IBM Engineering and Scientific Subroutine Library
for Linux on POWER

Version 6.1

*ESSL Guide and Reference*
Note
Before using this information and the product it supports, read the information in “Notices” on page 1249.

This edition applies to Version 6 Release 1 of the IBM® Engineering and Scientific Subroutine Library (ESSL) for Linux on POWER® licensed program (product number 5765-L61) and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright International Business Machines Corporation 2018.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
Contents

List of Tables.................................................................................................................. xv

About this information........................................................................................................ xxiii
  IBM Request for Enhancement (RFE) Community............................................................ xxiv
  Special Terms.................................................................................................................. xxiv
  Short and Long Precision.............................................................................................. xxiv
  Subroutines and Subprograms...................................................................................... xxiv
  Abbreviated Names...................................................................................................... xxiv
  Conventions and terminology used.............................................................................. xxv
  Fonts.............................................................................................................................. xxv
  Special Notations and Conventions.............................................................................. xxvi
  Special Characters, Symbols, Expressions, and Abbreviations................................. xxix
  How to Interpret the Subroutine Descriptions............................................................. xxxi
    Description.............................................................................................................. xxxi
    Syntax...................................................................................................................... xxxi
    On Entry................................................................................................................... xxxiii
    On Return.................................................................................................................. xxxiii
    Notes......................................................................................................................... xxxiii
    Function.................................................................................................................... xxxiii
    Special Usage.......................................................................................................... xxxiv
    Error Conditions...................................................................................................... xxxiv
    Examples................................................................................................................... xxxiv
  What’s new.................................................................................................................... xxxxiv

Chapter 1. Concepts........................................................................................................ 1
  Overview of ESSL......................................................................................................... 1
    Performance and Functional Capability..................................................................... 1
    Usability..................................................................................................................... 2
    The Variety of Mathematical Functions.................................................................... 2
    Accuracy of the Computations.................................................................................. 4
    High Performance of ESSL..................................................................................... 4
    The Fortran Language Interface to the Subroutines.................................................. 5
  Software and hardware .............................................................................................. 5
    Hardware Products Supported by ESSL................................................................. 5
    Operating Systems Supported by ESSL................................................................. 6
    Software Products Required by ESSL................................................................. 6
    Software Products for Installing and Customizing ESSL....................................... 6
    Man pages............................................................................................................... 7
  List of ESSL subroutines............................................................................................. 7
    Linear algebra subprograms..................................................................................... 7
    Matrix operations subroutines.................................................................................. 13
    Linear algebraic equations subroutines................................................................... 15
    Eigensystem analysis subroutines............................................................................ 25
    Fourier Transforms, Convolutions and Correlations, and Related Computations.... 27
    Sorting and searching subroutines.......................................................................... 29
    Interpolation subroutines......................................................................................... 30
    Numerical quadrature subroutines.......................................................................... 30
    Random number generation subroutines............................................................... 31
    Utilities subroutines............................................................................................... 32

Chapter 2. Planning......................................................................................................... 33
Chapter 3. Setting up data structures

Vector concepts

Vectors

Transpose of a Vector

Conjugate Transpose of a Vector

Vector Storage Representation

How Stride Is Used for Vectors

Sparse Vector

Matrices

Transpose of a Matrix

Conjugate Transpose of a Matrix
Chapter 4. Coding programs

Fortran Programs

Calling ESSL Subroutines and Functions in Fortran...................................................... 125
Setting Up a User-Supplied Subroutine for ESSL in Fortran.......................................... 125
Setting Up Scalar Data in Fortran.................................................................................. 125
Setting Up Arrays in Fortran......................................................................................... 126
Creating Multiple Threads and Calling ESSL from Your Fortran Program.................... 133
Handling Errors in Your Fortran Program...................................................................... 133
Example of Handling Errors in a Multithreaded Application Program............................ 141

C Programs.................................................................................................................. 142
Calling ESSL Subroutines and Functions in C............................................................... 142
Passing Arguments in C............................................................................................... 143
Setting Up a User-Supplied Subroutine for ESSL in C.................................................. 144
Setting Up Scalar Data Types in C................................................................................ 144
Setting Up Complex Data Types in C........................................................................... 145
Using Logical Data in C................................................................................................ 145
Setting Up Arrays in C.................................................................................................. 146
Creating Multiple Threads and Calling ESSL from Your C Program............................ 146
Handling Errors in Your C Program.............................................................................. 147

C++ Programs............................................................................................................. 155
Calling ESSL Subroutines and Functions in C++......................................................... 155
Passing Arguments in C++........................................................................................... 156
Setting Up a User-Supplied Subroutine for ESSL in C++.............................................. 157
Setting Up Scalar Data in C++..................................................................................... 157
Using Complex Data in C++......................................................................................... 158
Using Logical Data in C++.......................................................................................... 159
ISMIN and IDMIN (Position of the First or Last Occurrence of the Vector Element Having Minimum Value) .......................................................... 211
SASUM, DASUM, SCASUM, and DZASUM (Sum of the Magnitudes of the Elements in a Vector) .................................................. 214
SAXPY, DAXPY, CAXPY, and ZAXPY (Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in the Vector Y) .......................................................... 216
SAXPBY and DAXPBY (Scales Vector X and Vector Y by two Scalars, Adds them to one another, and Stores in the Vector Y) .......................................................... 219
SCOPY, DCOPY, CCOPY, and ZCOPY (Copy a Vector) .......................................................................................................................... 223
SDOT, DDOT, CDOTU, ZDOTU, CDOTC, and ZDOTC (Dot Product of Two Vectors) .................................................................................. 225
SNAXPY and DNAXPY (Compute SAXPY or DAXPY N Times) .......................................................................................................................... 229
SNRM2, DNRM2, SCNRM2, and DZNRM2 (Euclidean Length of a Vector with Scaling of Input to Avoid Destructive Underflow and Overflow) .......................................................................................................................... 238
SSPR2, DSPR2, CHPR2, ZHPR2, SSYR2, DSYR2, CHER2, ZHER2, SSLR2, and DSLR2 (Rank-Two Update of a General Band Matrix, Its Transpose, or Its Conjugate Transpose) .......................................................................................................................... 306
SGER2, DGER2, CGER2, ZGER2, SGER, DGER, CGER, and ZGER (Rank-One Update of a General Matrix) .......................................................................................................................... 336
SSBMV, DSBMV, CBMV, and ZHBMV (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Band Matrix) .......................................................................................................................... 343
Chapter 9. Matrix operations subroutines ........................................................... 383

Overview of the Matrix Operation Subroutines..................................................... 383
Use Considerations.................................................................................................. 384
Specifying Normal, Transposed, or Conjugate Transposed Input Matrices........ 384
Transposing or Conjugate Transposing:................................................................ 385
Performance and Accuracy Considerations.......................................................... 386
In General............................................................................................................ 386
For Large Matrices............................................................................................ 386
For Combined Operations.................................................................................... 386

Matrix Operation Subroutines............................................................................... 386
SGEADD, DGEADD, CGEADD, and ZGEADD (Matrix Addition for General Matrices or Their Transposes)................................. 386
SGESUB, DGESUB, CGESUB, and ZGESUB (Matrix Subtraction for General Matrices or Their Transposes)......................... 392
SGEMUL, DGEMUL, CGEMUL, and ZGEMUL (Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes).......................................................... 398
SGEMMS, DGEMMS, CGEMMS, and ZGEMMS (Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes Using Winograd's Variation of Strassen's Algorithm)................................................................. 408
SGEMM, DGEMM, CGEMM, and ZGEMM (Combined Matrix Multiplication and Addition for General Matrices, Their Transposes, or Conjugate Transposes) .................. 413
SSYMM, DSYMM, CSYMM, ZSYMM, CHEMM, and ZHEMM (Matrix-Matrix Product Where One Matrix is Real or Complex Symmetric or Complex Hermitian)................. 422
STRMM, DTRMM, CTRMM, and ZTRMM (Triangular Matrix-Matrix Product)........... 430
STRSM, DTRSM, CTRSM, and ZTRSM (Solution of Triangular Systems of Equations with Multiple Right-Hand Sides)......................... 437
SSYRK, DSYRK, CSYRK, ZSYRK, CHERK, and ZHERK (Rank-K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix).................................................. 445
SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, and ZHER2K (Rank-2K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix)........................................ 451
SGETMI, DGETMI, CGETMI, ZGETMI, CGECMI and ZGECMI (General Matrix Transpose or Conjugate Transpose [In-Place])........................ 459
SGETMO, DGETMO, CGETMO, ZGETMO, CGECMO, and ZGECMO (General Matrix Transpose or Conjugate Transpose [Out-of-Place])........................................ 462

Chapter 10. Linear algebraic equations subroutines .............................................. 467

Overview of the Linear Algebraic Equation Subroutines.................................... 467
Dense Linear Algebraic Equation Subroutines..................................................... 467
Banded Linear Algebraic Equation Subroutines.................................................. 472
Sparse Linear Algebraic Equation Subroutines.................................................... 474
Linear Least Squares Subroutines....................................................................... 475
Dense and Banded Linear Algebraic Equation Considerations............................ 477
Use Considerations............................................................................................ 477
Performance and Accuracy Considerations....................................................... 477
Sparse Matrix Direct Solver Considerations...................................................... 477
Dense Linear Algebraic Equation Subroutines................................................................................. 480
SGESV, DGESV, CGESV, ZGESV (General Matrix Factorization and Multiple Right-Hand Side
Solve)............................................................................................................................................. 480
SGETRF, DGETRF, CGETRF and ZGETRF (General Matrix Factorization). ........................................ 484
SGETRS, DGETRS, CGETRS, and ZGETRS (General Matrix Multiple Right-Hand Side Solve). .................. 488
SGEF, DGEF, CGEF, and ZGEF (General Matrix Factorization)........................................................... 493
SGES, DGES, CGES, and ZGES (General Matrix, Its Transpose, or Its Conjugate Transpose
Solve)............................................................................................................................................. 496
SGESM, DGESM, CGESM, and ZGESM (General Matrix, Its Transpose, or Its Conjugate
Transpose Multiple Right-Hand Side Solve)...................................................................................... 499
SGECON, DGECON, CGECON, and ZGECON (Estimate the Reciprocal of the Condition Number
of a General Matrix)........................................................................................................................ 504
SGEFCD and DGEFCD (General Matrix Factorization, Condition Number Reciprocal, and
Determinant).................................................................................................................................. 508
SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD (General Matrix Inverse, Condition
Number Reciprocal, and Determinant)......................................................................................... 511
SLANGE, DLANGE, CLANGE, and ZLANGE (General Matrix Norm). .................................................. 518
SPPSV, DPPSV, CPPSV, and ZPPSV (Positive Definite Real Symmetric and Complex Hermitian
Matrix Factorization and Multiple Right-Hand Side Solve). ......................................................... 521
SPOSV, DPOSV, CPOSV, and ZPOSV (Positive Definite Real Symmetric or Complex Hermitian
Matrix Factorization and Multiple Right-Hand Side Solve). ......................................................... 527
SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF,
ZPPTRF, SPPF, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix
Factorization).................................................................................................................................. 533
SPOTRS, DPOTRS, CPOTRS, ZPOTRS, SPOSM, DPOSM, CPOSM, ZPOSM, SPPTRS, DPPTRS,
CPPTRRS, and ZPPTRS (Positive Definite Real Symmetric or Complex Hermitian
Matrix Multiple Right-Hand Side Solve). ....................................................................................... 545
SPPS and DPPS (Positive Definite Real Symmetric Matrix Solve)..................................................... 552
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)............................................................................................................................ 555
SPFFCD, DPPFCD, SPOFCD, and DPOFCD (Positive Definite Real Symmetric Matrix
Factorization, Condition Number Reciprocal, and Determinant). ................................................. 563
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)........................................................ 569
SLANSY, DLANSY, CLANSY, ZLANSY, CLANHE, ZLANHE, SLANSP, DLANSP, CLANHP, and
ZLANHP (Real Symmetric or Complex Hermitian Matrix Norm). .................................................. 579
SSYSV, DSYSV, CSYSV, ZSYSV, CHESV, ZHESV, SSPSV, DSSPSV, CSPSV, ZSPSV, CHPSV, and
ZHPSV (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Factorization
and Multiple Right-Hand Side Solve).............................................................................................. 584
SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, ZHETRF, SSPTRF, DSPTRF, CSPTRF, ZSPTRF,
CHPTRF, and ZHPTRF (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix
Factorization)................................................................................................................................... 593
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)....................................................................................................... 601
<table>
<thead>
<tr>
<th>Subroutine Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBSSV (Symmetric Indefinite Matrix Factorization and Multiple Right-Hand Side Solve)</td>
<td>607</td>
</tr>
<tr>
<td>DBSTRF (Symmetric Indefinite Matrix Factorization)</td>
<td>613</td>
</tr>
<tr>
<td>DBSTRS (Symmetric Indefinite Matrix Multiple Right-Hand Side Solve)</td>
<td>617</td>
</tr>
<tr>
<td>STRTRI, DTRTRI, CTRTRI, ZTRTRI, STPTRI, DTPTRI, CTPTRI, and ZTPTRI (Triangular Matrix Inverse)</td>
<td>621</td>
</tr>
<tr>
<td>SLANTR, DLANTR, CLANTR, ZLANTR, SLANTP, DLANTP, CLANTP, and ZLANTP (Trapezoidal or Triangular Matrix Norm)</td>
<td>628</td>
</tr>
<tr>
<td><strong>Banded Linear Algebraic Equation Subroutines</strong></td>
<td>635</td>
</tr>
<tr>
<td>SGBSV, DGBSV, CGBSV, and ZGBSV (General Band Matrix Factorization and Multiple Right-Hand Side Solve)</td>
<td>635</td>
</tr>
<tr>
<td>SGBTBF, DGBTRF, CGBTRF and ZGBTRF (General Band Matrix Factorization)</td>
<td>639</td>
</tr>
<tr>
<td>SGBTRS, DGBTRS, CGBTRS, and ZGBTRS (General Band Matrix Multiple Right-Hand Side Solve)</td>
<td>643</td>
</tr>
<tr>
<td>SGBS and DGBS (General Band Matrix Solve)</td>
<td>649</td>
</tr>
<tr>
<td>SPBSV, DPBSV, CPBSV, and ZPBSV (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization and Multiple Right-Hand Side Solve)</td>
<td>651</td>
</tr>
<tr>
<td>SPBTRF, DPBTRF, CPBTRF, and ZPBTRF (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization)</td>
<td>656</td>
</tr>
<tr>
<td>SPBTRS, DPBTRS, CPBTRS, and ZPBTRS (Positive Definite Real Symmetric or Complex Hermitian Band Multiple Right-Hand Side Solve)</td>
<td>661</td>
</tr>
<tr>
<td>SGTSV, DGTTSV, CGTSV, and ZGTSV (General Tridiagonal Matrix Factorization and Multiple Right-Hand Side Solve)</td>
<td>666</td>
</tr>
<tr>
<td>SGTTTRF, DGTTRF, CGTTRF, and ZGTTRF (General Tridiagonal Matrix Factorization)</td>
<td>670</td>
</tr>
<tr>
<td>SGTRTS, DGTTRTS, CGTTRTS, and ZGTTRTS (General Tridiagonal Matrix Multiple Right-Hand Side Solve)</td>
<td>674</td>
</tr>
<tr>
<td>SPTSV, DPTSV, CPTSV, and ZPTSV (Positive Definite Real Symmetric or Complex Tridiagonal Matrix Factorization and Multiple Right-Hand Side Solve)</td>
<td>679</td>
</tr>
<tr>
<td>SPTTRF, DPTTRF, CPTTRF, and ZPTTRF (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization)</td>
<td>684</td>
</tr>
<tr>
<td>SPTTRTS, DPTTRTS, CPTTRTS, and ZPTTRTS (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Multiple Right-Hand Solve)</td>
<td>687</td>
</tr>
<tr>
<td>SGBF and DGBF (General Band Matrix Factorization)</td>
<td>693</td>
</tr>
<tr>
<td>SGBS and DGBS (General Band Matrix Solve)</td>
<td>696</td>
</tr>
<tr>
<td>SPBF, DPBF, SPBCHF, and DPBCHF (Positive Definite Symmetric Band Matrix Factorization)</td>
<td>698</td>
</tr>
<tr>
<td>SPBS, DPBS, SPBCHS, and DPBCHS (Positive Definite Symmetric Band Matrix Solve)</td>
<td>701</td>
</tr>
<tr>
<td>SGTF and DGT (General Tridiagonal Matrix Factorization)</td>
<td>704</td>
</tr>
<tr>
<td>SGTS and DGT (General Tridiagonal Matrix Solve)</td>
<td>706</td>
</tr>
<tr>
<td>SGTNP, DGTPN, CGTNP, and ZGTNP (General Tridiagonal Matrix Combined Factorization and Solve with No Pivoting)</td>
<td>708</td>
</tr>
<tr>
<td>SGTNPF, DGTPNF, CGTNPF, and ZGTNP (General Tridiagonal Matrix Factorization with No Pivoting)</td>
<td>711</td>
</tr>
<tr>
<td>SGTNPS, DGTPNS, CGTNPS, and ZGTNP (General Tridiagonal Matrix Solve with No Pivoting)</td>
<td>714</td>
</tr>
<tr>
<td>SPTF and DPTF (Positive Definite Symmetric Tridiagonal Matrix Factorization)</td>
<td>716</td>
</tr>
<tr>
<td>SPTS and DPTS (Positive Definite Symmetric Tridiagonal Matrix Solve)</td>
<td>718</td>
</tr>
<tr>
<td><strong>Sparse Linear Algebraic Equation Subroutines</strong></td>
<td>719</td>
</tr>
<tr>
<td>DGSF (General Sparse Matrix Factorization Using Storage by Indices, Rows, or Columns)</td>
<td>720</td>
</tr>
<tr>
<td>DGSS (General Sparse Matrix or Its Transpose Solve Using Storage by Indices, Rows, or Columns)</td>
<td>725</td>
</tr>
<tr>
<td>DGKFS (General Sparse Matrix or Its Transpose Factorization, Determinant, and Solve Using Skyline Storage Mode)</td>
<td>729</td>
</tr>
<tr>
<td>DSKFS (Symmetric Sparse Matrix Factorization, Determinant, and Solve Using Skyline Storage Mode)</td>
<td>744</td>
</tr>
<tr>
<td>DSRIS (Iterative Linear System Solver for a General or Symmetric Sparse Matrix Stored by Rows)</td>
<td>744</td>
</tr>
<tr>
<td>DSMCG (Sparse Positive Definite or Negative Definite Symmetric Matrix Iterative Solve Using Compressed-Matrix Storage Mode)</td>
<td>760</td>
</tr>
<tr>
<td>DSDDCG (Sparse Positive Definite or Negative Definite Symmetric Matrix Iterative Solve Using Compressed-Diagonal Storage Mode)</td>
<td>769</td>
</tr>
<tr>
<td>DSMGCG (General Sparse Matrix Iterative Solve Using Compressed-Matrix Storage Mode)</td>
<td>782</td>
</tr>
</tbody>
</table>
Chapter 11. Eigensystem analysis subroutines................................................... 847
Overview of the Eigensystem Analysis Subroutines........................................................................ 847
Performance and Accuracy Considerations..................................................................................... 848
Eigensystem Analysis Subroutines................................................................................................. 849
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). 850
SSYEV, DSYEV, CHEEV, ZHEEV, SSPEVD, DSYEVX, CHPEVX, ZHPEVX, SSYEVD, DSYEVX,
CHEEVD, ZHEEVD, SSYEV, DSYEVR, CHEEVR, ZHEEVR (Eigenvalues, and Optionally,
the Eigenvectors of a Real Symmetric or Complex Hermitian Matrix) ........................................ 867
SSYEVD, DSYEVD, CHEEVD, ZHEEVD, SSYEVD, DSYEVD, CHEEVD, and ZHEEVD (Eigenvalues
and, Optionally the Eigenvectors, of a Real Symmetric or Complex Hermitian Matrix Using a
Divide-and-Conquer Algorithm) .................................................................................................. 893
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).......................................................................................................... 905
SSYGVD, DSYGV, CHEGV, ZHEGV, SSYGVD, DSYGVX, CHEG, ZHEG (Eigenvalues, and Optionally, the Eigenvectors of a
Positive Definite Real Symmetric or Complex Hermitian Generalized Eigenproblem Using a
Divide-and-Conquer Algorithm) .................................................................................................. 922
SSYGV, DSYGV, CHEGV, ZHEGV, SSPGVX, DSPGVX, CHPGVX, ZHPGVX, SSYGV, DSYGVX,
CHEGVX, ZHEGVX (Eigenvalues, and Optionally, the Eigenvectors of a Positive Definite
Real Symmetric or Complex Hermitian Generalized Eigenproblem)........................................ 931

Chapter 12. Fourier Transforms, Convolutions and Correlations, and Related
Computations.............................................................................................................................. 953
Overview of the Signal Processing Subroutines.............................................................................. 953
Fourier Transforms Subroutines..................................................................................................... 953
Convolution and Correlation Subroutines....................................................................................... 954
Related-Computation Subroutines.................................................................................................. 955
Fourier Transforms, Convolutions, and Correlations Considerations.......................................... 955
Use Considerations......................................................................................................................... 955
Initializing Auxiliary Working Storage.......................................................................................... 958
Determining the Amount of Auxiliary Working Storage That You Need.................................... 958
Performance and Accuracy Considerations.................................................................................. 958
When Running on the Workstation Processors............................................................................. 958
Defining Arrays.............................................................................................................................. 958
Fourier Transform Considerations.................................................................................................. 959
How the Fourier Transform Subroutines Achieve High Performance......................................... 960
Convolution and Correlation Considerations.................................................................................. 960
Related Computation Considerations............................................................................................ 961
Accuracy Considerations................................................................................................................ 961
Fourier Transform Subroutines....................................................................................................... 962
SCFTD and DCFTD (Multidimensional Complex Fourier Transform).......................................... 962
Chapter 13. Sorting and searching subroutines .................................................. 1109

Overview of the Sorting and Searching Subroutines ........................................ 1109
Use Considerations .......................................................................................... 1109
Performance and Accuracy Considerations ..................................................... 1109

