
<td>1196</td>
</tr>
<tr>
<td>ZYAX</td>
<td>308</td>
</tr>
<tr>
<td>ZZXAXPY</td>
<td>312</td>
</tr>
</tbody>
</table>

subroutine, Parallel ESSL

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGECON</td>
<td>556</td>
</tr>
<tr>
<td>CPOCON</td>
<td>611</td>
</tr>
<tr>
<td>CPPCON</td>
<td>611</td>
</tr>
<tr>
<td>DGECON</td>
<td>556</td>
</tr>
<tr>
<td>DPOCON</td>
<td>611</td>
</tr>
<tr>
<td>DPPCON</td>
<td>611</td>
</tr>
<tr>
<td>SGECON</td>
<td>556</td>
</tr>
<tr>
<td>SPOCON</td>
<td>611</td>
</tr>
<tr>
<td>SPPCON</td>
<td>611</td>
</tr>
<tr>
<td>ZGEC0</td>
<td>556</td>
</tr>
<tr>
<td>ZPOCON</td>
<td>611</td>
</tr>
<tr>
<td>ZPPCON</td>
<td>611</td>
</tr>
</tbody>
</table>

subscript notation, what it means xxiv

<table>
<thead>
<tr>
<th>Notation</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>subscript notation, what it means xxiv</td>
<td></td>
</tr>
<tr>
<td>subtracting</td>
<td></td>
</tr>
<tr>
<td>general matrices or their transposes</td>
<td>440</td>
</tr>
<tr>
<td>subtracting (continued)</td>
<td></td>
</tr>
<tr>
<td>vector y from vector x and store in vector z</td>
<td>300</td>
</tr>
<tr>
<td>sum, calculating</td>
<td></td>
</tr>
<tr>
<td>absolute values</td>
<td>245</td>
</tr>
<tr>
<td>summ xxiv</td>
<td></td>
</tr>
<tr>
<td>superscript notation, what it means xxiv</td>
<td></td>
</tr>
<tr>
<td>support, IBM</td>
<td>207</td>
</tr>
<tr>
<td>SURAND 1283</td>
<td></td>
</tr>
<tr>
<td>SURNG 1276</td>
<td></td>
</tr>
<tr>
<td>SURXOR 1289</td>
<td></td>
</tr>
<tr>
<td>SVEA 296</td>
<td></td>
</tr>
<tr>
<td>SVEM 304</td>
<td></td>
</tr>
<tr>
<td>SVES 300</td>
<td></td>
</tr>
<tr>
<td>swap elements of two vectors</td>
<td>293</td>
</tr>
<tr>
<td>SWLEV 1196</td>
<td></td>
</tr>
<tr>
<td>SYAX 308</td>
<td></td>
</tr>
<tr>
<td>symbols, special usage of xxiv</td>
<td></td>
</tr>
<tr>
<td>symmetric band matrix</td>
<td></td>
</tr>
<tr>
<td>definition of</td>
<td>105</td>
</tr>
<tr>
<td>storage layout</td>
<td>105</td>
</tr>
<tr>
<td>symmetric indefinite matrix</td>
<td></td>
</tr>
<tr>
<td>symmetric indefinite matrix</td>
<td></td>
</tr>
<tr>
<td>real symmetric indefinite matrix</td>
<td>667</td>
</tr>
<tr>
<td>symmetric matrix</td>
<td></td>
</tr>
<tr>
<td>definition of</td>
<td>85</td>
</tr>
<tr>
<td>storage layout</td>
<td>85</td>
</tr>
<tr>
<td>symmetric tridiagonal matrix</td>
<td>114</td>
</tr>
<tr>
<td>definition of</td>
<td>114</td>
</tr>
<tr>
<td>storage layout</td>
<td>114</td>
</tr>
<tr>
<td>symmetric-tridiagonal storage mode</td>
<td>114</td>
</tr>
<tr>
<td>symptoms, identifying problem</td>
<td>209</td>
</tr>
<tr>
<td>syntax rules for call statements and data</td>
<td>50</td>
</tr>
<tr>
<td>syntax, conventions used in the subroutine descriptions xxvi</td>
<td></td>
</tr>
<tr>
<td>SZAXPY 312</td>
<td></td>
</tr>
</tbody>
</table>

**Table**

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>table, default values for ESSL error option</td>
<td>71</td>
</tr>
<tr>
<td>termination, program</td>
<td></td>
</tr>
<tr>
<td>attention messages</td>
<td>70</td>
</tr>
<tr>
<td>computational errors</td>
<td>68</td>
</tr>
<tr>
<td>input-argument errors</td>
<td>67</td>
</tr>
<tr>
<td>resource errors</td>
<td>70</td>
</tr>
<tr>
<td>terminology used for Fourier transforms, convolutions, and correlations</td>
<td>1027</td>
</tr>
<tr>
<td>terminology, names of products</td>
<td>xx</td>
</tr>
<tr>
<td>textbooks cited</td>
<td>1363</td>
</tr>
<tr>
<td>thread-safe</td>
<td></td>
</tr>
<tr>
<td>ESSL Library, why use it</td>
<td>31</td>
</tr>
<tr>
<td>three-dimensional data structures, how stride is used for</td>
<td>131</td>
</tr>
<tr>
<td>time-varying recursive filter</td>
<td>27, 1189</td>
</tr>
<tr>
<td>times notation, multiply xxiv</td>
<td></td>
</tr>
<tr>
<td>timings, achieving high performance in your program</td>
<td>65</td>
</tr>
<tr>
<td>Toeplitz matrix</td>
<td>91</td>
</tr>
<tr>
<td>definition of</td>
<td>91, 92</td>
</tr>
<tr>
<td>storage layout</td>
<td>92, 93</td>
</tr>
<tr>
<td>traceback map, using during diagnosis</td>
<td>210</td>
</tr>
<tr>
<td>transform lengths, calculating</td>
<td>58</td>
</tr>
<tr>
<td>transformation</td>
<td></td>
</tr>
<tr>
<td>modified Givens</td>
<td>284</td>
</tr>
<tr>
<td>transpose</td>
<td></td>
</tr>
<tr>
<td>conjugate, of a matrix</td>
<td>82</td>
</tr>
<tr>
<td>conjugate, of a vector</td>
<td>76</td>
</tr>
<tr>
<td>notation xxiv</td>
<td></td>
</tr>
<tr>
<td>of a matrix</td>
<td>82</td>
</tr>
<tr>
<td>of a matrix inverse notation</td>
<td>xxiv</td>
</tr>
<tr>
<td>of a vector</td>
<td>76, 77</td>
</tr>
</tbody>
</table>
usage considerations (continued)

sparse matrix subroutines (skyline solvers) 525
usage, special
conventions used in the subroutine description xxviii
for matrix addition 436
for matrix multiplication 449, 459, 465
for matrix subtraction 442
user applications 4
Using ESSL with netlib-java and Python 1353
using this documentation xvii, xviii
utility subroutines
DGKTRN 1327
DSKTRN 1332
DSRSM 1323
EINFO 1296
ERRSAV 1299
ERRSET 1300
ERRSTR 1302
IESSL 1303
SETGPU 1305
STRIDE 1307
usage considerations 1293

V

vector
compressed vector 80
conventions for xxii
description of 75
font for xxi
full vector 80
number of array elements needed for 77
sparse vector 80
storage of 77
stride for 78

vector-scalar linear algebra subprograms
ISAMAX, ICAMAX, IDAMAX, and IZAMAX 233
ISAMIN and IDAMIN 236
ISMAX and IDMAX 239
ISMIN and IDMIN 242
SASUM, DASUM, SCASUM, and ZDASUM 245
SAXPY, DAXPY, CAXPY, and ZAXPY 248
SCOPY, DCOPY, CCOPY, and ZCOPY 251
SDOT, DDOT, CDOTU, ZDOTU, CDOTC, and ZDOTC 254
SNAXPY and DNAXPY 258
SNDOT and DNDOT 263
SNORM2, DSNORM2, CNORM2, and ZNORM2 271
SNRM2, DSNRM2, SCNRM2, and DZNRM2 268
SDOT, DROT, CROT, ZROT, CSROT, and ZDROT 280
SROTG, DROTG, CROTG, and ZROTG 274
SROTM and DROTM 287
SROTMG and DROTMG 284
SSCAL, DSCAL, CSCAL, ZSCAL, CSCAL, and ZDSCAL 290
SSWAP, DSWAP, CSWAP, and ZSWAP 293
SVEA, DVEA, CVEA, and ZVEA 296
SVM, DVM, CVEM, and ZVEM 304
SVEE, DVEE, CVEE, and ZVEE 300
SYAX, DYAX, CYAX, ZYAX, CSYAX, and ZDYAX 308
SZAXPY, DZAXPY, CZAXPY, and ZZAXPY 312
version of ESSL, getting 1303
versions of subroutines 4