Sorting and Searching Subroutines

SORT, SORTX, and DSORT (Sort the Elements of a Sequence) ..................... 1110
SORTX, SORSORTX, and DSORTX (Sort the Elements of a Sequence and Note the Original Element Positions) ........................................ 1111
SORTS, SORSORTS, and DSORTS (Sort the Elements of a Sequence Using a Stable Sort and Note the Original Element Positions) ............ 1114
IBSRCH, SBSRCH, and DBSRCH (Binary Search for Elements of a Sequence X in a Sorted Sequence Y) .......................................................... 1117
ISSRCH, SSSRCH, and DSSRCH (Sequential Search for Elements of a Sequence X in the Sequence Y) ............................................................ 1120

Chapter 14. Interpolation subroutines .............................................................. 1125

Overview of the Interpolation Subroutines ..................................................... 1125
Use Considerations .......................................................................................... 1125
Performance and Accuracy Considerations ..................................................... 1125

Interpolation Subroutines

SPINT and DPINT (Polynomial Interpolation) .................................................. 1126
STPINT and DTPINT (Local Polynomial Interpolation) ................................. 1130
SCSINT and DCSINT (Cubic Spline Interpolation) ......................................... 1133
SCSIN2 and DCSIN2 (Two-Dimensional Cubic Spline Interpolation) ............. 1137

Chapter 15. Numerical quadrature subroutines .............................................. 1143

Overview of the Numerical Quadrature Subroutines ..................................... 1143
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appendix C. FFTW Version 3.1.2 to ESSL Wrapper Libraries</td>
<td>1239</td>
</tr>
<tr>
<td>Appendix D. Using ESSL with netlib-java and Python</td>
<td>1243</td>
</tr>
<tr>
<td>Accessibility features for ESSL</td>
<td>1247</td>
</tr>
<tr>
<td>Notices</td>
<td>1249</td>
</tr>
<tr>
<td>Trademarks</td>
<td>1250</td>
</tr>
<tr>
<td>Software update protocol</td>
<td>1250</td>
</tr>
<tr>
<td>Privacy policy considerations</td>
<td>1251</td>
</tr>
<tr>
<td>Bibliography</td>
<td>1253</td>
</tr>
<tr>
<td>Index</td>
<td>1261</td>
</tr>
</tbody>
</table>
# List of Tables

1. Abbreviated names.......................................................... xxiv
2. Summary of typographic conventions................................. xxv
3. Syntax for all subroutines.................................................. xxxii
4. Syntax for CBLAS subroutines........................................... xxxii
5. Syntax for LAPACKE subroutines....................................... xxxii
6. Translating character argument values to CBLAS enumerated types.................................................. xxxiii
7. Summary of ESSL Subroutines........................................... 3
8. Required Software Products for ESSL for Linux on POWER................................................. 6
9. List of Vector-Scalar Linear Algebra Subprograms.................. 7
10. List of Sparse Vector-Scalar Linear Algebra Subprograms......... 10
11. List of Matrix-Vector Linear Algebra Subprograms.................. 11
12. List of Sparse Matrix-Vector Linear Algebra Subprograms........... 13
13. List of Matrix Operation Subroutines.................................. 13
14. List of LAPACK Dense Linear Algebraic Equation Subroutines........................................ 15
15. List of Dense Linear Algebraic Equation Subroutines................ 19
16. List of LAPACK Banded Linear Algebraic Equation Subroutines........................................ 20
17. List of non-LAPACK Banded Linear Algebraic Equation Subroutines......................................... 22
18. List of Sparse Linear Algebraic Equation Subroutines................ 23
19. List of LAPACK Linear Least Squares Subroutines...................... 24
20. List of Non–LAPACK Linear Least Squares Subroutines................. 24
21. List of LAPACK Eigensystem Analysis Subroutines...................... 25
22. List of Fourier Transform Subroutines.................................. 27
23. List of Convolution and Correlation Subroutines...................... 29
24. List of Related-Computation Subroutines............................... 29
25. List of Sorting and Searching Subroutines............................... 30
26. List of Interpolation Subroutines.......................................... 30
27. List of Numerical Quadrature Subroutines.............................. 30
28. List of Random Number Generation Initialization Subroutines............... 31
29. List of Random Number Generation Subroutines....................... 31
30. List of Utility Subroutines.................................................. 32
31. VSX Alignment Requirements for SIMD Algorithms in Linear Algebra Subroutines........................................ 34
32. VSX Alignment Requirements for SIMD Algorithms in Fourier Transform Subroutines and Convolution and Correlation Subroutines........................................ 35
33. ESSL Subroutines that Automatically Use SIMD Algorithms When Alignment Restrictions are Met on VSX-enabled Processors................................................. 37
34. Multithreaded Subroutines.................................................. 39
35. List of ESSL SMP CUDA subroutines that are GPU-enabled................................. 43
36. ESSL Subroutines Requiring Auxiliary Working Storage....................... 50
37. Example of Input-Argument Error Recovery for Auxiliary Storage Sizes................................. 55
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>81. Data Types</td>
<td>265</td>
</tr>
<tr>
<td>82. Data Types</td>
<td>269</td>
</tr>
<tr>
<td>83. Data Types</td>
<td>273</td>
</tr>
<tr>
<td>84. Data Types</td>
<td>277</td>
</tr>
<tr>
<td>85. Data Types</td>
<td>280</td>
</tr>
<tr>
<td>86. Data Types</td>
<td>283</td>
</tr>
<tr>
<td>87. Data Types</td>
<td>285</td>
</tr>
<tr>
<td>88. Data Types</td>
<td>288</td>
</tr>
<tr>
<td>89. Data Types</td>
<td>290</td>
</tr>
<tr>
<td>90. Data Types</td>
<td>293</td>
</tr>
<tr>
<td>91. Data Types</td>
<td>296</td>
</tr>
<tr>
<td>92. Data Types</td>
<td>306</td>
</tr>
<tr>
<td>93. Data Types</td>
<td>313</td>
</tr>
<tr>
<td>94. Data Types</td>
<td>321</td>
</tr>
<tr>
<td>95. Data Types</td>
<td>328</td>
</tr>
<tr>
<td>96. Data Types</td>
<td>336</td>
</tr>
<tr>
<td>97. Data Types</td>
<td>343</td>
</tr>
<tr>
<td>98. Data Types</td>
<td>348</td>
</tr>
<tr>
<td>99. Data Types</td>
<td>354</td>
</tr>
<tr>
<td>100. Data Types</td>
<td>360</td>
</tr>
<tr>
<td>101. Data Types</td>
<td>366</td>
</tr>
<tr>
<td>102. List of Matrix Operation Subroutines</td>
<td>383</td>
</tr>
<tr>
<td>103. Data Types</td>
<td>386</td>
</tr>
<tr>
<td>104. Data Types</td>
<td>392</td>
</tr>
<tr>
<td>105. Data Types</td>
<td>398</td>
</tr>
<tr>
<td>106. Data Types</td>
<td>408</td>
</tr>
<tr>
<td>107. Data Types</td>
<td>413</td>
</tr>
<tr>
<td>108. Data Types</td>
<td>422</td>
</tr>
<tr>
<td>109. Data Types</td>
<td>430</td>
</tr>
<tr>
<td>110. Data Types</td>
<td>437</td>
</tr>
<tr>
<td>111. Data Types</td>
<td>445</td>
</tr>
<tr>
<td>112. Data Types</td>
<td>452</td>
</tr>
<tr>
<td>113. Data Types</td>
<td>459</td>
</tr>
<tr>
<td>114. Data Types</td>
<td>462</td>
</tr>
<tr>
<td>115. List of LAPACK Dense Linear Algebraic Equation Subroutines</td>
<td>467</td>
</tr>
<tr>
<td>116. List of Dense Linear Algebraic Equation Subroutines</td>
<td>471</td>
</tr>
<tr>
<td>117. List of LAPACK Banded Linear Algebraic Equation Subroutines</td>
<td>472</td>
</tr>
<tr>
<td>118. List of non-LAPACK Banded Linear Algebraic Equation Subroutines</td>
<td>474</td>
</tr>
<tr>
<td>119. List of Sparse Linear Algebraic Equation Subroutines</td>
<td>475</td>
</tr>
<tr>
<td>120. List of LAPACK Linear Least Squares Subroutines</td>
<td>476</td>
</tr>
<tr>
<td>121. List of Non-LAPACK Linear Least Squares Subroutines</td>
<td>476</td>
</tr>
<tr>
<td>122. Data Types</td>
<td>480</td>
</tr>
<tr>
<td>123. Syntax for supported computing languages</td>
<td>481</td>
</tr>
</tbody>
</table>
About this information

This document provides an overview of ESSL and guidance information for designing, coding, and processing your program, as well as migrating existing programs, diagnosing problems, and 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. This document also assumes that users are familiar with Fortran, C, and C++ programming.

Related information

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>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 RPM Package Manager (RPM): <a href="http://www.rpm.org/">http://www.rpm.org/</a></td>
</tr>
<tr>
<td>NVIDIA</td>
<td><a href="http://www.nvidia.com">http://www.nvidia.com</a></td>
</tr>
<tr>
<td></td>
<td>For the CUDA Toolkit Documentation site, see: <a href="http://docs.nvidia.com/cuda/#axzz3VafCSAvr">http://docs.nvidia.com/cuda/#axzz3VafCSAvr</a></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.

Interpreting 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 2.

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

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 C BLAS (CBLAS),” on page 1225.

Abbreviated Names

The abbreviated names used are defined below.

<table>
<thead>
<tr>
<th>Short Name</th>
<th>Full Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAS</td>
<td>Basic Linear Algebra Subprograms (see Appendix A, “Basic Linear Algebra</td>
</tr>
<tr>
<td></td>
<td>Subprograms (BLAS) and C BLAS (CBLAS),” on page 1225)</td>
</tr>
<tr>
<td>CBLAS</td>
<td>C interface to the BLAS (see Appendix A, “Basic Linear Algebra Subprograms</td>
</tr>
<tr>
<td></td>
<td>(BLAS) and C BLAS (CBLAS),” on page 1225)</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>FFTW</td>
<td>Fastest Fourier Transform in the West (see Appendix C, “FFTW Version 3.1.2</td>
</tr>
<tr>
<td></td>
<td>to ESSL Wrapper Libraries,” on page 1239)</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics processing unit</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext Markup Language</td>
</tr>
<tr>
<td>Short Name</td>
<td>Full Name</td>
</tr>
<tr>
<td>------------</td>
<td>-----------</td>
</tr>
<tr>
<td>LAPACK</td>
<td>Linear Algebra Package (see Appendix B, “LAPACK and LAPACKE,” on page 1231)</td>
</tr>
<tr>
<td>LAPACKE</td>
<td>C Interface to LAPACK (see Appendix B, “LAPACK and LAPACKE,” on page 1231)</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>

### Conventions and terminology used

Table 2 on page xxv describes the typographic conventions used.

**Table 2: Summary of typographic conventions.**

<table>
<thead>
<tr>
<th>Typographic</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bold</strong></td>
<td>Bold words or characters represent system elements that you must use literally, such as commands, flags, and path names.</td>
</tr>
</tbody>
</table>
| *Italic*      | *Italic* words or characters represent variable values that you must supply.  
*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.
### 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 programs,” on page 125.

<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&quot;</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:
For a definition of vector, see “Vectors” on page 71.

**Matrices**

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

\[
\begin{bmatrix}
    x_1 \\
    x_2 \\
    x_3 \\
    \vdots \\
    x_n
\end{bmatrix}
\]

For a definition of matrix, see “Matrices” on page 77.

**Sequences**

Sequences are used in the areas of sorting, searching, Fourier transforms, convolutions, and correlations. For a definition of sequences, see “Sequences” on page 120.

**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:

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

**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:
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:

\[
a_{11} \ a_{12} \ldots \ a_{1n} \\
a_{21} \ a_{22} \ldots \ a_{2n} \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
a_{m1} \ a_{m2} \ldots \ a_{mn}
\]

**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:

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

**Arrays**

Arrays contain vectors, matrices, or sequences. For a definition of array, see “How Do You Set Up Your Arrays?” on page 47.

**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:

\[(1.0, \ldots, 2.0, \ldots, 3.0, \ldots, 4.0, \ldots, 5.0)\]

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:

\[
\begin{array}{cccc}
1.0 & 11.0 & 5.0 & 25.0 \\
2.0 & 12.0 & 6.0 & 26.0 \\
3.0 & 13.0 & 7.0 & 27.0 \\
4.0 & 14.0 & 8.0 & 28.0
\end{array}
\]
Elements not significant to the computation are usually not shown in the array. One dot appears for each element not shown. The following array contains three rows and two columns not used in the computation:

| .    .    .    .    .    . |
| .    .    .    .    .    . |
| 1.0  2.0  5.0  4.0  .    . |
| 2.0  3.0  6.0  3.0  .    . |
| 3.0  4.0  7.0  2.0  .    . |
| 4.0  5.0  8.0  1.0  .    . |

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

**Three-Dimensional Arrays**

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

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

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

**Special Characters, Symbols, Expressions, and Abbreviations**

The mathematical and programming notations used 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: α, σ, ω, Ω</td>
<td>Symbolic scalar values</td>
</tr>
<tr>
<td>$</td>
<td>a</td>
</tr>
<tr>
<td>$\mathbf{a} \cdot \mathbf{b}$</td>
<td>The dot product of $\mathbf{a}$ and $\mathbf{b}$</td>
</tr>
<tr>
<td>$x_i$</td>
<td>The $i$-th element of vector $\mathbf{x}$</td>
</tr>
<tr>
<td>$c_{ij}$</td>
<td>The element in matrix $\mathbf{C}$ 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>$i = 1, \ldots, n$</td>
<td>$i$ is assigned the values 1 to $n$</td>
</tr>
<tr>
<td>$\mathbf{y} \leftarrow \mathbf{x}$</td>
<td>Vector $\mathbf{y}$ is replaced by vector $\mathbf{x}$</td>
</tr>
<tr>
<td>$\mathbf{x} \mathbf{y}$</td>
<td>Vector $\mathbf{x}$ times vector $\mathbf{y}$</td>
</tr>
<tr>
<td>$\mathbf{A} \cong \mathbf{B}$</td>
<td>$\mathbf{A}$ is congruent to $\mathbf{B}$</td>
</tr>
<tr>
<td>$\alpha^k$</td>
<td>$\alpha$ raised to the $k$ power</td>
</tr>
<tr>
<td>$e^x$</td>
<td>Exponential function of $x$</td>
</tr>
<tr>
<td>$\mathbf{A}^T; \mathbf{x}^T$</td>
<td>The transpose of matrix $\mathbf{A}$; the transpose of vector $\mathbf{x}$</td>
</tr>
<tr>
<td>$\overline{\mathbf{x}}; \overline{\mathbf{A}}$</td>
<td>The complex conjugate of vector $\mathbf{x}$; the complex conjugate of matrix $\mathbf{A}$</td>
</tr>
<tr>
<td>Item</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| $\bar{x}_i$, $\bar{c}_{jk}$ | The complex conjugate of the complex vector element $x_i$, where:  
  $\text{if } x_i = (a_i, b_i), \text{ then } \bar{x}_i = (a_i, -b_i)$  
  The complex conjugate of the complex matrix element $c_{jk}$ |
| $x^H$, $A^H$         | The complex conjugate transpose of vector $x$; the complex conjugate transpose of matrix $A$                                                    |
| $\sum_{i=1}^{n} x_i$ | The sum of elements $x_1$ to $x_n$                                                                                                          |
| $\sqrt{a+b}$         | The square root of $a+b$                                                                                                                   |
| $\int_{a}^{b} f(x)\, dx$ | The integral from $a$ to $b$ of $f(x)\, dx$                                                                                               |
| $\|x\|_2$            | The Euclidean norm of vector $x$, defined as:  
  $\sqrt{\sum_{j=1}^{n} |x_j|^2}$                                                                                                             |
| $\|A\|_1$            | The one norm of matrix $A$, defined as:  
  $\max \left\{ \sum_{i=1}^{m} |a_{ij}|, 1 \leq j \leq n \right\}$                                                                              |
| $\|A\|_2$            | The spectral norm of matrix $A$, defined as:  
  $\max \{\|Ax\|_2 : \|x\|_2 = 1\}$                                                                                                          |
| $\|A\|_F$            | The Frobenius or Euclidean norm of matrix $A$, defined as:  
  $\sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2}$                                                                                      |
| $\|A\|_\infty$       | The infinity norm of matrix $A$, defined as:  
  $\max \left\{ \sum_{j=1}^{n} |a_{ij}|, 1 \leq i \leq m \right\}$                                                                 |
<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A^{-1}$</td>
<td>The inverse of matrix $A$</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 \times 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>ceiling($x$)</td>
<td>The smallest integer that is greater than or equal to $x$</td>
</tr>
<tr>
<td>floor($x$)</td>
<td>The largest integer that is not greater than $x$</td>
</tr>
<tr>
<td>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 xxxii shows the syntax for the Fortran, C and C++ calling sequences. All subroutines will include this information.
- Table 4 on page xxxii 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 xxxii 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 [“11” on page 1253].
For a list of BLAS, see Appendix A, “Basic Linear Algebra Subprograms (BLAS) and C BLAS (CBLAS),” on page 1225.

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>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL NAME-1</td>
</tr>
<tr>
<td>C and C++</td>
<td>name-1</td>
</tr>
</tbody>
</table>

Table 4: Syntax for CBLAS subroutines.

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBLAS</td>
<td>cblas_name-1</td>
</tr>
</tbody>
</table>

Table 5: Syntax for LAPACKE subroutines.

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPACKE</td>
<td>LAPACKE_name-1</td>
</tr>
</tbody>
</table>

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 xxxiii.
- 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 179 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', 'T', 'C'</td>
<td>cblas_trans, cblas_transa, cblas_transb</td>
<td>CblasNoTrans, CblasTrans, CblasConjTrans</td>
</tr>
<tr>
<td>side</td>
<td>'L', 'R'</td>
<td>cblas_side</td>
<td>CblasLeft, CblasRight</td>
</tr>
<tr>
<td>uplo</td>
<td>'U', 'L'</td>
<td>cblas_uplo</td>
<td>CblasUpper, CblasLower</td>
</tr>
<tr>
<td>diag</td>
<td>'U', 'N'</td>
<td>cblas_diag</td>
<td>CblasUnit, 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 on page xxxiii). 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
dress 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 51.)
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.

What's new in ESSL Guide and Reference
Read about new or significantly changed information for Engineering and Scientific Subroutine Library
(ESSL) Version 6.1.

How to see what's new or changed
To help you see where technical changes occur, look for the following images:
• The > image to mark where new or changed information begins.
• The < image to mark where new or changed information ends.

August 2018
The following information is a summary of the updates that are made to the ESSL Version 6.1 Guide and
Reference documentation:
• ESSL Version 6.1 supports the following IBM Power Systems™:
  – IBM POWER9™ servers
    Notes:
    - The ESSL SMP CUDA Library is supported only on the IBM Power Systems AC922 (8335-GTC and
      8335-GTW) servers that include Tesla V100 with NVLink GPUs.
    - Red Hat Enterprise Linux 7.5 for Power® Little Endian (LE) (POWER9)
  – IBM POWER8™ servers
    Notes:
    - The ESSL SMP CUDA Library is supported only on IBM Power System S822LC (8335-GTB) servers
      with NVIDIA P100 GPUs.
    - Red Hat Enterprise Linux 7.5 (Little Endian (LE) mode)
  • Added information about the following error messages:
• 2538-2401 and 2538-2402 error messages in the “Resource Error Messages(2400-2499)” on page 196 topic.
• 2538-2701 error message in the “Miscellaneous Error Messages(2700-2799)” on page 196 topic.
• BLAS and CBLAS subroutines:
  – SAXPBY and DAXPBY (Scales Vector X and Vector Y by two Scalars, Adds them to one another, and Stores in the Vector Y)
• LAPACK and LAPACKE subroutines:
  – CLANSY and ZLANSY (Complex Symmetric Matrix Norm)
  – SSYEV, DSYEV, CHEEV, and ZHEEV (Eigenvalues, and Optionally, the Eigenvectors of a Real Symmetric or Complex Hermitian Matrix)
  – SSYGV, DSYGV, CHEG, and ZHEG (Eigenvalues, and Optionally, the Eigenvectors of a Positive Definite Real Symmetric or Complex Hermitian Generalized Eigenproblem Using a Divide-and-Conquer Algorithm)
• LAPACK GPU support:
  – SGESV, DGESV, CGESV, and ZGESV (General Matrix Factorization and Multiple Right-Hand Side Solve)
  – SGETRF, DGETRF, CGETRF, and ZGETRF (General Matrix Factorization)
  – SGETRS, DGETRS, CGETRS, and ZGETRS (General Matrix Multiple Right-Hand Side Solve)
  – SPOSV, DPOSV, COPSV, and ZPOSV (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)
  – SPOTRF, DPOTRF, CPOTRF, and ZPOTRF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)
  – SPOTRS, DPOTRS, CPOTRS, and ZPOTRS (Positive Definite Real Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)
  – SPPSV, DPPSV, CPPSV, and ZPPSV (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)
  – SPPTRF, DPPTRF, CPPTRF, and ZPPTRF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)
  – SPPTRS, DPPTRS, CPPTRS, and ZPPTRS (Positive Definite Real Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)
  – SGQR, DGQR, CGQR, and ZGQR (General Matrix QR Factorization)
  – SGELS, DGELS, CGELS, and ZGELS (Linear Least Squares Solution for a General Matrix)
• Fourier Transforms GPU support:
  – SCFTD and DCFTD (Multidimensional Complex Fourier Transform)
  – SRCFTD and DRCFTD (Multidimensional Real-to-Complex Fourier Transform)
  – SCRFTD and DCRFTD (Multidimensional Complex-to-Real Fourier Transform)
Chapter 1. ESSL concepts

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.

Overview of ESSL

ESSL is provided as run-time libraries that run on the servers and processors listed in “Hardware Products Supported by ESSL” on page 5.

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

To order ESSL, specify the following program number:

**ESSL for Linux**

> 5765-L61 <

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 Linux operating system.

ESSL provides the following run-time libraries (described in detail in “What ESSL Library Do You Want to Use?” on page 33):

- ESSL Serial Libraries and ESSL SMP Libraries, which run in the following environments:
  - 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

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. These ESSL libraries are described in detail in “What ESSL Library Do You Want to Use?” on page 33.
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 on page 3, 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

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 33.
<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><strong>Linear Algebra Subprograms:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector-scalar</td>
<td>0</td>
<td>&gt;144&lt;</td>
<td>&gt;144&lt;</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><strong>Matrix Operations:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addition, subtraction, multiplications,</td>
<td>0</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>triangular solves, rank-k updates, rank-2k updates, and matrix transposes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Linear Algebraic Equations:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dense linear algebraic equations</td>
<td>0</td>
<td>&gt;183&lt;</td>
<td>&gt;183&lt;</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><strong>Eigensystem Analysis:</strong></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>&gt;126&lt;</td>
<td>&gt;126&lt;</td>
</tr>
<tr>
<td><strong>Signal Processing Computations:</strong></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><strong>Sorting and Searching:</strong></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><strong>Interpolation:</strong></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><strong>Numerical Quadrature:</strong></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>
<tr>
<td><strong>Random Number Generation:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generating vectors of uniformly distributed and normally distributed random numbers</td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td><strong>Utilities:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>General service operations</td>
<td>9</td>
<td>&gt;14&lt;</td>
<td>&gt;14&lt;</td>
</tr>
<tr>
<td><strong>Total ESSL Subroutines</strong></td>
<td>15</td>
<td>&gt;1339&lt;</td>
<td>&gt;1350&lt;</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 61.)

For more information on accuracy, see “Getting the Best Accuracy” on page 60.

High Performance of ESSL

The ESSL subroutines have been designed to provide high performance. (See references [“40” on page 1255], [“51” on page 1256], and [“52” on page 1256].)

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 Power CPUs for a subset of the ESSL subroutines. For a list of the multithreaded GPU-enabled subroutines in the ESSL SMP CUDA Library, see Table 34 on page 39.

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 34 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.

For best performance, you must use the OMP_NUM_THREADS environment variable to specify the number of threads that you want to create and bind the threads by using the OMP_PROC_BIND or
OMP_PLACES environment variable. Your performance might be improved by setting the OMP_PLACES environment variable to active.

**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 34.

**SMT Mode**

Simultaneous multithreading (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. Single-threaded (ST) execution mode is also supported.

To obtain best performance for the IBM POWER9 servers, most HPC applications that use matrix multiplication extensively must be run in SMT-4 mode with four threads per core on POWER9 SMT-4 servers and in SMT-8 mode with eight threads per core on POWER9 SMT-8 servers. For example, in most cases, the performance of double precision general matrix multiplication (DGEMM) obtained in SMT-4 mode that uses the ESSL Symmetric Multi-Processing (SMP) Library with four threads that are bound to each core is better than the performance obtained by using one thread that is bound to each core or by running the applications in ST mode or SMT-2 mode.

To obtain the best performance for the IBM POWER8 servers, most HPC applications that use matrix multiplication extensively should use one thread bound to each core.

**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 programs,” on page 125.

**Software and hardware**

Review hardware and software products you can use with ESSL and products that you can use for installing ESSL.

**Hardware Products Supported by ESSL**

ESL for Linux on POWER is supported on IBM POWER8 and POWER9 processor-based servers.

**Notes:**

- The ESSL SMP CUDA Library is supported on the IBM Power System AC922 (8335-GTC and 8335-GTW) servers that include Tesla V100 with NVLink GPUs.
- The ESSL SMP CUDA Library is supported on IBM Power System S822LC (8335-GTB) servers with NVIDIA P100 GPUs.
Operating Systems Supported by ESSL

ESSL is supported in the Linux operating system on the following little endian mode environments:

- Red Hat Enterprise Linux 7.5 for Power Little Endian (POWER9).
- Red Hat Enterprise Linux 7.5 (Little Endian (LE) mode)

Software Products Required by ESSL

ESSL requires the software products listed in Software requirements 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 142 and “C++ Programs” on page 155.

Software requirements

The following table lists the required software products for ESSL for Linux on POWER:

<table>
<thead>
<tr>
<th>Table 8: Required Software Products for ESSL for Linux on POWER.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Required software</strong></td>
</tr>
<tr>
<td>-----------------------</td>
</tr>
<tr>
<td><strong>For Compiling</strong></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><strong>For Linking, Loading, or Running</strong> (See Note 1)</td>
</tr>
<tr>
<td>IBM XL Fortran Runtime Environment for Linux (See Note 2)</td>
</tr>
<tr>
<td>gcc and g++ 64-bit libraries</td>
</tr>
<tr>
<td>CUDA Toolkit (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. You can download the latest version of IBM XL Fortran Runtime Environment and Addons Library for Linux from the Latest updates for supported IBM XL Fortran compilers website.

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 required only to use the ESSL SMP CUDA library.

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.

The ESSL for Linux Installation Guide provides the detailed information you need to install ESSL on the Linux operating system.
**Man pages**

To display a specific man page, run the man command as follows: `man subroutine-name`.

The ESSL man pages are installed in the `/usr/share/man/man3` directory. The BLAS and LAPACK man pages that are provided by the NETLIB open source packages are installed in the `/usr/share/man/man1` directory. By default, the ESSL man pages are displayed rather than the BLAS or LAPACK man pages with the same names. If you want to access the BLAS or LAPACK man pages, you must set the `MANPATH` environment variable.

To display the man pages properly on the Linux operating system, the `LANG` environment variable must be set to either of the following values: `C` or `en_US.ISO8859-15`.

**List of ESSL subroutines**

ESSL provides several different types of subroutines.

Appendix A, “Basic Linear Algebra Subprograms (BLAS) and C BLAS (CBLAS),” on page 1225 contains a list of Level 1, 2, and 3 Basic Linear Algebra Subprograms (BLAS) included in ESSL.

Appendix B, “LAPACK and LAPACKE,” on page 1231 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 7)
- Sparse vector-scalar linear algebra subprograms (“Sparse Vector-Scalar Linear Algebra Subprograms” on page 10)
- Matrix-vector linear algebra subprograms (“Matrix-Vector Linear Algebra Subprograms” on page 11)
- Sparse matrix-vector linear algebra subprograms (“Sparse Matrix-Vector Linear Algebra Subprograms” on page 13)

**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 [“95” on page 1258]. 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>
<thead>
<tr>
<th>Table 9: List of Vector-Scalar Linear Algebra Subprograms.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Short-Precision Subprogram</strong></td>
</tr>
<tr>
<td>ISAMAX†, ICAMAX†, cblas_isamax†, cblas_icamax†</td>
</tr>
<tr>
<td>Short-Precision Subprogram</td>
</tr>
<tr>
<td>-----------------------------</td>
</tr>
<tr>
<td>ISAMIN†</td>
</tr>
<tr>
<td>ISMAX†</td>
</tr>
<tr>
<td>ISMIN†</td>
</tr>
<tr>
<td>SASUM†</td>
</tr>
<tr>
<td>SAXPY†</td>
</tr>
<tr>
<td>SCOPY†</td>
</tr>
<tr>
<td>SDOT†</td>
</tr>
<tr>
<td>SNAXPY</td>
</tr>
<tr>
<td>SNDOT</td>
</tr>
<tr>
<td>SNRM2†</td>
</tr>
<tr>
<td>Short-Precision Subprogram</td>
</tr>
<tr>
<td>----------------------------</td>
</tr>
<tr>
<td>SNORM2† CNORM2†</td>
</tr>
<tr>
<td>SROTG' CROTG' cblas_srotg' cblas_crotg'</td>
</tr>
<tr>
<td>SROT' CROT' CSROT' cblas_rot' cblas_crot' cblas_csrot'</td>
</tr>
<tr>
<td>SROTMG' cblas_srotmg'</td>
</tr>
<tr>
<td>SROTM' cblas_srotm'</td>
</tr>
<tr>
<td>SSCAL' CSCAL' CSSCAL'</td>
</tr>
<tr>
<td>SSWAP' CSWAP' cblas_sswap' cblas_cswap'</td>
</tr>
<tr>
<td>SVEA CVEA cblas_svea cblas_cvea</td>
</tr>
<tr>
<td>SVES CVES cblas_sves cblas_cvces</td>
</tr>
</tbody>
</table>
### Table 9: 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>SVEM CVEM</td>
<td>DVEM ZVEM</td>
<td>“SVEM, DVEM, CVEM, and ZVEM (Multiply a Vector X by a Vector Y and Store in a Vector Z)” on page 273</td>
</tr>
<tr>
<td>cblas_svem cblas_cvem</td>
<td>cblas_dvem cblas_zvem</td>
<td></td>
</tr>
<tr>
<td>SYAX CYAX CSYAX</td>
<td>DYAX ZYAX ZDYAX</td>
<td>“SYAX, DYAX, CYAX, ZYAX, CSYAX, and ZDYAX (Multiply a Vector X by a Scalar and Store in a Vector Y)” on page 276</td>
</tr>
<tr>
<td>cblas_syax cblas_cyax cblas_csyax</td>
<td>cblas_dyax cblas_zyax cblas_zdyax</td>
<td></td>
</tr>
<tr>
<td>SZAXPY CZAXPY</td>
<td>DZAXPY ZZAXPY</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 280</td>
</tr>
<tr>
<td>cblas_szaxpy cblas_czaxpy</td>
<td>cblas_dzaxpy 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 [“39” on page 1255]. Both real and complex versions of the subprograms are provided.

### Table 10: 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 CSCTR</td>
<td>DSCTR ZSCTR</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 283</td>
</tr>
<tr>
<td>SGTHR CGTHR</td>
<td>DGTHR ZGTHR</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 285</td>
</tr>
<tr>
<td>SGTHRZ CGTHRZ</td>
<td>DGTHRZ ZGTHRZ</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 288</td>
</tr>
<tr>
<td>SAXPYI CAXPYI</td>
<td>DAXPYI ZAXPYI</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 290</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

10 IBM Engineering and Scientific Subroutine Library for Linux on POWER  Version 6.1: ESSL Guide and Reference
### Table 10: 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 292</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 [“44” on page 1255] and [“45” on page 1255]. Both real and complex versions of the subprograms are provided.

### Table 11: 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 295</td>
</tr>
<tr>
<td>CGEMV*</td>
<td>ZGEMV*</td>
<td></td>
</tr>
<tr>
<td>SGEXM§</td>
<td>DGEXM§</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 306</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_cger*</td>
<td>cblas_zgeru*</td>
<td></td>
</tr>
<tr>
<td>cblas_cgerc*</td>
<td>cblas_zgerc*</td>
<td></td>
</tr>
<tr>
<td>SSPMV*</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 313</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>DSLMX§</td>
<td></td>
</tr>
<tr>
<td>cblas_sspmvm*</td>
<td>cblas_dspmv*</td>
<td></td>
</tr>
<tr>
<td>cblas_chpmv*</td>
<td>cblas_zhpmv*</td>
<td></td>
</tr>
<tr>
<td>cblas_ssymv*</td>
<td>cblas_dsymv*</td>
<td></td>
</tr>
<tr>
<td>cblas_chemv*</td>
<td>cblas_zhemv*</td>
<td></td>
</tr>
</tbody>
</table>

ESSL concepts 11
<table>
<thead>
<tr>
<th>Short-Precision Subprogram</th>
<th>Long-Precision Subprogram</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<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 321</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>DSLR1§</td>
<td></td>
</tr>
<tr>
<td>cblas_ssp*</td>
<td>cblas_dsp*</td>
<td></td>
</tr>
<tr>
<td>cblas_chpr*</td>
<td>cblas_zhpr*</td>
<td></td>
</tr>
<tr>
<td>cblas_ssy*</td>
<td>cblas_dsy*</td>
<td></td>
</tr>
<tr>
<td>cblas_cher*</td>
<td>cblas_zher*</td>
<td></td>
</tr>
<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 328</td>
</tr>
<tr>
<td>CHPR2*</td>
<td>ZHPR2*</td>
<td></td>
</tr>
<tr>
<td>SSYR2*</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_ssp2*</td>
<td>cblas_dsp2*</td>
<td></td>
</tr>
<tr>
<td>cblas_chpr2*</td>
<td>cblas_zhpr2*</td>
<td></td>
</tr>
<tr>
<td>cblas_ssy2*</td>
<td>cblas_dsy2*</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 336</td>
</tr>
<tr>
<td>GCBMV*</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 343</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 347</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, DTRSV, CTRSV, ZTRSV, STPSV, DTPSV, CTPSV, and ZTPSV (Solution of a Triangular System of Equations with a Single Right-Hand Side)” on page 354</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_ctrsv*</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>
</tbody>
</table>

12 IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and Reference
Table 11: 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 360</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 365</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 12: 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 371</td>
</tr>
<tr>
<td>DSMTM</td>
<td>“DSMTM (Transpose a Sparse Matrix in Compressed-Matrix Storage Mode)” on page 378</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 374</td>
</tr>
</tbody>
</table>

Matrix operations 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 [“42” on page 1255]. The matrix operation subroutines also include the commonly used matrix operations: addition, subtraction, multiplication, and transposition.

Table 13: 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 386</td>
</tr>
<tr>
<td>CGEADD</td>
<td>ZGEADD</td>
<td></td>
</tr>
<tr>
<td>Short-Precision Subroutine</td>
<td>Long-Precision Subroutine</td>
<td>Descriptive Name and Location</td>
</tr>
<tr>
<td>----------------------------</td>
<td>---------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>SGESUB CGESUB</td>
<td>DGESUB ZGESUB</td>
<td>“SGESUB, DGESUB, CGESUB, and ZGESUB (Matrix Subtraction for General Matrices or Their Transposes)” on page 392</td>
</tr>
<tr>
<td>SGEMUL CGEMUL</td>
<td>DGEMUL ZGEMUL DGEMUL$</td>
<td>“SGEMUL, DGEMUL, CGEMUL, and ZGEMUL (Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes)” on page 398</td>
</tr>
<tr>
<td>SGEMMS CGEMMS</td>
<td>DGEMMS ZGEMMS</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 408</td>
</tr>
<tr>
<td>SGEMM GPU CGEMM GPU cblas_sgemm cblas_cgemm</td>
<td>DGEMM GPU ZGEMM GPU cblas_dgemm cblas_zgemm</td>
<td>“SGEMM, DGEMM, CGEMM, and ZGEMM (Combined Matrix Multiplication and Addition for General Matrices, Their Transposes, or Conjugate Transposes)” on page 413</td>
</tr>
<tr>
<td>SSYMM GPU CSYMM GPU CHEMM GPU cblas_ssymm cblas_csymm cblas_chemm</td>
<td>DSYMM GPU ZSYMM GPU ZHEMM GPU cblas_dsymm cblas_zsymm cblas_zhemm</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 422</td>
</tr>
<tr>
<td>STRMM GPU CTRMM GPU cblas_strmm cblas_ctrmm</td>
<td>DTRMM GPU ZTRMM GPU cblas_dtrmm cblas_ztrmm</td>
<td>“STRMM, DTRMM, CTRMM, and ZTRMM (Triangular Matrix-Matrix Product)” on page 430</td>
</tr>
<tr>
<td>STRSM GPU CTRSM GPU cblas_strsm cblas_ctrsm</td>
<td>DTRSM GPU ZTRSM GPU cblas_dtrsm cblas_ztrsm</td>
<td>“STRSM, DTRSM, CTRSM, and ZTRSM (Solution of Triangular Systems of Equations with Multiple Right-Hand Sides)” on page 437</td>
</tr>
<tr>
<td>SSYRK GPU CSYRK GPU CHERK GPU cblas_syrk cblas_csymr cblas_chemr</td>
<td>DSYRK GPU ZSYRK GPU ZHERK GPU cblas_dsyrk cblas_zsymr cblas_zherk</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 445</td>
</tr>
</tbody>
</table>
Table 13: 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>SGETMI CGETMI CGECMI</td>
<td>DGETMI ZGETMI ZGECMI</td>
<td>“SGETMI, DGETMI, CGETMI, ZGETMI, CGECMI and ZGECMI (General Matrix Transpose or Conjugate Transpose [In-Place])” on page 459</td>
</tr>
<tr>
<td>SGETMO CGETMO CGECMO</td>
<td>DGETMO ZGETMO ZGECMO</td>
<td>“SGETMO, DGETMO, CGETMO, ZGETMO, CGECMO, and ZGECMO (General Matrix Transpose or Conjugate Transpose [Out-of-Place])” on page 462</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.

GPU This subroutine can offload computations to the GPUs when the ESSL SMP CUDA Library is used.

Linear algebraic equations subroutines

The linear algebraic equations consist of:

- “Dense Linear Algebraic Equation Subroutines” on page 15
- “Banded Linear Algebraic Equations” on page 20
- “Sparse Linear Algebraic Equations” on page 22
- “Linear Least Squares” on page 23

Note: Some of the linear algebraic equations were designed in accordance with the LAPACK 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 subprograms, the updates could require modifications of the calling application program. For details on LAPACK, see [“9” on page 1253].

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 [ “9” on page 1253].