U

underflow
avoiding underflow 268
why mask it 65
uniformly distributed pseudo-random numbers,
generate 1276
uniformly distributed random numbers, generate 1283, 1289
upper band width 100
upper storage mode 85, 89
upper-band-packed storage mode 106
upper-packed storage mode 85, 87
upper-storage-by-rows for symmetric sparse matrices 122
upper-trapezoidal storage mode 98
upper-trapezoidal-packed storage mode 94, 95
upper-triangular storage mode 110, 114, 115
upper-triangular-packed storage mode 94
upper-tridiagonal storage mode 112, 113
upper-tridiagonal-packed storage mode 113
usability of subroutines 3
usability of the ESSL subroutines 4
usage considerations
direct sparse matrix solvers 524
for Fourier transforms, convolutions, and correlations 1027
for interpolation 1221
for linear algebra subprograms 230
for linear algebraic equations 524
for matrix operations 430
for numerical quadrature 1243
for random number generation 1269
for sorting and searching 1201
for utility subroutines 1293
sparse matrix subroutines (iterative linear system solvers) 526
Wiener-Levinson filter coefficients 1196
Wiener-Levinson filter coefficients subroutine 27
working auxiliary storage, list of subroutines using 51
working storage for band matrix 101
workstations
   migrating from one IBM hardware platform to
   another 205
required for ESSL for AIX 8
required for ESSL for Linux 8

Z
ZAXPY 248
ZAXPYI 326
ZCOPY 251
ZDOTC 254
ZDOTCI 329
ZDOTU 254
ZDOTUI 329
ZDROT 280
ZDSCAL 290
ZDYAX 308
zero stride, for vectors 79
ZGBMV 379
ZGBSV 698
ZGBTRF 702
ZGBTRS 706
ZGEADD 434
ZGECMI 509
ZGECMO 512
ZGECON 556
ZGE EV 942
ZGE EVX 942
ZGEF 544
ZGELS 901
ZGELSD 911
ZGEMM 461
ZGEMMS 455
ZGEMUL 446
ZGEMV 334
ZGEQRF 895
ZGERC 345
ZGERU 345
ZGES 547
ZGESDD 881
ZGESM 551
ZGESUB 440
ZGESV 530
ZGESVD 881
ZGETI 509
ZGETMO 512
ZGETRF 534
ZGETRI 565
ZGFRS 539
ZGGEV 991
ZGGEVX 991
ZGTHR 320
ZGTHRZ 323
ZGTNP 780
ZTNPF 783
ZTNPS 786
ZHB MV 386
ZHEEVD 978
ZHEEVX 959
ZHELVX 1008
ZHELM 370
ZHLNE 572
ZLANHE 637
ZLANHP 637
ZLANTP 690
ZLANTR 690
ZNORM2 271
ZPBSV 715
ZPBTRF 721
ZPBTRS 726
ZPOF 587
ZPOSTF 587
ZPOTRI 626
ZPOTRS 600
ZPPCON 611
ZPPSV 575
ZPPTRF 587
ZPPTRI 626
ZPPTRS 600
ZPTSV 746
ZPTTRF 751
ZPTTRS 755
ZROT 280
ZROTG 274
ZSCAL 290
ZSCTR 317
ZSWAP 293
ZSYMM 470
ZSYR2K 501
ZSYRK 494
ZTBMV 405
ZTBSV 411
ZTPMV 391
ZTPSV 398
ZTPTRI 682
ZTRMM 478
ZTRMV 391
ZTRSM 486
ZTRSV 398
ZTRTRI 682
ZVEA 296
ZVEM 304
ZVES 300
ZWLEV 1196
ZYAX 308
ZZAXPY 312

ZHEMM 470
ZHEMV 353
ZHER 362
ZHER2 370
ZHER2K 501
ZHERK 494
ZHEPEVD 978
ZHEPEVX 959
ZHPGVX 1008
ZHPMV 353
ZHPR 362
ZHPTR 370
ZLANE 572
ZLANHE 637
ZLANHP 637
ZLANTP 690
ZLANTR 690
ZNORM2 271
ZPBSV 715
ZPBTRF 721
ZPBTRS 726
ZPOF 587
ZPOSTF 587
ZPOTRI 626
ZPOTRS 600
ZPPCON 611
ZPPSV 575
ZPPTRF 587
ZPPTRI 626
ZPPTRS 600
ZPTSV 746
ZPTTRF 751
ZPTTRS 755
ZROT 280
ZROTG 274
ZSCAL 290
ZSCTR 317
ZSWAP 293
ZSYMM 470
ZSYR2K 501
ZSYRK 494
ZTBMV 405
ZTBSV 411
ZTPMV 391
ZTPSV 398
ZTPTRI 682
ZTRMM 478
ZTRMV 391
ZTRSM 486
ZTRSV 398
ZTRTRI 682
ZVEA 296
ZVEM 304
ZVES 300
ZWLEV 1196
ZYAX 308
ZZAXPY 312
Product Number: 5765-H25
5765-L51

Printed in USA