Table 14: 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>
</table>
| SGE

ESSL concepts 15
<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGETF&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>DGETRF&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SGETRF, DGETRF, CGETRF and ZGETRF (General Matrix Factorization)” on page 484</td>
</tr>
<tr>
<td>CGETRF&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>ZGETRF&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SGETRF, DGETRF, CGETRF and ZGETRF (General Matrix Factorization)” on page 484</td>
</tr>
<tr>
<td>LAPACKE_sgetrf&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>LAPACKE_dgetrf&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SGETRF, DGETRF, CGETRF and ZGETRF (General Matrix Factorization)” on page 484</td>
</tr>
<tr>
<td>LAPACKE_cgetrf&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>LAPACKE_zgetrf&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SGETRF, DGETRF, CGETRF and ZGETRF (General Matrix Factorization)” on page 484</td>
</tr>
<tr>
<td>SGETRS&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>DGETRS&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SGETRS, DGETRS, CGETRS, and ZGETRS (General Matrix Multiple Right-Hand Side Solve)” on page 488</td>
</tr>
<tr>
<td>CGETRS&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>ZGETRS&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SGETRS, DGETRS, CGETRS, and ZGETRS (General Matrix Multiple Right-Hand Side Solve)” on page 488</td>
</tr>
<tr>
<td>LAPACKE_sgetrs&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>LAPACKE_dgetrs&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SGETRS, DGETRS, CGETRS, and ZGETRS (General Matrix Multiple Right-Hand Side Solve)” on page 488</td>
</tr>
<tr>
<td>LAPACKE_cgetrs&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>LAPACKE_zgetrs&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SGETRS, DGETRS, CGETRS, and ZGETRS (General Matrix Multiple Right-Hand Side Solve)” on page 488</td>
</tr>
<tr>
<td>SGECON&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>DGECON&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SGECON, DGECON, CGECON, and ZGECON (Estimate the Reciprocal of the Condition Number of a General Matrix)” on page 504</td>
</tr>
<tr>
<td>CGECON&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>ZGECON&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SGECON, DGECON, CGECON, and ZGECON (Estimate the Reciprocal of the Condition Number of a General Matrix)” on page 504</td>
</tr>
<tr>
<td>LAPACKE_sgecon&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>LAPACKE_dgecon&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SGECON, DGECON, CGECON, and ZGECON (Estimate the Reciprocal of the Condition Number of a General Matrix)” on page 504</td>
</tr>
<tr>
<td>LAPACKE_cgecon&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>LAPACKE_zgecon&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SGECON, DGECON, CGECON, and ZGECON (Estimate the Reciprocal of the Condition Number of a General Matrix)” on page 504</td>
</tr>
<tr>
<td>SGETRI&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>DGETRI&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD (General Matrix Inverse, Condition Number Reciprocal, and Determinant)” on page 511</td>
</tr>
<tr>
<td>CGETRI&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>ZGETRI&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD (General Matrix Inverse, Condition Number Reciprocal, and Determinant)” on page 511</td>
</tr>
<tr>
<td>LAPACKE_sgetri&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>LAPACKE_dgetri&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD (General Matrix Inverse, Condition Number Reciprocal, and Determinant)” on page 511</td>
</tr>
<tr>
<td>LAPACKE_cgetri&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>LAPACKE_zgetri&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD (General Matrix Inverse, Condition Number Reciprocal, and Determinant)” on page 511</td>
</tr>
<tr>
<td>SLANGE&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>DLANGE&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SLANGE, DLANGE, CLANGE, and ZLANGE (General Matrix Norm)” on page 518</td>
</tr>
<tr>
<td>CLANGE&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>ZLANGE&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SLANGE, DLANGE, CLANGE, and ZLANGE (General Matrix Norm)” on page 518</td>
</tr>
<tr>
<td>LAPACKE_slange&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>LAPACKE_clange&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SLANGE, DLANGE, CLANGE, and ZLANGE (General Matrix Norm)” on page 518</td>
</tr>
<tr>
<td>LAPACKE_zlange&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>LAPACKE_zlange&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SLANGE, DLANGE, CLANGE, and ZLANGE (General Matrix Norm)” on page 518</td>
</tr>
<tr>
<td>SPPSV&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>DPPSV&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SPPSV, DPPSV, CPPSV, and ZPPSV (Positive Definite Real Symmetric and Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)” on page 521</td>
</tr>
<tr>
<td>CPPSV&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>ZPPSV&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SPPSV, DPPSV, CPPSV, and ZPPSV (Positive Definite Real Symmetric and Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)” on page 521</td>
</tr>
<tr>
<td>LAPACKE_spps&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>LAPACKE_dpss&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SPPSV, DPPSV, CPPSV, and ZPPSV (Positive Definite Real Symmetric and Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)” on page 521</td>
</tr>
<tr>
<td>LAPACKE_cppsv&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>LAPACKE_zpssv&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SPPSV, DPPSV, CPPSV, and ZPPSV (Positive Definite Real Symmetric and Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)” on page 521</td>
</tr>
<tr>
<td>SPOSV&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>DPOSV&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SPOSV, DPOSV, CPOSV, and ZPOSV (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)” on page 527</td>
</tr>
<tr>
<td>CPOSV&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>ZPOSV&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SPOSV, DPOSV, CPOSV, and ZPOSV (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)” on page 527</td>
</tr>
<tr>
<td>LAPACKE_sposv&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>LAPACKE_dposv&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SPOSV, DPOSV, CPOSV, and ZPOSV (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)” on page 527</td>
</tr>
<tr>
<td>LAPACKE_zposv&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>LAPACKE_zposv&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SPOSV, DPOSV, CPOSV, and ZPOSV (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)” on page 527</td>
</tr>
<tr>
<td>SPOTRF&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>DPOTRF&lt;sup&gt;Δ&lt;/sup&gt;</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 533</td>
</tr>
<tr>
<td>CPOTRF&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>ZPOTRF&lt;sup&gt;Δ&lt;/sup&gt;</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 533</td>
</tr>
<tr>
<td>SPPTRF&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>DPPTRF&lt;sup&gt;Δ&lt;/sup&gt;</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 533</td>
</tr>
<tr>
<td>CPPTRF&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>ZPPTRF&lt;sup&gt;Δ&lt;/sup&gt;</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 533</td>
</tr>
<tr>
<td>LAPACKEスポット trifΔ&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>LAPACKE_dpotrfΔ&lt;sup&gt;Δ&lt;/sup&gt;</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 533</td>
</tr>
<tr>
<td>LAPACKE_zpotrfΔ&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>LAPACKE_zpotrfΔ&lt;sup&gt;Δ&lt;/sup&gt;</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 533</td>
</tr>
<tr>
<td>LAPACKEスポット trifΔ&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>LAPACKE_dpptrfΔ&lt;sup&gt;Δ&lt;/sup&gt;</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 533</td>
</tr>
<tr>
<td>LAPACKE_zpotrfΔ&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>LAPACKE_zpotrfΔ&lt;sup&gt;Δ&lt;/sup&gt;</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 533</td>
</tr>
</tbody>
</table>
Table 14: 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>SPOTRS&lt;sub&gt;GPU&lt;/sub&gt;</td>
<td>DPOTRS&lt;sub&gt;GPU&lt;/sub&gt;</td>
<td>“SPOTRS, DPOTRS” on page 545</td>
</tr>
<tr>
<td>CPOTRS&lt;sub&gt;GPU&lt;/sub&gt;</td>
<td>ZPOTRS&lt;sub&gt;GPU&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>SPPTRS&lt;sub&gt;GPU&lt;/sub&gt;</td>
<td>DPPTRS&lt;sub&gt;GPU&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>CPPTRS&lt;sub&gt;GPU&lt;/sub&gt;</td>
<td>ZPPTRS&lt;sub&gt;GPU&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spotrs&lt;sub&gt;A&lt;/sub&gt;</td>
<td>LAPACKE_dpotrs&lt;sub&gt;A&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpotrs&lt;sub&gt;A&lt;/sub&gt;</td>
<td>LAPACKE_zpotrs&lt;sub&gt;A&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spptrs&lt;sub&gt;A&lt;/sub&gt;</td>
<td>LAPACKE_dpptrs&lt;sub&gt;A&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpptrs&lt;sub&gt;A&lt;/sub&gt;</td>
<td>LAPACKE_zpptrs&lt;sub&gt;A&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>SPOCON&lt;sub&gt;A&lt;/sub&gt;</td>
<td>DPOCON&lt;sub&gt;A&lt;/sub&gt;</td>
<td>“SPOCON, DPOCON, ZPOCON” on page 555</td>
</tr>
<tr>
<td>CPPCON&lt;sub&gt;A&lt;/sub&gt;</td>
<td>ZPPCON&lt;sub&gt;A&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spocon&lt;sub&gt;A&lt;/sub&gt;</td>
<td>LAPACKE_dpocon&lt;sub&gt;A&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_zpocon&lt;sub&gt;A&lt;/sub&gt;</td>
<td>LAPACKE_dpcon&lt;sub&gt;A&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sppcon&lt;sub&gt;A&lt;/sub&gt;</td>
<td>LAPACKE_zppcon&lt;sub&gt;A&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cppcon&lt;sub&gt;A&lt;/sub&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DPOTRI&lt;sub&gt;A&lt;/sub&gt;</td>
<td>ZPOTRI&lt;sub&gt;A&lt;/sub&gt;</td>
<td>“DPOTRI, ZPOTRI, SPOICD, DPOICD” on page 569</td>
</tr>
<tr>
<td>ZPPTRI&lt;sub&gt;A&lt;/sub&gt;</td>
<td>ZPPTRI&lt;sub&gt;A&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_dpotri&lt;sub&gt;A&lt;/sub&gt;</td>
<td>LAPACKE_zpotri&lt;sub&gt;A&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_dptri&lt;sub&gt;A&lt;/sub&gt;</td>
<td>LAPACKE_zpptri&lt;sub&gt;A&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_zpptri&lt;sub&gt;A&lt;/sub&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLANSY&lt;sub&gt;A&lt;/sub&gt;</td>
<td>ZLANSY&lt;sub&gt;A&lt;/sub&gt;</td>
<td>“SLANSY, ZLANSY, SLANHE, ZLANHE” on page 579</td>
</tr>
<tr>
<td>CLANHE&lt;sub&gt;A&lt;/sub&gt;</td>
<td>ZLANHE&lt;sub&gt;A&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>SLANSP&lt;sub&gt;A&lt;/sub&gt;</td>
<td>ZLANSP&lt;sub&gt;A&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>CLANHP&lt;sub&gt;A&lt;/sub&gt;</td>
<td>ZLANHP&lt;sub&gt;A&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_slansy&lt;sub&gt;A&lt;/sub&gt;</td>
<td>LAPACKE_zlansy&lt;sub&gt;A&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_zlanhe&lt;sub&gt;A&lt;/sub&gt;</td>
<td>LAPACKE_zlanhp&lt;sub&gt;A&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_slansp&lt;sub&gt;A&lt;/sub&gt;</td>
<td>LAPACKE_zlanhp&lt;sub&gt;A&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_clanhe&lt;sub&gt;Δ&lt;/sub&gt;</td>
<td>LAPACKE_clanhp&lt;sub&gt;Δ&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>Short-Precision Subroutine</td>
<td>Long-Precision Subroutine</td>
<td>Descriptive Name and Location</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>SSYSV(\Delta)</td>
<td>DSYSV(\Delta)</td>
<td>“SSYSV, DSYSV, CSYSV, ZSYSV, CHESV, ZHESV, SSPSV, DSPSV, CPSV, ZPSV, CHPSV, and ZHPSV (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)” on page 584</td>
</tr>
<tr>
<td>CSYSV(\Delta)</td>
<td>ZSYSV(\Delta)</td>
<td></td>
</tr>
<tr>
<td>CHESV(\Delta)</td>
<td>ZHESV(\Delta)</td>
<td></td>
</tr>
<tr>
<td>SSPSV(\Delta)</td>
<td>DSPSV(\Delta)</td>
<td></td>
</tr>
<tr>
<td>CPSV(\Delta)</td>
<td>ZSPSV(\Delta)</td>
<td></td>
</tr>
<tr>
<td>CHPSV(\Delta)</td>
<td>ZHPSV(\Delta)</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssysv(\Delta)</td>
<td>LAPACKE_dsysv(\Delta)</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_csysv(\Delta)</td>
<td>LAPACKE_zsysv(\Delta)</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chesv(\Delta)</td>
<td>LAPACKE_zhesv(\Delta)</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sspsv(\Delta)</td>
<td>LAPACKE_dspsv(\Delta)</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cspsv(\Delta)</td>
<td>LAPACKE_zspsv(\Delta)</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chpsv(\Delta)</td>
<td>LAPACKE_zhpsv(\Delta)</td>
<td></td>
</tr>
<tr>
<td>DSYTRF(\Delta)</td>
<td>ZSYTRF(\Delta)</td>
<td>“SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, ZHETRF, SSPTRF, DSPTRF, CPSTRF, ZSPTRF, CHPTRF, and ZH PTRF (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Factorization)” on page 593</td>
</tr>
<tr>
<td>CSYTRF(\Delta)</td>
<td>ZSYTRF(\Delta)</td>
<td></td>
</tr>
<tr>
<td>CHETRF(\Delta)</td>
<td>ZHETRF(\Delta)</td>
<td></td>
</tr>
<tr>
<td>SSPTRF(\Delta)</td>
<td>DSPTRF(\Delta)</td>
<td></td>
</tr>
<tr>
<td>CPSTRF(\Delta)</td>
<td>ZSPTRF(\Delta)</td>
<td></td>
</tr>
<tr>
<td>CHPTRF(\Delta)</td>
<td>ZHPTRF(\Delta)</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sstyrf(\Delta)</td>
<td>LAPACKE_dsytrf(\Delta)</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cstyrf(\Delta)</td>
<td>LAPACKE_zsytrf(\Delta)</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chetrf(\Delta)</td>
<td>LAPACKE_zhetrf(\Delta)</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssptrfs(\Delta)</td>
<td>LAPACKE_dsptrfs(\Delta)</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_csptrfs(\Delta)</td>
<td>LAPACKE_zsptrfs(\Delta)</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chptrf(\Delta)</td>
<td>LAPACKE_zhptrfs(\Delta)</td>
<td></td>
</tr>
<tr>
<td>SYTRS(\Delta)</td>
<td>SYTR(\Delta)</td>
<td>“SSYTRS, DSYTRS, CSYTRS, ZSYTRS, CHETRS, ZHETRS, SSPTRS, DSPTRS, CPSTRS, ZSPTRS, CHPTRS, and ZH PTRS (Indefinite Real or Complex Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)” on page 601</td>
</tr>
<tr>
<td>CSYTR(\Delta)</td>
<td>ZSYTR(\Delta)</td>
<td></td>
</tr>
<tr>
<td>CHETR(\Delta)</td>
<td>ZHETR(\Delta)</td>
<td></td>
</tr>
<tr>
<td>SSPTR(\Delta)</td>
<td>DSPTR(\Delta)</td>
<td></td>
</tr>
<tr>
<td>CPSTR(\Delta)</td>
<td>ZSPTR(\Delta)</td>
<td></td>
</tr>
<tr>
<td>CHPTR(\Delta)</td>
<td>ZHPTR(\Delta)</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sstyr(\Delta)</td>
<td>LAPACKE_dsytr(\Delta)</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cstyr(\Delta)</td>
<td>LAPACKE_zsytr(\Delta)</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chet(\Delta)</td>
<td>LAPACKE_zhet(\Delta)</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssptrfs(\Delta)</td>
<td>LAPACKE_dsptrfs(\Delta)</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_csptrfs(\Delta)</td>
<td>LAPACKE_zsptrfs(\Delta)</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chptr(\Delta)</td>
<td>LAPACKE_zhptrfs(\Delta)</td>
<td></td>
</tr>
<tr>
<td>STRTRI(\Delta)</td>
<td>DTRTRI(\Delta)</td>
<td>“STRTRI, DTRTRI, CTRTRI, ZTRTRI, STPTRI, DTPTRI, CTPTRI, and ZTPTRI (Triangular Matrix Inverse)” on page 621</td>
</tr>
<tr>
<td>CTRTRI(\Delta)</td>
<td>ZTRTRI(\Delta)</td>
<td></td>
</tr>
<tr>
<td>STPTRI(\Delta)</td>
<td>DTPTRI(\Delta)</td>
<td></td>
</tr>
<tr>
<td>CTPTRI(\Delta)</td>
<td>ZTPTRI(\Delta)</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_strtr(\Delta)</td>
<td>LAPACKE_dtrtr(\Delta)</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_stptr(\Delta)</td>
<td>LAPACKE_dtprtr(\Delta)</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ctrtr(\Delta)</td>
<td>LAPACKE_ztprtr(\Delta)</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ctptr(\Delta)</td>
<td>LAPACKE_ztptr(\Delta)</td>
<td></td>
</tr>
</tbody>
</table>
Table 14: 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>SLANTR^Δ</td>
<td>DLANTR^Δ</td>
<td>“SLANTR, DLANTR, CLANTR, ZLANTR, SLANTP, DLANTP, CLANTP, and ZLANTP (Trapezoidal or Triangular Matrix Norm)” on page 628</td>
</tr>
<tr>
<td>CLANTR^Δ</td>
<td>ZLANTR^Δ</td>
<td></td>
</tr>
<tr>
<td>SLANTP^Δ</td>
<td>DLANTP^Δ</td>
<td></td>
</tr>
<tr>
<td>CLANTP^Δ</td>
<td>ZLANTP^Δ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_slantr^Δ</td>
<td>LAPACKE_dlanr^Δ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_clantr^Δ</td>
<td>LAPACKE_zlantr^Δ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_slantp^Δ</td>
<td>LAPACKE_dlantp^Δ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_clantp^Δ</td>
<td>LAPACKE_zlantp^Δ</td>
<td></td>
</tr>
</tbody>
</table>

^Δ LAPACK

>^GPU This subroutine can offload computations to the GPUs when the ESSL SMP CUDA Library is used.<

Table 15: 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>DGGEF</td>
<td>“SGEF, DGEF, CGEF, and ZGEF (General Matrix Factorization)” on page 493</td>
</tr>
<tr>
<td>CGEF</td>
<td>ZGEF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DGGEFP^€</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 499</td>
</tr>
<tr>
<td>CGESM</td>
<td>ZGESM</td>
<td></td>
</tr>
<tr>
<td>SGEES</td>
<td>DGES</td>
<td>“SGES, DGES, CGES, and ZGES (General Matrix, Its Transpose, or Its Conjugate Transpose Solve)” on page 496</td>
</tr>
<tr>
<td>CGES</td>
<td>ZGES</td>
<td></td>
</tr>
<tr>
<td>SGEFCD</td>
<td>DGGEFCD</td>
<td>“SGEFCD and DGEFCD (General Matrix Factorization, Condition Number Reciprocal, and Determinant)” on page 508</td>
</tr>
<tr>
<td>SGEICD</td>
<td>DGGEICD</td>
<td>“SGEICD and DGEGICD (General Matrix Factorization, Condition Number Reciprocal, and Determinant)” on page 511</td>
</tr>
<tr>
<td>SPOF</td>
<td>DPOTRF</td>
<td>“SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPFF, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)” on page 533</td>
</tr>
<tr>
<td>CPOF</td>
<td>ZPOF</td>
<td></td>
</tr>
<tr>
<td>SPPF</td>
<td>DPPF</td>
<td></td>
</tr>
<tr>
<td>SPPFCD</td>
<td>DPPFCD</td>
<td>“SPPFCD, DPPFCD, SPOFCD, and DPOFCD (Positive Definite Real Symmetric Matrix Factorization, Condition Number Reciprocal, and Determinant)” on page 563</td>
</tr>
<tr>
<td>SPPS</td>
<td>DPPS</td>
<td>“SPPS and DPPS (Positive Definite Real Symmetric Matrix Solve)” on page 552</td>
</tr>
<tr>
<td>SPPFCD</td>
<td>DPPFCD</td>
<td>“SPPFCD, DPPFCD, SPOFCD, and DPOFCD (Positive Definite Real Symmetric Matrix Factorization, Condition Number Reciprocal, and Determinant)” on page 563</td>
</tr>
</tbody>
</table>

ESSL concepts 19
Table 15: 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>SPPICD</td>
<td>DPPICD</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 569</td>
</tr>
<tr>
<td>DBSSV</td>
<td></td>
<td>“DBSSV (Symmetric Indefinite Matrix Factorization and Multiple Right-Hand Side Solve)” on page 607</td>
</tr>
<tr>
<td>DBSTRF</td>
<td></td>
<td>“DBSTRF (Symmetric Indefinite Matrix Factorization)” on page 613</td>
</tr>
<tr>
<td>DBSTRS</td>
<td></td>
<td>“DBSTRS (Symmetric Indefinite Matrix Multiple Right-Hand Side Solve)” on page 617</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 621</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 16: 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>SGBSVΔ CGBSVΔ</td>
<td>DGBSVΔ</td>
<td>“SGBSV, DGBSV, CGBSV, and ZGBSV (General Band Matrix Factorization and Multiple Right-Hand Side Solve)” on page 635</td>
</tr>
<tr>
<td>LAPACKE_sgsbvΔ LAPACKE_cgsbvΔ</td>
<td>LAPACKE_dgsbvΔ LAPACKE_zgsbvΔ</td>
<td></td>
</tr>
<tr>
<td>SGBTRFΔ CGBTRFΔ</td>
<td>DGBTRFΔ</td>
<td>“SGBTRF, DGBTRF, CGBTRF and ZGBTRF (General Band Matrix Factorization)” on page 639</td>
</tr>
<tr>
<td>LAPACKE_sgbtrfΔ LAPACKE_cgbtrfΔ</td>
<td>LAPACKE_dgbtrfΔ LAPACKE_zgbtrfΔ</td>
<td></td>
</tr>
<tr>
<td>SGBTRSΔ CGBTRSΔ</td>
<td>DGBTRSΔ</td>
<td>“SGBTRS, DGBTRS, CGBTRS, and ZGBTRS (General Band Matrix Multiple Right-Hand Side Solve)” on page 643</td>
</tr>
<tr>
<td>LAPACKE_sgbtrsΔ LAPACKE_cgbtrsΔ</td>
<td>LAPACKE_dgbtrsΔ LAPACKE_zgbtrsΔ</td>
<td></td>
</tr>
</tbody>
</table>
Table 16: 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>SPBSVΔ</td>
<td>DPBSVΔ</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 651</td>
</tr>
<tr>
<td>CPBSVΔ</td>
<td>ZPBSVΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spbsvΔ</td>
<td>LAPACKE_dpbsvΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpbsvΔ</td>
<td>LAPACKE_zpbsvΔ</td>
<td></td>
</tr>
<tr>
<td>DPBTRFΔ</td>
<td>ZPBTRFΔ</td>
<td>“SPBTRF, DPBTRF, CPBTRF, and ZPBTRF (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization)” on page 656</td>
</tr>
<tr>
<td>CPBTRFΔ</td>
<td>ZPBTRFΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spbtrfΔ</td>
<td>LAPACKE_dpbrfΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpbtrfΔ</td>
<td>LAPACKE_zpbtrfΔ</td>
<td></td>
</tr>
<tr>
<td>SPBTRSΔ</td>
<td>DPBTRSΔ</td>
<td>“SPBTRS, DPBTRS, CPBTRS, and ZPBTRS (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Multiple Right-Hand Side Solve)” on page 661</td>
</tr>
<tr>
<td>CPBTRSΔ</td>
<td>ZPBTRSΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spbtrsΔ</td>
<td>LAPACKE_dpbrfΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpbtrsΔ</td>
<td>LAPACKE_zpbtrfΔ</td>
<td></td>
</tr>
<tr>
<td>SGTSVΔ</td>
<td>DGTSVΔ</td>
<td>“SGTSV, DGTSV, CGTSV, and ZGTSV (General Tridiagonal Matrix Factorization and Multiple Right-Hand Side Solve)” on page 666</td>
</tr>
<tr>
<td>CGTSVΔ</td>
<td>ZGTSVΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgtsvΔ</td>
<td>LAPACKE_sgtsvΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgtsvΔ</td>
<td>LAPACKE_cgtsvΔ</td>
<td></td>
</tr>
<tr>
<td>SGTTTRFΔ</td>
<td>DGTTTRFΔ</td>
<td>“SGTTTRF, DGTTTRF, CGTTTRF, and ZGTTTRF (General Tridiagonal Matrix Factorization)” on page 670</td>
</tr>
<tr>
<td>CGTTTRFΔ</td>
<td>ZGTTTRFΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgtttrfΔ</td>
<td>LAPACKE_dgttrfΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgtttrfΔ</td>
<td>LAPACKE_zgtttrfΔ</td>
<td></td>
</tr>
<tr>
<td>SGTTTRSΔ</td>
<td>DGTTTRSΔ</td>
<td>“SGTTTRS, DGTTTRS, CGTTTRS, and ZGTTTRS (General Tridiagonal Matrix Multiple Right-Hand Side Solve)” on page 674</td>
</tr>
<tr>
<td>CGTTTRSΔ</td>
<td>ZGTTTRSΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgtttrsΔ</td>
<td>LAPACKE_dgttrsΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgtttrsΔ</td>
<td>LAPACKE_zgtttrsΔ</td>
<td></td>
</tr>
<tr>
<td>SPTSVΔ</td>
<td>DPTSVΔ</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 679</td>
</tr>
<tr>
<td>CPTSVΔ</td>
<td>ZPTSVΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sptsvΔ</td>
<td>LAPACKE_dptsvΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cptsvΔ</td>
<td>LAPACKE_zptsvΔ</td>
<td></td>
</tr>
<tr>
<td>SPTTRFΔ</td>
<td>DPTTRFΔ</td>
<td>“SPTTRF, DPTTRF, CPTTRF, and ZPTTRF (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization)” on page 684</td>
</tr>
<tr>
<td>CPTTRFΔ</td>
<td>ZPTTRFΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spttrfΔ</td>
<td>LAPACKE_dpttrfΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpttrfΔ</td>
<td>LAPACKE_zpttrfΔ</td>
<td></td>
</tr>
</tbody>
</table>
Table 16: 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>SPTTRSΔ</td>
<td>DPTTRSΔ</td>
<td>“SPTTRS, DPTTRS, CPTTRS, and ZPTTRS (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Multiple Right-Hand Solve)” on page 687</td>
</tr>
<tr>
<td>CPTTRSΔ</td>
<td>ZPTTRSΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spttrsΔ</td>
<td>LAPACKE_dpttrsΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpttrsΔ</td>
<td>LAPACKE_zpttrsΔ</td>
<td></td>
</tr>
</tbody>
</table>

Δ LAPACK

Table 17: 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 693</td>
</tr>
<tr>
<td>SGBS§</td>
<td>DGBS§</td>
<td>“SGBS and DGBS (General Band Matrix Solve)” on page 649</td>
</tr>
<tr>
<td>SPBF§</td>
<td>DPBF§</td>
<td>“SPBF, DPBF, SPBCHF, and DPBCHF (Positive Definite Symmetric Band Matrix Factorization)” on page 698</td>
</tr>
<tr>
<td>SPBCHF§</td>
<td>DPBCHF§</td>
<td></td>
</tr>
<tr>
<td>SPBS§</td>
<td>DPBS§</td>
<td>“SPBS, DPBS, SPBCHS, and DPBCHS (Positive Definite Symmetric Band Matrix Solve)” on page 701</td>
</tr>
<tr>
<td>SPBCHS§</td>
<td>DPBCHS§</td>
<td></td>
</tr>
<tr>
<td>SGTF§</td>
<td>DGTF§</td>
<td>“SGTF and DGTF (General Tridiagonal Matrix Factorization)” on page 704</td>
</tr>
<tr>
<td>SGTS§</td>
<td>DGTS§</td>
<td>“SGTS and DGTS (General Tridiagonal Matrix Solve)” on page 706</td>
</tr>
<tr>
<td>SGTNP</td>
<td>DGTNP</td>
<td>“SGTNP, DGTNP, CGTNP, and ZGTNP (General Tridiagonal Matrix Combined Factorization and Solve with No Pivoting)” on page 708</td>
</tr>
<tr>
<td>CGTNP</td>
<td>ZGTNP</td>
<td></td>
</tr>
<tr>
<td>SGTNPF</td>
<td>DGTNPF</td>
<td>“SGTNPF, DGTNPF, CGTNPF, and ZGTNPF (General Tridiagonal Matrix Factorization with No Pivoting)” on page 711</td>
</tr>
<tr>
<td>CGTNPF</td>
<td>ZGTNPF</td>
<td></td>
</tr>
<tr>
<td>SGTNPS</td>
<td>DGTNPS</td>
<td>“SGTNPS, DGTNPS, CGTNPS, and ZGTNPS (General Tridiagonal Matrix Solve with No Pivoting)” on page 714</td>
</tr>
<tr>
<td>CGTNPS</td>
<td>ZGTNPS</td>
<td></td>
</tr>
<tr>
<td>SPTF§</td>
<td>DPTF§</td>
<td>“SPTF and DPTF (Positive Definite Symmetric Tridiagonal Matrix Factorization)” on page 716</td>
</tr>
<tr>
<td>SPTS§</td>
<td>DPTS§</td>
<td>“SPTS and DPTS (Positive Definite Symmetric Tridiagonal Matrix Solve)” on page 718</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 18: 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 720</td>
</tr>
<tr>
<td>DGSS</td>
<td>“DGSS (General Sparse Matrix or Its Transpose Solve Using Storage by Indices, Rows, or Columns)” on page 725</td>
</tr>
<tr>
<td>DGKFS</td>
<td>“DGKFS (General Sparse Matrix or Its Transpose Factorization, Determinant, and Solve Using Skyline Storage Mode)” on page 729</td>
</tr>
<tr>
<td>DSKFS</td>
<td>“DSKFS (Symmetric Sparse Matrix Factorization, Determinant, and Solve Using Skyline Storage Mode)” on page 744</td>
</tr>
<tr>
<td>DSRIS</td>
<td>“DSRIS (Iterative Linear System Solver for a General or Symmetric Sparse Matrix Stored by Rows)” on page 760</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 769</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 776</td>
</tr>
<tr>
<td>DSMGCG‡</td>
<td>“DSMGCG (General Sparse Matrix Iterative Solve Using Compressed-Matrix Storage Mode)” on page 782</td>
</tr>
<tr>
<td>DSDGCG</td>
<td>“DSDGCG (General Sparse Matrix Iterative Solve Using Compressed-Diagonal Storage Mode)” on page 788</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

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 [“9” on page 1253].
### Table 19: 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>SGESVDΔ</td>
<td>DGESVDΔ</td>
<td>“SGESVD, DGESVD, CGESVD, ZGESVD, SGESDD, DGESDD, CGESDD, and ZGESDD (Singular Value Decomposition for a General Matrix)” on page 794</td>
</tr>
<tr>
<td>CGESVDΔ</td>
<td>ZGESVDΔ</td>
<td></td>
</tr>
<tr>
<td>SGESDDΔ</td>
<td>DGESDDΔ</td>
<td></td>
</tr>
<tr>
<td>CGESDDΔ</td>
<td>ZGESDDΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKESgesvdΔ</td>
<td>LAPACKE_dgesvdΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKECgesvdΔ</td>
<td>LAPACKE_zgesvdΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKESgesddΔ</td>
<td>LAPACKE_dgesddΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKECgesddΔ</td>
<td>LAPACKE_zgesddΔ</td>
<td></td>
</tr>
<tr>
<td>DGESVDΔ</td>
<td>ZGESVDΔ</td>
<td></td>
</tr>
<tr>
<td>ZGESVDΔ</td>
<td>DGESDDΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKEDgesvdΔ</td>
<td>LAPACKE_dgesvdΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKEZgesvdΔ</td>
<td>LAPACKE_zgesvdΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKEDgesddΔ</td>
<td>LAPACKE_dgesddΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKEZgesddΔ</td>
<td>LAPACKE_zgesddΔ</td>
<td></td>
</tr>
<tr>
<td>GEPQRFΔ</td>
<td>ZPGQRFΔ</td>
<td></td>
</tr>
<tr>
<td>CGEPQRFΔ</td>
<td>ZBGQRFΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKESgeqrfΔ</td>
<td>LAPACKE_dgeqrfΔ</td>
<td>“SGEQRF, DGEQRF, CGEQRF, and ZGEQRF (General Matrix QR Factorization)” on page 807</td>
</tr>
<tr>
<td>LAPACKECgeqrfΔ</td>
<td>LAPACKE_zgeqrfΔ</td>
<td></td>
</tr>
<tr>
<td>DGELSΔ GPU</td>
<td>ZGELSΔ GPU</td>
<td>“SGELS, DGELS, CGELS, and ZGELS (Linear Least Squares Solution for a General Matrix)” on page 812</td>
</tr>
<tr>
<td>CGELSΔ GPU</td>
<td>ZCGELSΔ GPU</td>
<td></td>
</tr>
<tr>
<td>LAPACKESgelsΔ</td>
<td>LAPACKE_dgelsΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKECgelsΔ</td>
<td>LAPACKE_zgelsΔ</td>
<td></td>
</tr>
<tr>
<td>SGELSDΔ</td>
<td>ZGELSDΔ</td>
<td>“SGELSD, DGELSD, CGELSD, and ZGELSD (Linear Least Squares Solution for a General Matrix Using the Singular Value Decomposition)” on page 822</td>
</tr>
<tr>
<td>CGELSDΔ</td>
<td>ZCGELSDΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKESgelsdΔ</td>
<td>LAPACKE_dgelsdΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKECgelsdΔ</td>
<td>LAPACKE_zgelsdΔ</td>
<td></td>
</tr>
</tbody>
</table>

Δ LAPACK

> GPU This subroutine can offload computations to the GPUs when the ESSL SMP CUDA Library is used.

### Table 20: 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>SGESVF§</td>
<td>DGESVF§</td>
<td>“SGESVF and DGESVF (Singular Value Decomposition for a General Matrix)” on page 830</td>
</tr>
<tr>
<td>SGESVS§</td>
<td>DGESVS§</td>
<td>“SGESVS and DGESVS (Linear Least Squares Solution for a General Matrix Using the Singular Value Decomposition)” on page 837</td>
</tr>
<tr>
<td>SGELLS§</td>
<td>DGELLS§</td>
<td>“SGELLS and DGELLS (Linear Least Squares Solution for a General Matrix with Column Pivoting)” on page 841</td>
</tr>
</tbody>
</table>
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 [“9” on page 1253].

Table 21: 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>SGEEVΔ</td>
<td>DGEEVΔ</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 850</td>
</tr>
</tbody>
</table>
Table 21: 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>SSYEVΔ</td>
<td>DSYEVΔ</td>
<td>“SSYEV, DSYEV, CHEEV, ZHEEV, SSPEVX, DSPEVX, CHPEVX, ZHPEVX, SSYEVD, DSYEVD, CHEEVD, and ZHEEVD (Eigenvalues and, Optionally, the Eigenvectors of a Real Symmetric or Complex Hermitian Matrix) ” on page 893</td>
</tr>
<tr>
<td>CHEEVΔ</td>
<td>ZHEEVΔ</td>
<td></td>
</tr>
<tr>
<td>SSPEVXΔ</td>
<td>DSPEVXΔ</td>
<td></td>
</tr>
<tr>
<td>CHPEVXΔ</td>
<td>ZHPEVXΔ</td>
<td></td>
</tr>
<tr>
<td>SSYEVXΔ</td>
<td>DSYEVXΔ</td>
<td></td>
</tr>
<tr>
<td>CHEEVXΔ</td>
<td>ZHEEVXΔ</td>
<td></td>
</tr>
<tr>
<td>SSYEVRΔ</td>
<td>DSYEVRΔ</td>
<td></td>
</tr>
<tr>
<td>CHEEVRΔ</td>
<td>ZHEEVRΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssyevΔ</td>
<td>LAPACKE_dsyevΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cheevΔ</td>
<td>LAPACKE_zheevevΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sspevx</td>
<td>LAPACKE_dspevxΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chpevx</td>
<td>LAPACKE_zhepevxΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssyevx</td>
<td>LAPACKE_dsyevxΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cheevx</td>
<td>LAPACKE_zheevxΔ</td>
<td></td>
</tr>
<tr>
<td>DGGEVΔ</td>
<td>ZGGEVΔ</td>
<td>“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 905</td>
</tr>
<tr>
<td>CGGEVΔ</td>
<td>ZCGGEVΔ</td>
<td></td>
</tr>
<tr>
<td>SGGEVΔ</td>
<td>DGGEVΔ</td>
<td></td>
</tr>
<tr>
<td>CGGEVXΔ</td>
<td>ZCGGEVXΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sggevΔ</td>
<td>LAPACKE_dggevΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgggevΔ</td>
<td>LAPACKE_zggevΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sggevx</td>
<td>LAPACKE_dggevxΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgggevx</td>
<td>LAPACKE_zggevxΔ</td>
<td></td>
</tr>
</tbody>
</table>

26 IBM Engineering and Scientific Subroutine Library for Linux on POWER  Version 6.1: ESSL Guide and Reference
### Table 21: 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>&gt;</td>
<td>DSYGVD^[\Delta]</td>
<td>“SSYGVD, DSYGVD, CHEGVD, ZHEGVD (Eigenvalues, and Optionally, the Eigenvectors of a Positive Definite Real Symmetric or Complex Hermitian Generalized Eigenproblem Using a Divide-and-Conquer Algorithm)” on page 922</td>
</tr>
</tbody>
</table>
| > | ZHEGVD\^[\Delta] | \[
\begin{align*}
&\text{LAPACKE_ssygvd}\^[\Delta] \\
&\text{LAPACKE_zhegvd}\^[\Delta]
\end{align*}
\] |
| > | LAPACKE_ssygvd\^[\Delta] | \[
\begin{align*}
&\text{DYGVD}\^[\Delta] \\
&\text{ZHEGVD}\^[\Delta] \\
&\text{LAPACKE_dsygvd}\^[\Delta] \\
&\text{LAPACKE_zhegvd}\^[\Delta]
\end{align*}
\] |
| > | LAPACKE_zhegvd\^[\Delta] | \[
\begin{align*}
&\text{LAPACKE_ssygvd}\^[\Delta] \\
&\text{LAPACKE_chegvd}\^[\Delta] \\
&\text{LAPACKE_ssygvd}\^[\Delta] \\
&\text{LAPACKE_chegvd}\^[\Delta]
\end{align*}
\] |

\^[\Delta] LAPACK

### 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 22: 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^[\GPU]</td>
<td>DCFTD^[\GPU]</td>
<td>“SCFTD and DCFTD (Multidimensional Complex Fourier Transform)” on page 962</td>
</tr>
</tbody>
</table>
### Table 22: 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>SRCFTD&lt;sup&gt;GPU&lt;/sup&gt;</td>
<td>DRCFTD&lt;sup&gt;GPU&lt;/sup&gt;</td>
<td>“SRCFTD and DRCFTD (Multidimensional Real-to-Complex Fourier Transform)” on page 970</td>
</tr>
<tr>
<td>SCRFTD&lt;sup&gt;GPU&lt;/sup&gt;</td>
<td>DCRFTD&lt;sup&gt;GPU&lt;/sup&gt;</td>
<td>“SCRFTD and DCRFTD (Multidimensional Complex-to-Real Fourier Transform)” on page 978</td>
</tr>
<tr>
<td>SCFT§</td>
<td>DCFT§</td>
<td>“SCFT and DCFT (Complex Fourier Transform)” on page 986</td>
</tr>
<tr>
<td>SRCFT§</td>
<td>DRCFT§</td>
<td>“SRCFT and DRCFT (Real-to-Complex Fourier Transform)” on page 993</td>
</tr>
<tr>
<td>SCRFT§</td>
<td>DCRFT§</td>
<td>“SCRFT and DCRFT (Complex-to-Real Fourier Transform)” on page 1000</td>
</tr>
<tr>
<td>SCOSF&lt;sup&gt;§,ND&lt;/sup&gt;</td>
<td>DCOSF</td>
<td>“SCOSF and DCOSF (Cosine Transform)” on page 1008</td>
</tr>
<tr>
<td>SSINF&lt;sup&gt;§&lt;/sup&gt;</td>
<td>DSINF</td>
<td>“SSINF and DSINF (Sine Transform)” on page 1015</td>
</tr>
<tr>
<td>SCFT2§</td>
<td>DCFT2§</td>
<td>“SCFT2 and DCFT2 (Complex Fourier Transform in Two Dimensions)” on page 1021</td>
</tr>
<tr>
<td>SRCFT2§</td>
<td>DRCFT2§</td>
<td>“SRCFT2 and DRCFT2 (Real-to-Complex Fourier Transform in Two Dimensions)” on page 1028</td>
</tr>
<tr>
<td>SCRFT2§</td>
<td>DCRFT2§</td>
<td>“SCRFT2 and DCRFT2 (Complex-to-Real Fourier Transform in Two Dimensions)” on page 1034</td>
</tr>
<tr>
<td>SCFT3§</td>
<td>DCFT3§</td>
<td>“SCFT3 and DCFT3 (Complex Fourier Transform in Three Dimensions)” on page 1041</td>
</tr>
<tr>
<td>SRCFT3§</td>
<td>DRCFT3§</td>
<td>“SRCFT3 and DRCFT3 (Real-to-Complex Fourier Transform in Three Dimensions)” on page 1047</td>
</tr>
<tr>
<td>SCRFT3§</td>
<td>DCRFT3§</td>
<td>“SCRFT3 and DCRFT3 (Complex-to-Real Fourier Transform in Three Dimensions)” on page 1053</td>
</tr>
</tbody>
</table>

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

<sup>ND</sup> Documentation for this subroutine is no longer provided.

<sup>GPU</sup> This subroutine can offload computations to the GPUs when the ESSL SMP CUDA Library is used.

### 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 23: 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 1060</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 1065</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 1070</td>
</tr>
<tr>
<td>SDCON</td>
<td>DDCON</td>
<td>“SDCON, DDCON, SDCOR, and DDCOR (Convolution or Correlation with Decimated Output Using a Direct Method)” on page 1079</td>
</tr>
<tr>
<td>SACOR§</td>
<td></td>
<td>“SACOR (Autocorrelation of One or More Sequences)” on page 1083</td>
</tr>
<tr>
<td>SACORF</td>
<td></td>
<td>“SACORF (Autocorrelation of One or More Sequences Using the Mixed-Radix Fourier Method)” on page 1086</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 24: 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 1091</td>
</tr>
<tr>
<td>SIZC</td>
<td>DIZC</td>
<td>“SIZC and DIZC (I-th Zero Crossing)” on page 1094</td>
</tr>
<tr>
<td>STREC</td>
<td>DTREC</td>
<td>“STREC and DTREC (Time-Varying Recursive Filter)” on page 1096</td>
</tr>
<tr>
<td>SQINT</td>
<td>DQINT</td>
<td>“SQINT and DQINT (Quadratic Interpolation)” on page 1099</td>
</tr>
<tr>
<td>SWLEV</td>
<td>DWLEV</td>
<td>“SWLEV, DWLEV, CWLEV, and ZWLEV (Wiener-Levinson Filter Coefficients)” on page 1102</td>
</tr>
<tr>
<td>CWLEV</td>
<td>ZWLEV</td>
<td></td>
</tr>
</tbody>
</table>

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 25: 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 1110</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 1111</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 1114</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 1117</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 1120</td>
</tr>
</tbody>
</table>

### Interpolation subroutines

The interpolation subroutines provide the capabilities of doing polynomial interpolation, local polynomial interpolation, and both one- and two-dimensional cubic spline interpolation (Table 26 on page 30).

### Table 26: 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>“SPINT and DPINT (Polynomial Interpolation)” on page 1126</td>
</tr>
<tr>
<td>STPINT</td>
<td>DTPINT</td>
<td>“STPINT and DTPINT (Local Polynomial Interpolation)” on page 1130</td>
</tr>
<tr>
<td>SCSINT</td>
<td>DCSINT</td>
<td>“SCSINT and DCSINT (Cubic Spline Interpolation)” on page 1133</td>
</tr>
<tr>
<td>SCSIN2</td>
<td>DCSIN2</td>
<td>“SCSIN2 and DCSIN2 (Two-Dimensional Cubic Spline Interpolation)” on page 1137</td>
</tr>
</tbody>
</table>

### 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 27: 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 1145</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 1147</td>
</tr>
</tbody>
</table>
Table 27: List of Numerical Quadrature 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>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 1149</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 1155</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 1158</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 1161</td>
</tr>
</tbody>
</table>

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

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
- Tausworthe exclusive-or algorithm

Table 28: 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 1166</td>
</tr>
</tbody>
</table>

Table 29: 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 1170</td>
</tr>
<tr>
<td>SNRNG</td>
<td>DNRNG</td>
<td>“SNRNG and DNRNG (Generate a Vector of Normally Distributed Pseudo-Random numbers)” on page 1173</td>
</tr>
<tr>
<td>SURAND</td>
<td>DURAND</td>
<td>“SURAND and DURAND (Generate a Vector of Uniformly Distributed Random Numbers)” on page 1176</td>
</tr>
<tr>
<td>SNRAND</td>
<td>DNRAND</td>
<td>“SNRAND and DNRAND (Generate a Vector of Normally Distributed Random Numbers)” on page 1178</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 1181</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 subroutines

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

Table 30: 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 1188</td>
</tr>
<tr>
<td>ERRSAV</td>
<td>“ERRSAV (ESSL ERRSAV Subroutine)” on page 1190</td>
</tr>
<tr>
<td>ERRSET</td>
<td>“ERRSET (ESSL ERRSET Subroutine)” on page 1191</td>
</tr>
<tr>
<td>ERRSTR</td>
<td>“ERRSTR (ESSL ERRSTR Subroutine)” on page 1192</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 1193</td>
</tr>
<tr>
<td>SETGPUS</td>
<td>“SETGPUS (Set the Number of GPUs and Identify Which GPUs ESSL Should Use)” on page 1194</td>
</tr>
<tr>
<td>STRIDE</td>
<td>“STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)” on page 1195</td>
</tr>
<tr>
<td>SLAMCH and DLAMCH</td>
<td>“SLAMCH and DLAMCH (Determine machine parameters for floating-point arithmetic)” on page 1210</td>
</tr>
<tr>
<td>DSRSM</td>
<td>“DSRSM (Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode)” on page 1212</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 1215</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 1219</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 for ESSL

Planning your ESSL program involves several tasks.

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 programs,” on page 125 and Chapter 5, “Processing Your Program,” on page 169.

Serial and SMP Libraries Provided by ESSL

ESSL provides the following serial library:

- ESSL Serial Libraries, which support the following environments:
  - 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, 64-bit pointer environment
  - 64-bit integer, 64-bit pointer environment

- ESSL SMP CUDA Library, which supports the following environment:
  - 32-bit integer, 64-bit pointer environment

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 Power CPUs

The GPU enabled subroutines that the ESSL SMP CUDA Library contains are listed in “Using the ESSL SMP CUDA Library” on page 42.

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. 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 or the GPUs; 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 31 on page 34 and Table 32 on page 35.

See Table 33 on page 37 for a list of the ESSL subroutines that automatically use SIMD algorithms when the appropriate alignment restrictions (as described in Table 31 on page 34 and Table 32 on page 35) 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 51), 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>
<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>
### Table 31: VSX Alignment Requirements for SIMD Algorithms in Linear Algebra Subroutines. (continued)

<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></td>
<td></td>
<td></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 1191.

### Table 32: 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 36 and “2” on page 36)</td>
<td>1 (see Note “3” on page 36)</td>
<td>Multiple of 2 (see Note “3” on page 36)</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>Doubleword</td>
<td>1 (see Note “3” on page 36)</td>
<td>Multiple of 4 (see Note “3” on page 36)</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 36)</td>
</tr>
<tr>
<td>Data Type</td>
<td>Vector and Matrix Alignment</td>
<td>Stride Between Elements Within Sequence</td>
<td>Stride Between Sequences</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------</td>
<td>----------------------------------------</td>
<td>-------------------------</td>
</tr>
</tbody>
</table>

**Notes:**

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 _COSF and _SINF, 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 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:
   - 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
   - Error 2211 will be issued and your program will terminate
5. If the alignment restrictions in this table are not met, in some cases 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 1191.
Table 33: ESSL Subroutines that Automatically Use SIMD Algorithms When Alignment Restrictions are Met on VSX-enabled Processors.

### Subroutine Names

**Vector-Scalar Linear Algebra Subprograms (See Note):**

- ISAMAX, IDMAX, ICAMAX, IZAMAX
- ISAMIN, IDMIN
- ISMAX, IDMAX
- ISMIN, IDMIN
- SASUM, DASUM, SCASUM, DZASUM
- SAXPY, DAXPY, CAXPY, ZAXPY
- ISAXPY, DAXPY
- SCOPY, DCOPY, CCOPY, ZCOPY
- SDOT, DDOT, CDOTU, ZDOTU, CDOTC, ZDOTC
- DNRM2, DZRNM2
- DTRNORM2, ZNORM2
- SROTM, DROTM, CROTM, ZROTM
- CSCCAL, DSCCAL, CSCCAL, DZSCAL
- SSWAP, DSWAP, CSWAP, ZSWAP
- SVEA, DVEA, CVEA, ZVEA
- SVES, DVES, CVES, ZVES
- SVEM, DVEM, CVEM
- SYAX, DYAX, CYAX, ZYAX, CSYAX, ZDYAX
- SZAXPY, DZAXPY, CZAXPY, ZZAXPY

**Matrix-Vector Linear Algebra Subprograms (See Note):**

- SGEMV, DGEMV, CGEMV, ZGEMV
- SGEMV, DGEMV, CGEMV, ZGEMV
- SGEMV, DGEMV, CGEMV, ZGEMV
- SPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV
- SPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV
- SPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV
- SPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV
- SPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV
- SPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV
- SPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV
- SPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV
- SPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV

**Matrix Operations (See Note):**

- SGEMUL, DGEMUL, CGEMUL, ZGEMUL
- SGEMM, DGEMM, CGEMM, ZGEMM
- SMM, DMM, CMM, ZMM
- STRMM, DTRMM, CTRMM, ZTRMM
- STRSM, DTRSM, CTRSM, ZTRSM
- SSYRK, DSYRK, CSSYRK, ZSYRK, CHERK, ZHERK
- SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, ZHER2K
- SGEMTI, DGETM, CGEMTI, ZGETM, CGECEM, ZGECM
Table 33: ESSL Subroutines that Automatically Use SIMD Algorithms When Alignment Restrictions are Met on VSX-enabled Processors. (continued)

<table>
<thead>
<tr>
<th>Subroutine Names</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fourier Transforms:</td>
<td></td>
</tr>
<tr>
<td>SCFTD, DCFTD</td>
<td></td>
</tr>
<tr>
<td>SRCFTD, DRCFTD</td>
<td></td>
</tr>
<tr>
<td>SCRFTD, DCRFTD</td>
<td></td>
</tr>
<tr>
<td>SCFT, DCFT</td>
<td></td>
</tr>
<tr>
<td>SRCFT, DRCFT</td>
<td></td>
</tr>
<tr>
<td>SCOSF, DCOSF</td>
<td></td>
</tr>
<tr>
<td>SSINF, DSINF</td>
<td></td>
</tr>
<tr>
<td>SCFT2, DCFT2</td>
<td></td>
</tr>
<tr>
<td>SRCFT2, DRCFT2</td>
<td></td>
</tr>
<tr>
<td>SCRFT2, DCRFT2</td>
<td></td>
</tr>
<tr>
<td>SCFT3, DCFT3</td>
<td></td>
</tr>
<tr>
<td>SRCFT3, DRCFT3</td>
<td></td>
</tr>
<tr>
<td>SCRFT3, DCRFT3</td>
<td></td>
</tr>
<tr>
<td>Convolutions and Correlations:</td>
<td></td>
</tr>
<tr>
<td>SCONF, SCORF, SACORF</td>
<td></td>
</tr>
<tr>
<td>Random Number Generation:</td>
<td></td>
</tr>
<tr>
<td>SURNG, DURNG</td>
<td></td>
</tr>
<tr>
<td>SNRNG, DNRNG</td>
<td></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.

**Multithreaded Subroutines Provided by ESSL**

Table 34 on page 39 lists the multithreaded subroutines provided by ESSL and also indicates which of those subroutines use SIMD algorithms.
Table 34: Multithreaded Subroutines.

<table>
<thead>
<tr>
<th>Subroutine Category</th>
<th>Multithreaded Subroutine</th>
<th>Does this subroutine also use SIMD algorithms on VSX-enabled processors?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector-Scalar Linear Algebra Subprograms⁴</td>
<td>SASUM, DASUM, SCASUM, DZASUM</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SAXPY, DAXPY, CAXPY, ZAXPY</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>&gt;ISAXPBY, DAXPBYI&lt;</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SCOPY, DCOPY, CCOPY, ZCOPY</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SDOT, DDOT, CDOTU, ZDOTU, CDOTC</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SNDOT, DNDOT</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SNORM2, DNORM2, CNORM2, ZNORM2</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SROT, DROT, CROT, ZROT, CSROT, ZDROT</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SSCAL, DSCAL, CSCAL, ZSCALE, CSSCAL, ZDSCAL</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SSWAP, DSWAP, CSWAP, ZSWAP</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SVEA, DVEA, CVEA, ZVEA</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SVES, DVES, CVEA, ZVES</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SVEM, DVEM, CVEM, ZVEM</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SYAX, DYAX, CYAX, ZYAX, CSYAX, ZDYAX</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SZAXPY, DZAXPY, CZAXPY, ZZAXPY</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>
</tr>
<tr>
<td>---------------------------------------------------------</td>
<td>----------------------------------------------------------------</td>
<td>---------------------------------------------------------------------</td>
</tr>
<tr>
<td>Matrix-Vector Linear Algebra Subprograms(^1)</td>
<td>SGEMV, DGEMV, CGEMV, ZGEMV</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SGER, DGER, CGERU, ZGERU, CGERC, ZGERC</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SSPMV, DSPMV, CHPMV, ZHPMV</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SSYMV, DSYMV, CHEMV, ZHEMV</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SSPR, DSPR, CHPR, ZHPR</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SSYR, DSYR, CHER, ZHER</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SSPR2, DSPR2, CHPR2, ZHPR2</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SSYR2, DSYR2, CHER2, ZHER2</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SGBMV(^2), DGBMV(^2)</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>CGBMV(^2), ZGBMV(^2)</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SSBMV(^2), DSBMV(^2)</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>CHBMV(^2), ZHBMV(^2)</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>STRMV, DTRMV, CTRMV, ZTRMV</td>
<td>Yes except DTRMV and ZTRMV</td>
</tr>
<tr>
<td></td>
<td>STPMV, DTPMV, CTPMV, ZTPMV</td>
<td>Yes except DTPMV and ZTPMV</td>
</tr>
<tr>
<td></td>
<td>STRSV, DTRSV, CTRSV, ZTRSV</td>
<td>Yes except DTRSV and ZTRSV</td>
</tr>
<tr>
<td></td>
<td>STPSV, DTPSV, CTPSV, ZTPSV</td>
<td>Yes except DTPSV, ZTPSV</td>
</tr>
<tr>
<td></td>
<td>STBMV(^2), DTBMV(^2)</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>CTBMV(^2), ZTBVM(^2)</td>
<td>No</td>
</tr>
<tr>
<td>Sparse Matrix Vector Linear Algebra Subprograms</td>
<td>DSMMX, DSDMX</td>
<td>No</td>
</tr>
</tbody>
</table>

\(^1\) SIMD algorithms on VSX-enabled processors

See “SIMD Algorithms on VSX-Enabled Processors” on page 34.
<table>
<thead>
<tr>
<th>Subroutine Category</th>
<th>Multithreaded Subroutine</th>
<th>Does this subroutine also use SIMD algorithms on VSX-enabled processors?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Operations²</td>
<td>SGEADD, DGEADD, CGEADD, ZGEADD</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SGESUB, DGESUB, CGESUB, ZGESUB</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SGEMUL, DGEMUL, CGEMUL, ZGEMUL</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SGEMM, DGEMM, CGEMM, ZGEMM</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SSYMM, DSYMM, CSYMM, ZSYMM, CHEMM, ZHEMM</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>STRMM, DTRMM, CTRMM, ZTRMM</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>STRSM, DTRSM, CTRSM, ZTRSM</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SSYRK, DSYRK, CSYRK, ZSYRK, CHERK, ZHERK</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, ZHER2K</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SGETMI, DGETMI, CGETMI, ZGETMI, CGECMI, ZGECMI</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SGETMO, DGETMO, CGETMO, ZGETMO, CGECMO, ZGECMO</td>
<td>Yes</td>
</tr>
<tr>
<td>Dense Linear Algebraic Equations</td>
<td>SGESV, DGESV, CGESV, ZGESV</td>
<td>See Note “1” on page 42</td>
</tr>
<tr>
<td></td>
<td>SGEF, DGEF, CGEF, ZGEF</td>
<td>See Note “1” on page 42</td>
</tr>
<tr>
<td></td>
<td>SGES, DGES, CGES, ZGES</td>
<td>See Note “1” on page 42</td>
</tr>
<tr>
<td></td>
<td>SGETRF, DGETRF, CGETRF, ZGETRF</td>
<td>See Note “1” on page 42</td>
</tr>
<tr>
<td></td>
<td>SGETRS, DGETRS, CGETRS, ZGETRS</td>
<td>See Note “1” on page 42</td>
</tr>
<tr>
<td></td>
<td>SPPSV, DPPSV, CPPSV, ZPPSV</td>
<td>See Note “1” on page 42</td>
</tr>
<tr>
<td></td>
<td>SPPF, DPPF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, ZPPTRF</td>
<td>See Note “1” on page 42</td>
</tr>
<tr>
<td></td>
<td>SPPTRS, DPPTRS, CPPTRS, ZPPTRS</td>
<td>See Note “1” on page 42</td>
</tr>
<tr>
<td></td>
<td>SPOSV, DPOSV, CPOSV, ZPOSV</td>
<td>See Note “1” on page 42</td>
</tr>
<tr>
<td></td>
<td>SPOSM, DPOSM, CPOSM, ZPOSM</td>
<td>See Note “1” on page 42</td>
</tr>
<tr>
<td></td>
<td>SPPFC³, DPPFC³, DPOFC³</td>
<td>See Note “1” on page 42</td>
</tr>
<tr>
<td></td>
<td>SPPTRI, DPPTRI, CPPTRI, ZPPTRI, SPPIC³, DPPIC³, DPOIC³</td>
<td>See Note “1” on page 42</td>
</tr>
<tr>
<td></td>
<td>STRI, DTRI, STRTRI, DTRTRI, CTRTRI, ZTRTRI</td>
<td>See Note “1” on page 42</td>
</tr>
</tbody>
</table>
Table 34: 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>
</tr>
</thead>
<tbody>
<tr>
<td>Banded Linear Algebraic Equations</td>
<td>SGBSV, DGBSV, CGBSV, ZGBSV</td>
<td>See Note “1” on page 42</td>
</tr>
<tr>
<td></td>
<td>SGBTRS, DGBTRS, CGBTRS, ZGBTRS</td>
<td>See Note “1” on page 42</td>
</tr>
<tr>
<td></td>
<td>SPBSV, DPBSV, CPBSV, ZPBSV</td>
<td>See Note “1” on page 42</td>
</tr>
<tr>
<td></td>
<td>SPBTRS, DPBTRS, CPBTRS, ZPBTRS</td>
<td>See Note “1” on page 42</td>
</tr>
<tr>
<td>Sparse Linear Algebraic Equations</td>
<td>DSRIS^4</td>
<td>No</td>
</tr>
<tr>
<td>Linear Least Squares</td>
<td>SGEQRF, DGEQRF, CGEQRF, ZGEQRF</td>
<td>See Note “1” on page 42</td>
</tr>
<tr>
<td>Fourier Transforms</td>
<td>SCFTD, SRCFTD, SCRFTD, SCFT, SRCFT, SCFT2, SRCFT2, SCRFT2, SCRFT3</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>DCFTD, DRCFTD, DCRFTD, DCFT, DRCFT, DCFT2, DRCFT2, DCRFT2, DCRFT3</td>
<td>Yes</td>
</tr>
<tr>
<td>Convolution and Correlation</td>
<td>SCOND, SCORD</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SDCON, SDCOR, DDCON, DDCOR</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>SCONF, SCORF SACORF</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. The Level 2 Banded BLAS use multiple threads only when the bandwidth is sufficiently large.
3. Multiple threads are used for the factor or inverse computation.
4. DSRIS only uses multiple threads when IPARM(4) = 1 or 2.

Using the ESSL SMP CUDA Library

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 Power CPUs and NVIDIA GPUs

The ESSL SMP CUDA Library Linear Algebra subroutines leverage 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 Fourier Transform subroutines leverage NVIDIA CUDA Fast Fourier Transform library (cuFFT) and use GPUs only when the GPU memory is large enough to accommodate the computation (data and working space) of a single transform.

The ESSL SMP CUDA Library contains GPU-enabled versions of the following subroutines:
Table 35: List of ESSL SMP CUDA subroutines that are GPU-enabled.

<table>
<thead>
<tr>
<th>Type of subroutine</th>
<th>Subroutine name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Operations</td>
<td>• SGEMM, DGEMM, CGEMM, and ZGEMM</td>
</tr>
<tr>
<td></td>
<td>• SSYMM, DSYMM, CSYMM, ZSYM, CHEMM, and ZHEMM</td>
</tr>
<tr>
<td></td>
<td>• STRMM, DTRMM, CTRMM, and ZTRMM</td>
</tr>
<tr>
<td></td>
<td>• STRSM, DTRSM, CTRSM, and ZTRSM</td>
</tr>
<tr>
<td></td>
<td>• SSYRK, DSYRK, CSYRK, ZSYRK, CHERK, and ZHERK</td>
</tr>
<tr>
<td></td>
<td>• SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, and ZHER2K</td>
</tr>
<tr>
<td>Dense Linear Algebraic Equations</td>
<td>• SGESV, DGESV, CGESV, and ZGESV</td>
</tr>
<tr>
<td></td>
<td>• SGETRF, DGETRF, CGETRF, and ZGETRF</td>
</tr>
<tr>
<td></td>
<td>• SGETRS, DGETRS, CGETRS, and ZGETRS</td>
</tr>
<tr>
<td></td>
<td>• SPPSV, DPPSV, CPPSV, and ZPPSV</td>
</tr>
<tr>
<td></td>
<td>• SPOTRF, DPOTRF, CPOTRF, and ZPOTRF</td>
</tr>
<tr>
<td></td>
<td>• SPOTRS, DPOTRS, CPOTRS, and ZPOTRS</td>
</tr>
<tr>
<td>Linear Least Squares</td>
<td>• SGEQRF, DGEQRF, CGEQR, and ZGEQRF</td>
</tr>
<tr>
<td></td>
<td>• SGELS, DGELS, CGELS, and ZGELS</td>
</tr>
<tr>
<td>Fourier Transforms</td>
<td>• SCFTD and DCFTD</td>
</tr>
<tr>
<td></td>
<td>• SRCFTD and DRCFTD</td>
</tr>
<tr>
<td></td>
<td>• SCRFDT and DCRFTD</td>
</tr>
</tbody>
</table>

**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 -lesslsmpcuda (see Chapter 5, “Processing Your Program,” on page 169). 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 44.

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:

**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 44 for information on the CUDA_VISIBLE_DEVICES environment variable.

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 44 for information on the ESSL_CUDA_HYBRID environment variable.

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” on page 44 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:

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

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

```
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 1194.

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.
- For the Fourier transform subroutines, when the transform length or data layout are not supported by the NVIDIA cuFFT library or when the problem size is too big to fit in the GPU memory.

**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).

---

1 NVIDIA CUDA Multi Process Service (MPS) is a feature that allows multiple CUDA processes to share a single GPU context.
The default hybrid mode for the Linear Algebra subroutines (ESSL_CUDA_HYBRID=yes) means that the ESSL SMP CUDA Library subroutines can run on both Power CPUs and NVIDIA GPUs.

Subroutines SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, and ZHER2K only use the Power CPUs for scaling operations.

For the Fourier Transform subroutines, ESSL_CUDA_HYBRID=yes means that the subroutines can run on either CPUs or NVIDIA GPUs (not both), depending on performance. ESSL_CUDA_HYBRID=no means that the subroutines must run on GPUs if the transform length and data layout are supported by NVIDIA cuFFT.

**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).

If you want ESSL to pin your host memory buffers on entry to gpu-enabled subroutines and unpinned 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 SETGPUS subroutine or the environmental variable CUDA_VISIBLE_DEVICES with the local rank of the MPI tasks to ensure that 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, it is recommended that you run by using the NVIDIA MPS. The NVIDIA MPS is a runtime service that is designed to let multiple MPI processes by using CUDA run concurrently on a single GPU in a way that is transparent to the MPI program. NVIDIA MPS supports at most 48 MPI tasks per V100 GPU and 16 MPI tasks per P100 GPUs, but if you are using ESSL, it is recommended that you use Core Affinity and no more tasks than the number of cores that are being used.

If you are sharing GPUs, it is possible that ESSL cannot allocate workspace on the GPU. To reduce the amount of context local storage per MPI task if you are using V100 GPUs, set the CUDA_MPS_ACTIVE_THREADS_PERCENTAGE environment variable to 200/n, where n is the number of MPI tasks per GPU. If possible, reduce the number of MPI tasks per node or increase the number of GPUs that are being used per node to potentially eliminate the allocation failures.
If error cudaStreamCreate failed with CUDA message: all CUDA-capable devices are busy or unavailable occurs when you are 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 1194) or the environmental variable CUDA_VISIBLE_DEVICES with the local rank of the MPI tasks to inform ESSL which GPUs your MPI Tasks can use.

For best performance, consider increasing the block size that 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 60. For a description of these data types, see “How Do You Set Up Your Scalar Data?” on page 47.
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” on page 47.

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 programs,” on page 125.

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 data structures,” on page 71. 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 126. 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 146.
- For C++, see “Setting Up Arrays in C++” on page 159.

**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 34.

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 chose a storage mode that conserves storage and potentially improves performance. For definitions of the various data structures and their corresponding storage modes, see Chapter 3, “Setting up data structures,” on page 71. 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 DSMX.
- 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 data structures,” on page 71. 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 125
- “C Programs” on page 142
- “C++ Programs” on page 155
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 145 and “Using Logical Data in C” on page 145.
  - Short-precision complex and logical data in C++ programs, using the guidelines in “Using Logical Data in C++” on page 159.

- Character values must be one of the specified values. For example, it may have to be ‘N’, ‘T’, or ‘C’.

- Numeric values must fall within the correct range for that argument. For example, a numeric value may need to be greater than or equal to 0, or it may have to be a nonzero value.

- Arrays must be defined correctly; that is, they must have the correct dimensions, or the dimensions must fall within the correct range. For example, input and output matrices may need to be conformable, or the number of rows in the matrix must be less than or equal to the leading dimension specified. (ESSL assumes all arrays are stored in column-major order.)

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

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

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

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

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

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

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

For more examples of specifying vectors and matrices, see Chapter 3, “Setting up data structures,” on page 71.

Using Auxiliary Storage in ESSL

For the ESSL subroutines listed in Table 36 on page 50, 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>
<thead>
<tr>
<th>Table 36: ESSL Subroutines Requiring Auxiliary Working Storage.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Subroutine Names</strong></td>
</tr>
<tr>
<td>Linear Algebra Subprograms:</td>
</tr>
<tr>
<td>DSMTM</td>
</tr>
<tr>
<td>Matrix Operations:</td>
</tr>
<tr>
<td>_GEMMS</td>
</tr>
<tr>
<td>Dense Linear Algebraic Equations:</td>
</tr>
<tr>
<td>_GEFCD _PPFCD _GEICD _PPICD _POFCD</td>
</tr>
<tr>
<td>_POICD DGEFP^A DPPFP^A</td>
</tr>
<tr>
<td>Sparse Linear Algebraic Equations:</td>
</tr>
<tr>
<td>DGSF DGSS DGKFS DGKFS^A DSKFS DSKFS^A DSRIS DSMCG DSDCG DSMCG DSDCG</td>
</tr>
<tr>
<td>Linear Least Squares:</td>
</tr>
<tr>
<td>_GESVF _GELLS</td>
</tr>
<tr>
<td>Fourier Transforms:</td>
</tr>
<tr>
<td>_CFTD _RCFTD _CRFTD _CFT _RCFT _CRFT _COSF _SINF</td>
</tr>
<tr>
<td>SCOSF^A _CFT2 _RCFT2 _CRFT2 _CFT3</td>
</tr>
<tr>
<td>_RCFT3 _CRFT3 SCFTP^A SCFT2^A SCFT3^A</td>
</tr>
<tr>
<td>Convolutions and Correlations:</td>
</tr>
<tr>
<td>SCONF SCORF SACORF</td>
</tr>
<tr>
<td>Related Computations:</td>
</tr>
<tr>
<td>_WLEV</td>
</tr>
<tr>
<td>Interpolation:</td>
</tr>
<tr>
<td>_TPINT _CSIN2</td>
</tr>
<tr>
<td>Random Number Generation:</td>
</tr>
<tr>
<td>_NRAND</td>
</tr>
<tr>
<td>Utilities:</td>
</tr>
<tr>
<td>DGKTRN DSKTRN</td>
</tr>
</tbody>
</table>
Dynamic Allocation of Auxiliary Storage

Dynamic allocation for the auxiliary storage is performed when error 2015 is unrecoverable and \( n_{aux} = 0 \). 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 aux work area in the \( n_{aux} \) 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 sufficient values for your current and future needs. Use them if ease of migration to future machines and future releases of ESSL is your primary concern. For details, see “How Do You Calculate the Size of Auxiliary Storage Using the Formulas?” on page 51.
- Use the ESSL error-handling facilities to return to you a minimum value for the particular processor you are currently running on. (Values vary by platform.) Use this approach if conserving storage is your primary concern. For details, see “How Do You Get ESSL to Calculate the Size of Auxiliary Storage Using ESSL Error Handling?” on page 51.

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

The formulas provided for calculating \( n_{aux} \) indicate a sufficient amount of auxiliary storage required, which, in most cases, is larger than the minimum amount, returned by ESSL error handling. There are two types of formulas:

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

- Processor-independent formulas
  These are given 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 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 \( n, m, n_1 \), or \( n_2 \). 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:

- By leaving error 2015 unrecoverable, you can obtain the minimum required value of \( n_{aux} \) from the input-argument error message, but your program terminates.
- By making error 2015 recoverable, you can obtain the minimum required value of \( n_{aux} \) from the input-argument error message and have the updated \( n_{aux} \) 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 $n_{aux}$ 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 $n_{aux}$ 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. 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 53 illustrates what happens when error 2015 is unrecoverable.
* This check applies only to several NAUX1 arguments in the Fourier transform, convolution, and correlation subroutines.

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

Having ESSL Calculate Auxiliary Storage Size with Recoverable Error 2015

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

Use EINFO and ERRSET with an ESSL error exit routine, ENOTRM, to make error 2015 recoverable. This allows you to dynamically determine in your program the minimum sizes required for the auxiliary working storage areas, specified in the naux arguments. Run your program with the naux values smaller than required by the subroutine for the particular processor you are running on. As a general guideline, specify values smaller than those listed in the formulas. However, if a lower limit is specified in the syntax (only for several naux1 arguments in the Fourier transform, convolution, and correlation subroutines), you should not go below that limit. The ESSL error monitor returns the necessary sizes of the aux storage areas in the input-argument error message and a return code is passed back to your program, indicating that updated values are also returned in the naux arguments. You can then react to these updated values during run time in your program. ESSL does not perform any computation when this error occurs. For details on how to do this, see Chapter 4, “Coding programs,” on page 125. (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 on page 54 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 naux, 20 and 300, are used only for the purposes of this example and do not relate to any of the ESSL subroutines.)

NAUX

Meaning of the NAUX Value

20

Lower limit of naux required for using recoverable input-argument error-handling facilities in ESSL. (This applies only to several naux1 arguments in the Fourier transform, convolution, and correlation subroutines. You can find the lower limit in the syntax description for the naux1 argument. For a list of subroutines, see “Using Auxiliary Storage in ESSL” on page 49.)
Minimum value of \( \text{naux} \), required for successful running (on the processor the program is being run on).

Table 37 on page 55 describes the actions taken by ESSL in every possible situation for the values given in this example.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{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 \text{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 ( \text{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>( \text{naux} \geq 300 )</td>
<td>Your application program runs successfully.</td>
<td>Your application program runs successfully.</td>
</tr>
</tbody>
</table>

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 136
- For C, see “Input-Argument Errors in C Example” on page 151
- For C++, see “Input-Argument Errors in C++ Example” on page 164

You may want to provide a separate subroutine to calculate the auxiliary storage size whenever you need it. Figure 3 on page 56 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 \( \text{naux} \) arguments. Upon return, your program checks the return code. If it is nonzero, the \( \text{naux} \) 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 \( \text{naux} \) 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 38 on page 56, 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 38: ESSL Subroutines Requiring Transform Lengths.

<table>
<thead>
<tr>
<th>Subroutine Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fourier Transforms:</td>
</tr>
<tr>
<td>_CFT _RCFT _CRFT _COSF _SINF</td>
</tr>
<tr>
<td>SCOSFT _CFT2 _RCFT2 _CRFT2 _CFT3</td>
</tr>
<tr>
<td>_RCFT3 _CRFT3 SCFTP SCFT2P SCFT3P</td>
</tr>
</tbody>
</table>

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 956 to determine an acceptable length. For details, see “How Do You Calculate the Transform Length Using the Table or Formula?” on page 57.
- 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 57.
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 956. 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 57 illustrates what happens when error 2030 is unrecoverable.

![Diagram](image)

**Figure 4: How to Obtain an \( N \) Value from an Error Message, but Terminate**
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 programs,” on page 125. (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 58 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>

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

---

IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and Reference
Table 39 on page 59 describes the actions taken by ESSL in every possible situation for the values given in this example.

<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>$n = 7208960$ or $n = 7340032$</td>
<td>Your application program runs successfully.</td>
<td>Your application program runs successfully.</td>
</tr>
<tr>
<td>$7208960 &lt; n &lt; 7340032$</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 136.
- For C, see “Input-Argument Errors in C Example” on page 151.
- For C++, see “Input-Argument Errors in C++ Example” on page 164.

You may want to provide a separate subroutine to calculate the transform length whenever you need it. Figure 6 on page 60 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.
You might want to combine the request for auxiliary storage sizes along with your request for transform lengths. Figure 7 on page 60 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 55.

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 47. 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 programs,” on page 125.

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 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, 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 \( aA \), where \( a \) is a scalar and \( A \) is a matrix, if \( a \) 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 programs to ESSL,” on page 179.
• 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 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 42.

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 73.

• 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.

• 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 VSX enabled processors, see “SIMD Algorithms on VSX-Enabled Processors” on page 34. 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 1195. For additional information, see “Setting Up Your Data” on page 959.

• When you use the ESSL SMP CUDA library subroutines SCFTD, DCFTD, SRCFTD, DRCFTD, SCRF TD, and DCRFTD, stride values from the STRIDE subroutine might negatively impact performance or might also prevent the computations that are running on the GPUs. <

• 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 42.

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 4.

• Migration considerations concerning performance are described in Chapter 6, “Migrating programs to ESSL,” on page 179.

• 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 [“40” on page 1255], [“51” on page 1256], [“52” on page 1256].
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, “Troubleshooting ESSL,” on page 183.

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 187.

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 49.

• 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 56.

• 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 613 and “DBSSV (Symmetric Indefinite Matrix Factorization and Multiple Right-Hand Side Solve)” on page 607.

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...
occurs, the input-argument error message is issued with the updated values. In addition, the updated values are returned to your program in the input arguments named in the error message, along with a nonzero return code and processing continues. Return code values associated with these recoverable errors are described under “Error Conditions” for each ESSL subroutine in Part 2.

For details on how to code the necessary statements in your program to make 2015, 2030 and 2200 recoverable, see the following:

- “Input-Argument Errors in Fortran” on page 134
- “Input-Argument Errors in C” on page 147
- “Input-Argument Errors in C++” on page 161

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 191.

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 65 shows what happens when a computational error occurs.

![Figure 8: How to Obtain Computational Error Information from an Error Message, but Terminate](image)

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 1188. 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 1188. 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 136
- “Computational Errors in C” on page 151
- “Computational Errors in C++” on page 165

Figure 9 on page 66 shows what happens if you make a computational error recoverable.

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

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

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

**All Resource Errors**

ESSL returns a resource error and terminates your program when an attempt to allocate work area fails. Some ESSL subroutines attempt to allocate work area for their internal use. Other ESSL subroutines
attempt to dynamically allocate auxiliary storage when a user requests it through calling sequence arguments, such as aux and naux. For information on how you could reduce memory constraints on the system or increase the amount of memory available before rerunning the application program, see “ESSL Resource Error Messages” on page 185.

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 196.

**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 185.

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 196.

**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 40 on page 67 shows the default values established in the table when ESSL is installed.

<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>Range of Error Messages (From–To)</td>
<td>Number of Allowable Errors (ALLOW)</td>
<td>Number of Messages Printed (PRINT)</td>
<td>Modifiable Table Entry (MODENT)</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>------------------------------------</td>
<td>-----------------------------------</td>
<td>---------------------------------</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>
<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 subroutines,” on page 1187.

**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 64. For how to use ERRSET to do this, see the information specific to your programming language in Chapter 4, “Coding programs,” on page 125.
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 64. For how to use ERRSET to do this, see the information specific to your programming language in Chapter 4, “Coding programs,” on page 125.

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 65. For how to use ERRSET to do this, see the information specific to your programming language in Chapter 4, “Coding programs,” on page 125.

To Allow Parts of Your Application to Have Unique Error-Handling Environments

If your program is part of a large application, you may want to dynamically save and restore entries in the error option table that have been altered by ERRSET. This ensures the integrity of the error option table when it is used by multiple programs within an application. For a more detailed discussion, see “How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?” on page 69. For how to use ERRSAV and ERRSTR, see the information specific to your programming language in Chapter 4, “Coding programs,” on page 125.

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 programs,” on page 125.

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 67. 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 141.

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 programs,” on page 125.

• All ESSL error messages are listed under “Messages” on page 186.

• 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, “Troubleshooting ESSL,” on page 183.
Chapter 3. Setting up data structures

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

Vector 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 $x_1, y_1, x_2, y_2, x_3, y_3, \ldots$ 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:

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

A row vector appears symbolically as follows:

$$
\mathbf{x} = \begin{bmatrix}
  x_1 & x_2 & x_3 & \ldots & x_n
\end{bmatrix}
$$
Vectors can contain either real or complex numbers. When they contain real numbers, they are sometimes called real vectors. When they contain complex numbers, they are called complex vectors.

**Transpose of a Vector**

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

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

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, 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:

\[
\mathbf{x} = \begin{bmatrix}
    x_1 \\
    x_2 \\
    x_3 \\
    . \\
    . \\
    . \\
    x_n
\end{bmatrix} \quad \mathbf{x}^H = \begin{bmatrix}
    \overline{x}_1 & \overline{x}_2 & \overline{x}_3 & \ldots & \overline{x}_n
\end{bmatrix}
\]

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

\[ X = \text{SASUM} (4, A, 2) \]
where the number of elements to be summed in the vector is 4, the location of the vector is A, and the stride is 2.

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

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

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

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

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

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

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

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

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

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

For a complete description of how vectors are stored within arrays, see “How Stride Is Used for Vectors” on page 73.

For a complex vector, a special storage arrangement is used to accommodate the two parts, \( a \) and \( b \), of each complex number \( \mathbf{a} + \mathbf{b} \) 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 47 for a description of real and complex numbers, and “How Do You Set Up Your Arrays?” on page 47 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 \( \mathbf{x} \), 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:

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

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 126. 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 \((x_1)\) is by using the following arguments:

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

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

\[
X(BEGIN + (-n+1)inc)
\]

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

\[
X = (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0)
\]

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

\[
X(2+(-4+1)(-2)) = X(8)
\]

where element \(X(8)\) is 8.0.

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

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

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, $INDX$, also of length 5, contains:

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

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

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

then the array $INDX$ contains:

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

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

For each $x_j \neq 0$, for $j = 1,n$

there exists $i$, where $1 \leq i \leq nz$,

such that $X(i) = x_j$ and $INDX(i) = j$.

where:

- $x_1, ..., x_n$ 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.
- $INDX$ is the array containing the $nz$ indices indicating the element positions.

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

---

76 IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and Reference
Matrices

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

\[
A = \begin{bmatrix}
a_{11} & \cdots & a_{1n} \\
\vdots & \ddots & \vdots \\
a_{m1} & \cdots & a_{mn}
\end{bmatrix}
\]

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:

\[
A = \begin{bmatrix}
a_{11} & \cdots & a_{1n} \\
\vdots & \ddots & \vdots \\
a_{m1} & \cdots & a_{mn}
\end{bmatrix} \quad A^T = \begin{bmatrix}
a_{11} & \cdots & a_{m1} \\
\vdots & \ddots & \vdots \\
a_{1n} & \cdots & a_{mn}
\end{bmatrix}
\]

ESSSL 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 \textit{transa} and \textit{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:

\[
A = \begin{bmatrix}
a_{11} & \cdots & a_{1n} \\
\vdots & \ddots & \vdots \\
a_{m1} & \cdots & a_{mn}
\end{bmatrix} \quad A^H = \begin{bmatrix}
\bar{a}_{11} & \cdots & \bar{a}_{m1} \\
\vdots & \ddots & \vdots \\
\bar{a}_{1n} & \cdots & \bar{a}_{mn}
\end{bmatrix}
\]
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” on page 78 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 47 for a description of real and complex numbers, and “How Do You Set Up Your Arrays?” on page 47 for a description of how real and complex data is stored in arrays.

How Leading Dimension Is Used for Matrices

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

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

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

\[
(E2-E1+1)
\]

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

```
| 1.0  8.0 15.0 22.0 29.0 |
| 2.0  9.0 16.0 23.0 30.0 |
| 3.0 10.0 17.0 24.0 31.0 |
| 4.0 11.0 18.0 25.0 32.0 |
| 5.0 12.0 19.0 26.0 33.0 |
| 6.0 13.0 20.0 27.0 34.0 |
| 7.0 14.0 21.0 28.0 35.0 |
```
then processing begins at the element at row 3 and column 0 in array A, which is 3.0.

The leading dimension is used to find the starting point for the matrix elements in each of the \( n \) successive columns in the array. ESSL subroutines assume that the arrays are stored in column-major order, as described under “How Do You Set Up Your Arrays?” on page 47, 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:

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

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:

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

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:

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

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:

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

Symmetric Matrix Storage Representation

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

- “Lower-Packed Storage Mode” on page 80
- “Upper-Packed Storage Mode” on page 82
- “Lower Storage Mode” on page 83
- “Upper Storage Mode” on page 83

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:

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

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

$$
AP(1) = a_{11} \text{ (start the first column)} \\
AP(2) = a_{21} \\
AP(3) = a_{31} \\
\vdots \\
\vdots \\
$$
Following is an example of a symmetric matrix that uses the element values to show the order in which the matrix elements are stored in the array.

Given the following matrix \( A \):

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

the array is:

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

**Note:** Additional work storage is required in the array for some ESSL subroutines; for example, in the simultaneous linear algebraic equation subroutines SPPF, DPPF, SPPS, and DPPS.

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

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

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

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

\[
\begin{align*}
AP(1) &= a_{11} \text{ (start the first column)} \\
AP(2) &= a_{12} \text{ (start the second column)} \\
AP(3) &= a_{22} \\
AP(4) &= a_{13} \text{ (start the third column)} \\
AP(5) &= a_{23} \\
AP(6) &= a_{33} \\
AP(7) &= a_{14} \text{ (start the fourth column)} \\
&\quad \cdot \\
&\quad \cdot \\
&\quad \cdot \\
&\quad \cdot \\
AP(j(j-1)/2+1) &= a_{1j} \text{ (start the } j\text{-th column)} \\
AP(j(j-1)/2+2) &= a_{2j} \\
AP(j(j-1)/2+3) &= a_{3j} \\
&\quad \cdot \\
&\quad \cdot \\
&\quad \cdot \\
&\quad \cdot \\
AP(j(j-1)/2+j) &= a_{jj} \text{ (end of the } j\text{-th column)} \\
&\quad \cdot \\
&\quad \cdot \\
&\quad \cdot \\
&\quad \cdot \\
&\quad \cdot
\end{align*}
\]
\[ AP(n(n+1)/2) \]
\[ = a_{nn} \]

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

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

the array is:

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

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

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

**Lower Storage Mode**

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

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

Given the following matrix \( A \):

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

the array is:

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

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

**Upper Storage Mode**

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

Following is an example of a symmetric matrix \( A \) of order 5 and how it is stored in an array \( AU \).
Given the following matrix $A$:

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

the array is:

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

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 80.

**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 80.

**Complex Hermitian Matrix**

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

$H = H^H$

**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 80. (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 82. (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 83.
- Upper storage mode, as described in “Upper Storage Mode” on page 83.

Following is an example of a complex Hermitian matrix $H$ of order 5.
Given the following matrix $H$:

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

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

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 $x^H Ax$ is positive for all nonzero vectors $x$. A complex Hermitian matrix $A$ is negative definite if and only if $x^H Ax$ 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 84.

**Indefinite Complex Hermitian Matrix**

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

Setting up data structures 85
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 84.

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 84.

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

$$A = \begin{bmatrix}
    a_{11} & a_{21} & a_{31} & \cdots & a_{n1} \\
    a_{21} & a_{11} & a_{21} & \cdots & a_{n2} \\
    a_{31} & a_{21} & a_{11} & \cdots & a_{n3} \\
    \vdots & \vdots & \vdots & \ddots & \vdots \\
    a_{n1} & \cdots & a_{31} & a_{21} & a_{11}
\end{bmatrix}$$

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 $x_i = a_{i1}$ 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 $A$:

<table>
<thead>
<tr>
<th>99 12 13 14 15 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 99 13 14 15 16</td>
</tr>
<tr>
<td>13 12 99 12 13 14</td>
</tr>
<tr>
<td>14 13 12 99 12 13</td>
</tr>
<tr>
<td>15 14 13 12 99 12</td>
</tr>
</tbody>
</table>
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:

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

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 or negative definite complex Hermitian matrix, see “Positive Definite or Negative Definite Complex Hermitian Matrix” on page 85.

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

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

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 $A$:

\[
\begin{align*}
| & (10.0, 0.0) & (2.0, -3.0) & (3.0, 1.0) & (1.0, 1.0) \\
| & (2.0, 3.0) & (10.0, 0.0) & (2.0, -3.0) & (-3.0, 1.0) \\
| & (-3.0, -1.0) & (2.0, 3.0) & (10.0, 0.0) & (2.0, -3.0) \\
| & (1.0, -1.0) & (-3.0, -1.0) & (2.0, 3.0) & (10.0, 0.0) \\
\end{align*}
\]

the array is:

\[
X = ((10.0, 0.0), (2.0, -3.0), (-3.0, 1.0), (1.0, 1.0))
\]

**Triangular Matrix**

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

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

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

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

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

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

\[
U = \begin{bmatrix}
    u_{11} & u_{12} & u_{13} & \ldots & u_{1n} \\
    0 & u_{22} & u_{23} & \ldots & \cdot \\
    0 & 0 & u_{33} & \ldots & \cdot \\
    \cdot & \cdot & \cdot & \ldots & \cdot \\
    0 & \ldots & 0 & u_{nn}
\end{bmatrix} \\
L = \begin{bmatrix}
    l_{11} & 0 & 0 & \ldots & 0 \\
    l_{21} & l_{22} & 0 & \ldots & \cdot \\
    l_{31} & l_{32} & l_{33} & \ldots & \cdot \\
    \cdot & \cdot & \cdot & \ldots & \cdot \\
    l_{n1} & \ldots & 0 & \ldots & l_{nn}
\end{bmatrix}
\]

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:
Triangular Matrix Storage Representation

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

• “Upper-Triangular-Packed Storage Mode” on page 89
• “Lower-Triangular-Packed Storage Mode” on page 89
• “Upper-Triangular Storage Mode” on page 90
• “Lower-Triangular Storage Mode” on page 90

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 82.

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

Given the following matrix \( U \):

\[
\begin{bmatrix}
1 & u_{12} & u_{13} & \cdots & u_{1n} \\
0 & 1 & u_{23} & \cdots & . \\
0 & 0 & 1 & \cdots & . \\
. & . & . & \cdots & . \\
0 & . & . & . & 0 & 1
\end{bmatrix}
\]

the array is:

\[
\text{UP} = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
\]

Lower-Triangular-Packed Storage Mode

When a lower-triangular matrix is stored in lower-triangular-packed storage mode, the lower triangle of the matrix is stored, including the diagonal, in a one-dimensional array. The lower triangle is packed by columns. The elements are packed sequentially, column by column, in \( n(n+1)/2 \) elements of a one-dimensional array. To calculate the location of each element of the triangular matrix in the array, use the technique described in “Lower-Packed Storage Mode” on page 80.

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

\[
\begin{bmatrix}
1 & 0 & 0 & \cdots & 0 \\
0 & 1 & 0 & \cdots & . \\
0 & 0 & 1 & \cdots & . \\
. & . & . & \cdots & . \\
0 & . & . & . & 1
\end{bmatrix}
\]

The matrix \( L \) is:

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

the array is:

\[
\text{LP} = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
\]
Given the following matrix $L$:

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

the array is:

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

**Upper-Triangular Storage Mode**

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

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

Given the following matrix $U$:

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

the array is:

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

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

**Lower-Triangular Storage Mode**

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

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

Given the following matrix $L$:

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

the array is:

\[
LTA = \begin{bmatrix}
11 & * & * & * & * \\
21 & 22 & * & * & * \\
31 & 32 & 33 & * & * \\
41 & 42 & 43 & 44 & * \\
51 & 52 & 53 & 54 & 55
\end{bmatrix}
\]
where "*" means you do not have to store a value in that position in the array.

**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 \( m \geq n \):

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

\[
L = \begin{bmatrix}
    l_{11} & 0 & 0 & \ldots & 0 \\
    l_{21} & l_{22} & 0 & \ldots & 0 \\
    l_{31} & l_{32} & l_{33} & \ldots & 0 \\
    \vdots & \vdots & \vdots & \ddots & \vdots \\
    l_{m1} & l_{m2} & l_{m3} & \ldots & l_{mn}
\end{bmatrix}
\]

If \( m < n \):

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

\[
L = \begin{bmatrix}
    l_{11} & 0 & 0 & \ldots & 0 \\
    l_{21} & l_{22} & 0 & \ldots & 0 \\
    l_{31} & l_{32} & l_{33} & \ldots & 0 \\
    \vdots & \vdots & \vdots & \ddots & \vdots \\
    l_{m1} & l_{m2} & l_{m3} & \ldots & l_{mn}
\end{bmatrix}
\]

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 \( m \geq n \):

\[
U = \begin{bmatrix}
1 & u_{12} & u_{13} & \cdots & u_{1n} \\
0 & 1 & u_{23} & \cdots & \cdot \\
0 & 0 & 1 & \cdots & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot \\
0 & \cdots & 0 & 1 & \cdot \\
0 & \cdots & 0 & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot \\
0 & \cdots & 0 & \cdot & \cdot \\
0 & \cdots & 0 & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot \\
\end{bmatrix}
\]

If \( m < n \):

\[
U = \begin{bmatrix}
1 & u_{12} & u_{13} & \cdots & u_{1,n-1} & u_{1n} \\
0 & 1 & u_{23} & \cdots & u_{2,n-1} & u_{2n} \\
0 & 0 & 1 & \cdots & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot & \cdot \\
0 & \cdots & 0 & 1 & \cdot & \cdot \\
0 & \cdots & 0 & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot & \cdot \\
\end{bmatrix}
\]

\[
L = \begin{bmatrix}
1 & 0 & 0 & \cdots & 0 \\
0 & \cdot & \cdot & \cdots & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot \\
\cdot & \cdot & \cdot & \cdots & \cdot \\
\end{bmatrix}
\]

### Trapezoidal Matrix Storage Representation

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

- “Upper-Trapezoidal Storage Mode” on page 92
- “Lower-Trapezoidal Storage Mode” on page 93

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 \( U \):

\[
U = \begin{bmatrix}
11 & 12 & 13 & 14 & 15 \\
16 & 17 & 18 & 19 & 20 \\
21 & 22 & 23 & 24 & 25 \\
26 & 27 & 28 & 29 & 30 \\
31 & 32 & 33 & 34 & 35 \\
\end{bmatrix}
\]
the array is:

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

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 \(U\):

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

the array is:

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

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 \(L\):

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

the array is:

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

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

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 $mu = p-1$:

\[
\begin{bmatrix}
\begin{array}{cccccc}
 a_{11} & a_{12} & a_{13} & \cdot & \cdot & a_{1p} \\
 a_{21} & a_{22} & a_{23} & \cdot & \cdot & \cdot \\
 a_{31} & a_{32} & a_{33} & \cdot & \cdot & \cdot \\
 \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
 a_{q1} & & & & & \cdot \\
 \cdot & 0 & & & & \cdot \\
 \cdot & 0 & & & & \cdot \\
 \cdot & 0 & & & & \cdot \\
 0 & \cdot & 0 & \cdot & \cdot & \cdot \\
 0 & \cdot & 0 & \cdot & \cdot & \cdot \\
 \end{array}
\end{bmatrix}
\]

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 $a_{11}, a_{22}, a_{33}, \ldots, a_{nn}$. Following is a general band matrix with $m > n$:
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 $AGB(ml+mu+1, 1)$:

$$
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 3 & 4 & 5 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 4 & 5 & 6 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 4 & 5 & 6 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 5 & 6 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 6 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 7 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 9 & 0 & 0 & 0 \\
\end{array}
$$

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 $A$:

$$
\begin{array}{cccccccccc}
11 & 12 & 13 & 14 & 0 & 0 & 0 & 0 & 0 & 0 \\
21 & 22 & 23 & 24 & 25 & 0 & 0 & 0 & 0 & 0 \\
31 & 32 & 33 & 34 & 35 & 36 & 0 & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0 & 42 & 43 & 44 & 45 & 46 & 47 & 0 & 0 & 0 \\
0 & 0 & 53 & 54 & 55 & 56 & 57 & 58 & 0 & 0 \\
0 & 0 & 0 & 64 & 65 & 66 & 67 & 68 & 69 & 0 \\
0 & 0 & 0 & 0 & 75 & 76 & 77 & 78 & 79 & 0 \\
0 & 0 & 0 & 0 & 0 & 86 & 87 & 88 & 89 & 0 \\
\end{array}
$$
you store it in general-band storage mode in a 23 by 9 array \( \text{AGB} \) as follows, where \( a_{11} \) is stored in \( \text{AGB}(6,1) \):

\[
\begin{array}{cccccccccc}
* & * & * & * & * & * & * & * & * & * \\
* & * & * & 14 & 25 & 36 & 47 & 58 & 69 & * \\
* & * & 12 & 34 & 45 & 56 & 67 & 78 & 89 & * \\
11 & 22 & 33 & 44 & 55 & 66 & 77 & 88 & 99 & * \\
21 & 32 & 43 & 54 & 65 & 76 & 87 & 98 & * & * \\
31 & 42 & 53 & 64 & 75 & 86 & 97 & * & * & * \\
* & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * \\
* & * & * & * & * & * & * & * & * & * \\
\end{array}
\]

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

\[
\begin{align*}
\text{MD} & = \text{ML} + \text{MU} + 1 \\
\text{DO} & 1 \ J=1,N \\
& \quad \text{DO} 1 \ I=\max(J-\text{MU},1),\min(J+\text{ML},N) \\
& \quad \text{AGB}(I-J+\text{MD},J) = A(I,J) \\
& 1 \ \text{CONTINUE}
\end{align*}
\]

**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 \( \text{AGB} \) 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 \( \text{AGB} \), which is declared as \( \text{AGB}(ml+mu+1,n) \), the columns of elements in matrix \( A \) are stored in each column in array \( \text{AGB} \) as follows, where \( a_{11} \) is stored at \( \text{AGB}(mu+1,1) \):
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 \( AGB(4,1) \):

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

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 \( A \):

\[
\begin{bmatrix}
| 11 & 12 & 13 & 14 & 0 & 0 & 0 & 0 & 0 | \\
| 21 & 22 & 23 & 24 & 25 & 0 & 0 & 0 & 0 | \\
| 31 & 32 & 33 & 34 & 35 & 36 & 0 & 0 & 0 | \\
| 0 & 42 & 43 & 44 & 45 & 46 & 47 & 0 & 0 | \\
| 0 & 0 & 53 & 54 & 55 & 56 & 57 & 58 & 0 | \\
| 0 & 0 & 0 & 64 & 65 & 66 & 67 & 68 & 69 | \\
| 0 & 0 & 0 & 0 & 75 & 76 & 77 & 78 & 79 |
\end{bmatrix}
\]
you store it in array AGB, declared as $AGB(6,9)$, as follows, where $a_{11}$ is stored in $AGB(4,1)$ and the leading diagonal does not fill up the whole row:

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

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:

```fortran
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 $k = q-1$:

$$
A = \begin{bmatrix}
  a_{11} & a_{21} & a_{31} & \cdots & a_{q1} & 0 & \cdots & 0 \\
  a_{21} & a_{22} & a_{32} & & 0 & \cdots & \cdots & \cdots \\
  a_{31} & a_{32} & a_{33} & & & 0 & \cdots & \cdots \\
    & & & \cdots & & & \cdots & \cdots \\
  a_{q1} & & & & & & & \cdots \\
  0 & & & & & & & \cdots \\
  & 0 & & & & & & \cdots \\
  & & 0 & & & & & \cdots \\
  & & & 0 & & & & \cdots \\
  & & & & 0 & & & \cdots \\
  & & & & & 0 & & \cdots \\
  & & & & & & 0 & \cdots \\
  & & & & & & & a_{nn}
\end{bmatrix}
$$

### 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 100
- “Lower-Band-Packed Storage Mode” on page 100

Setting up data structures 99
**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:

$$
ASB = \begin{bmatrix}
* & \cdots & \cdots & \star & a_{1p} & a_{2,p+1} & \cdots & a_{n-k,n} \\
\vdots & \ddots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\
\vdots & \ddots & \ddots & \vdots & \ddots & \ddots & \ddots & \vdots \\
\vdots & \ddots & \ddots & \vdots & \ddots & \ddots & \ddots & \vdots \\
\star & a_{13} & a_{24} & \cdots & \cdots & \cdots & \cdots & \cdots \\
\star & a_{12} & a_{23} & \cdots & \cdots & \cdots & \cdots & \cdots \\
a_{11} & a_{22} & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\end{bmatrix}
$$

where "\star" 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:

```fortran
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:

$$
ASB = \begin{bmatrix}
  a_{11} & a_{21} & a_{22} & \cdots & \cdots & a_{n,n} \\
  a_{21} & a_{31} & a_{32} & \cdots & * & \\
  a_{31} & a_{41} & a_{42} & \cdots & \cdots & \\
    & \cdots & \cdots & \cdots & & \\
  a_{q1} & a_{q+1,2} & \cdots & a_{n,n-k} & * & \cdots & *
\end{bmatrix}
$$

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

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

Given the following matrix $A$:

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

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

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

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

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

**Positive Definite Symmetric Band Matrix**

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

**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 99.

**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 100
- Upper-band-packed storage mode, as described in “Upper-Band-Packed Storage Mode” on page 100

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 $H$:

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

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

- In lower-band-packed storage mode:

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

- In upper-band-packed storage mode:

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

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

Positive Definite Complex Hermitian Band Matrix

A complex Hermitian band matrix $A$ is positive definite if and only if $x^H A x$ 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 101.

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:

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

Its band elements are arranged uniformly near the diagonal in the upper triangle of the matrix, such that:
\[ u_{ij} = 0 \text{ if } j-i > k \]

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

\[
U = \begin{bmatrix}
  u_{11} & u_{21} & u_{31} & \ldots & u_{q1} & 0 & \ldots & 0 \\
  0 & u_{22} & u_{32} & \ldots & 0 & \ldots & \ldots & \ldots \\
  0 & 0 & u_{33} & \ldots & 0 & \ldots & \ldots & \ldots \\
  \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \ddots & \ddots \\
  0 & \ldots & 0 & \ldots & 0 & u_{n-1,n} & \ldots & u_{nn}
\end{bmatrix}
\]

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:

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

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

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

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

\[
L = \begin{bmatrix}
  l_{11} & 0 & \ldots & \ldots & 0 \\
  l_{21} & l_{22} & 0 & \ldots & \ldots \\
  l_{31} & l_{32} & l_{33} & \ldots & \ldots \\
  \vdots & \vdots & \vdots & \ddots & \ddots \\
  0 & \ldots & 0 & \ldots & 0 \\
  0 & \ldots & 0 & \ldots & \ldots \\
  0 & \ldots & 0 & \ldots & l_{nn}
\end{bmatrix}
\]

A triangular band matrix can also be a unit triangular band matrix if all the diagonal elements have a value of 1. For an illustration of a unit triangular matrix, see “Triangular Matrix” on page 88.
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” on page 104
- “Lower-Triangular-Band-Packed Storage Mode” on page 105

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:

\[
UTB = \begin{bmatrix}
* & * & * & U_{1p} & U_{2,p+1} & \ldots & U_{n-k,n} \\
* & * & * & * & U_{13} & U_{24} & \ldots \\
* & * & * & * & * & U_{12} & U_{23} \\
U_{11} & * & * & * & * & * & U_{nn}
\end{bmatrix}
\]

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=MAX(1,J-K),J
        UTB(M+I,J)=U(I,J)
    10 CONTINUE
20 CONTINUE
```

**Lower-Triangular-Band-Packed Storage Mode**

Only the band elements of the lower triangular part of a lower triangular band matrix, including the main diagonal, are stored for lower-triangular-band-packed storage mode.

**Note:** As an alternative to this storage mode, you can specify your arguments in your subroutine in a special way so that ESSL selects the matrix elements properly, and you can leave your matrix stored in full-matrix storage mode.

For 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:

\[
L_{TB} = \begin{bmatrix}
    l_{11} & l_{22} & \ldots & l_{nn} \\
    l_{21} & l_{32} & \ast & \\
    l_{31} & l_{42} & \ast & \\
    \vdots & \vdots & \ddots & \\
    l_{q1} & l_{q+1,2} & \ldots & l_{n,n-k} & \ast & \ast & \ast
\end{bmatrix}
\]

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.

**Setting up data structures**

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:

\[
LTB = \begin{bmatrix}
    11 & 22 & 33 & 44 & 55 & 66 \\
    21 & 32 & 43 & 54 & 65 & \ast \\
    31 & 42 & 53 & 64 & \ast & \ast
\end{bmatrix}
\]
Following is an example of how to transform your lower triangular band matrix to lower-triangular-band-packed storage mode:

```
DO 20 J=1,N
   M=1-J
   DO 10 I=J,MIN(N,J+K)
      LTB(M+I,J)=L(I,J)
   10 CONTINUE
20 CONTINUE
```

**General Tridiagonal Matrix**

A general tridiagonal matrix is a matrix whose nonzero elements are found only on the diagonal, subdiagonal, and superdiagonal of the matrix; that is:

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

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

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

**General Tridiagonal Matrix Storage Representation**

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

- “LAPACK-General Tridiagonal Storage Mode” on page 106
- “General Tridiagonal Storage Mode” on page 107

**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:

\[
\begin{bmatrix}
    11 & 12 & 0 & 0 & 0 \\
    21 & 22 & 23 & 0 & 0 \\
    0 & 32 & 33 & 34 & 0 \\
    0 & 0 & 43 & 44 & 45
\end{bmatrix}
\]
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:

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

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:

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

where "\( \star \)" 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:

| 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 |

which you store in tridiagonal storage mode in arrays \( C \), \( D \), and \( E \), each of length 5, as follows:

| \( C = (\star, 21, 32, 43, 54) \) |
| \( D = (11, 22, 33, 44, 55) \) |
| \( E = (12, 23, 34, 45, \star) \) |

**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 \( n \):

| 10 12 0 0 0 |
| 21 22 23 0 0 |
| 0 32 33 34 0 |
| 0 0 43 44 45 |
| 0 0 0 54 55 |
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” on page 108
- “Symmetric-Tridiagonal Storage Mode” on page 108

**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 $D$ of 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:

$$\begin{bmatrix}
    10 & 1 & 0 & 0 & 0 \\
    1 & 20 & 2 & 0 & 0 \\
    0 & 2 & 30 & 3 & 0 \\
    0 & 0 & 3 & 40 & 4 \\
    0 & 0 & 0 & 4 & 50
\end{bmatrix}$$

which you store in 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:

$$C = (\ast, a_{21}, a_{32}, a_{43}, ..., a_{n,n-1})$$

where "\ast" 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:

\[
\begin{pmatrix}
10 & 1 & 0 & 0 & 0 \\
1 & 20 & 2 & 0 & 0 \\
0 & 2 & 30 & 3 & 0 \\
0 & 0 & 3 & 40 & 4 \\
0 & 0 & 0 & 4 & 50
\end{pmatrix}
\]

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 \( x^T Ax \) 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 107.

**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:

\[
\begin{pmatrix}
(10, \ 0) & (1, \ 1) & (1, \ 2) & (1, \ 3) & (1, \ 4) \\
(1, \ -1) & (20, \ 0) & (2, \ 1) & (2, \ 2) & (2, \ 3) \\
(1, \ -2) & (20, \ 0) & (3, \ 1) & (3, \ 2) & (3, \ 3) \\
(1, \ -3) & (20, \ 0) & (4, \ 1) & (4, \ 2) & (4, \ 3) \\
(1, \ -4) & (20, \ 0) & (5, \ 1) & (5, \ 2) & (5, \ 3)
\end{pmatrix}
\]

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” on page 109.

**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 $A$:

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

**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” on page 110
- “Compressed-Diagonal Storage Mode” on page 111
- “Storage-by-Indices” on page 114
- “Storage-by-Columns” on page 114
- “Storage-by-Rows” on page 115
- “Diagonal-Out Skyline Storage Mode” on page 117
- “Profile-In Skyline Storage Mode” on page 119

**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 [“89” on page 1258]. 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 $A$:

```
| 11  0  13  0  0  0 |
| 21 22  0 24  0  0 |
|  0 32  33  0 35  0 |
|  0  0  43 44  0 46 |
| 51  0  0 54  55  0 |
| 61 62  0  0 65  66 |
```

the arrays are:

```
AC = | 11 13  0  0 |
     | 22 21 24  0 |
     | 33 32 35  0 |
     | 44 43 46  0 |
     | 55 51 54  0 |
     | 66 61 62 65 |

KA = | 1 3  *  * |
     | 2 1  4  * |
     | 3 2  5  * |
     | 4 3  6  * |
     | 5 1  4  * |
     | 6 1  2  5 |
```

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

**Compressed-Diagonal Storage Mode**

The storage mode used for square sparse matrices stored in compressed-diagonal storage mode has two variations, depending on whether the matrix is a general sparse matrix or a symmetric sparse matrix. This 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 $1$ to $n-1$. Then the diagonals can be referred to as $d_k$, where $k = 1$ to $n-1$.

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

$$
A = \begin{bmatrix}
  a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\
  a_{21} & a_{22} & a_{23} & \cdots & \cdot \\
  a_{31} & a_{32} & a_{33} & \cdots & \cdot \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  a_{n1} & \cdot & \cdot & \cdots & a_{nn}
\end{bmatrix}
$$

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 $A$:

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

the arrays are:

$$
AD = \begin{bmatrix}
  11 & 13 & 0 & 0 & 0 \\
  22 & 24 & 21 & 0 & 0 \\
  33 & 35 & 32 & 0 & 0 \\
  44 & 46 & 43 & 0 & 0 \\
\end{bmatrix}
$$

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:

\[
\begin{bmatrix}
11 & 13 & 0 \\
22 & 24 & 0 \\
33 & 35 & 0 \\
44 & 46 & 0 \\
55 & 0 & 51 \\
66 & 0 & 62 \\
\end{bmatrix}
\]

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

For each \( d_k \neq (0, ..., 0) \), for \( k = 1-n \), \( n-1 \) for general square sparse matrices, or for each unique \( d_k \neq (0, ..., 0) \), for \( k = 1-n \), \( n-1 \) for symmetric sparse matrices,

there exists \( l \), where \( 1 \leq l \leq nd \), such that \( LA(l) = k \) and column \( l \) in array \( AD \) contains \( dp_k \).

where:

- Array \( AD \) is defined as \( AD(lda, nd) \), where \( lda \geq n \), 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, \( d_k \), where \( k = i-j \).
- \( dp_k \) are the diagonals, \( d_k \), 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: \( dp_k = (a_{1,k+1}, a_{2,k+2}, ..., a_{n-k,n}, 0_1, ..., 0_k) \)
  - For the main diagonal \( (k = 0) \), \( dp_0 \) has no padding: \( dp_0 = (a_{11}, a_{22}, ..., a_{nn}) \)
  - For subdiagonals \( (k < 0) \), \( dp_k \) has \( |k| \) leading zeros: \( dp_k = (0_1, ..., 0_{|k|}, a_{|k|+1,1}, a_{|k|+2,2}, ..., a_{n,n-|k|}) \)
**Storage-by-Indices**

For a sparse matrix \( A \), 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:

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

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:

\[
\begin{align*}
AR(k) &= a_{ij} \\
IA(k) &= i \\
JA(k) &= j
\end{align*}
\]

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

\begin{align*}
AR(k) &= a_{ij} \\
IA(k) &= i
\end{align*}

And for $j = 1, n$,

\begin{align*}
JA(j) &= k, \text{ where } a_{ij} \text{ in } AR(k), \text{ is the first element stored in } AR \text{ for column } j \\
JA(j) &= JA(j+1), \text{ where all } a_{ij} = 0 \text{ in column } j \\
JA(n+1) &= ne+1
\end{align*}

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 $AR$.
- **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 $JA$. 

Setting up data structures 115
Given the following matrix \( \mathbf{A} \):

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

the arrays are:

\[
\begin{align*}
\mathbf{AR} &= (11, 13, 24, 22, 21, 32, 33, 44, 43, 46, 61, 62, 66) \\
\mathbf{IA} &= (1, 3, 6, 8, 11, 11, 14) \\
\mathbf{JA} &= (1, 3, 2, 1, 2, 3, 4, 3, 6, 1, 2, 6)
\end{align*}
\]

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 \( \mathbf{AR} \), \( \mathbf{IA} \), and \( \mathbf{JA} \).

Consider the following as an example of a symmetric sparse matrix \( \mathbf{A} \) of order 6 and how it can be stored in arrays \( \mathbf{AR} \), \( \mathbf{IA} \), and \( \mathbf{JA} \) using upper-storage-by-rows, which stores only the upper triangle and diagonal elements.

Given the following matrix \( \mathbf{A} \):

\[
\begin{bmatrix}
11 & 0 & 13 & 0 & 0 & 0 \\
0 & 22 & 23 & 24 & 0 & 0 \\
13 & 23 & 33 & 0 & 35 & 0 \\
0 & 24 & 0 & 44 & 0 & 46 \\
0 & 0 & 35 & 0 & 55 & 0 \\
0 & 0 & 0 & 46 & 0 & 0
\end{bmatrix}
\]

the arrays are:

\[
\begin{align*}
\mathbf{AR} &= (11, 13, 22, 23, 24, 33, 35, 46, 44, 55) \\
\mathbf{IA} &= (1, 3, 6, 8, 10, 11, 11) \\
\mathbf{JA} &= (1, 3, 2, 3, 1, 2, 4, 5, 3, 4)
\end{align*}
\]

Using the same symmetric matrix \( \mathbf{A} \), consider the following as an example of how it can be stored in arrays \( \mathbf{AR} \), \( \mathbf{IA} \), and \( \mathbf{JA} \) using lower-storage-by-rows, which stores only the lower triangle and diagonal elements:

\[
\begin{align*}
\mathbf{AR} &= (11, 22, 23, 33, 13, 24, 44, 55, 35, 46) \\
\mathbf{IA} &= (1, 2, 3, 6, 8, 10, 11) \\
\mathbf{JA} &= (1, 2, 2, 3, 1, 2, 4, 5, 3, 4)
\end{align*}
\]

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

\[
\begin{align*}
\mathbf{AR}(k) &= a_{ij} \\
\mathbf{JA}(k) &= j
\end{align*}
\]
And for $i = 1, m$, 
\[ \text{IA}(i) = k, \text{where } a_{ij} \text{ in AR}(k), \text{is the first element stored in AR for row } i \]
\[ \text{IA}(i) = \text{IA}(i+1), \text{where all } a_{ij} = 0 \text{ in row } i \]
\[ \text{IA}(m+1) = ne+1 \]

where:

- $a_{ij}$ are the elements of sparse matrix $A$, which is either an $m$ by $n$ general sparse matrix or a symmetric sparse matrix of order $m$ containing $ne$ nonzero elements.
- Arrays $AR$ and $JA$ each have $ne$ elements.
- Array $IA$ has $m+1$ elements.

**Diagonal-Out Skyline Storage Mode**

The diagonal-out skyline storage mode used for sparse matrices has two variations, depending on whether the matrix is a general sparse matrix or a symmetric sparse matrix. Both of these variations are explained here.

For a general sparse matrix $A$, diagonal-out skyline storage mode uses four one-dimensional arrays to define the sparse matrix storage, $AU$, $IDU$, $AL$, and $IDL$. Given the sparse matrix $A$ of order $n$, containing $nu+nl-n$ elements under the top and left profiles, the arrays are set up as follows:

- $AU$ of (at least) length $nu$ contains the upper triangle of the sparse matrix $A$, where the columns are stored consecutively from 1 to $n$ in $AU$ in the following way. For each column, the elements starting at the diagonal element and ending at the topmost nonzero element in the column are stored contiguously in $AU$. The elements stored may include zero elements along with the nonzero elements. If all elements in the column to be stored are zero, the diagonal element, $a_{ii}$, having a value of zero, is stored in $AU$ for that column. A total of $nu$ elements are stored for the upper triangle of $A$.
- $IDU$, an integer array of (at least) length $n+1$ contains the relative position of each diagonal element of matrix $A$ in array $AU$; that is, each element $IDU(i)$ of the diagonal pointer array indicates where diagonal element $a_{ii}$ is stored in array $AU$. One-origin is used, so the first element of $IDU$ is always 1. The last element, $IDU(n+1)$, indicates the position after the last element in array $AU$, which is $nu+1$.
- $AL$ of (at least) length $nl$ contains the lower triangle of the sparse matrix $A$, where the rows are stored consecutively from 1 to $n$ in $AL$ in the following way. For each row, the elements starting at the diagonal element and ending at the leftmost nonzero element in the row are stored contiguously in $AL$. The elements stored may include zero elements along with the nonzero elements. If all elements in the row to be stored are zero, the diagonal element, $a_{ii}$, having a value of zero, is stored in $AL$ for that row. A total of $nl$ elements are stored for the lower triangle of $A$. The values of the diagonal elements are meaningless, so you can store any values in those positions in $AL$.
- $IDL$, an integer array of (at least) length $n+1$ contains the relative position of each diagonal element of matrix $A$ in array $AL$; that is, each element $IDL(i)$ of the diagonal pointer array indicates where diagonal element $a_{ii}$ is stored in array $AL$. One-origin is used, so the first element of $IDL$ is always 1. The last element, $IDL(n+1)$, indicates the position after the last element in array $AL$, which is $nl+1$.

Consider the following as an example of a 6 by 6 general sparse matrix $A$ and how it is stored in arrays $AU$, $IDU$, $AL$, and $IDL$.

Given the following matrix $A$:

| 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 |
the arrays are:

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

\[ IDU = (1, 2, 4, 7, 10, 12, 16) \text{ where } nu=15 \]

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

\[ IDL = (1, 2, 4, 7, 11, 13, 17) \text{ where } nl=16 \]

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 \( n \), diagonal-out skyline storage mode uses the same storage technique as for the upper triangle and diagonal elements of the general sparse matrix; therefore, only the \( AU \) and \( IDU \) arrays are needed.

Consider the following as an example of a symmetric sparse matrix \( A \) of order 6 and how it is stored in arrays \( AU \) and \( IDU \).

Given the following matrix \( A \):

\[
\begin{bmatrix}
0 & 12 & 13 & 0 & 0 & 0 \\
12 & 22 & 0 & 24 & 0 & 0 \\
13 & 0 & 33 & 34 & 0 & 36 \\
0 & 24 & 34 & 44 & 45 & 0 \\
0 & 0 & 0 & 45 & 55 & 56 \\
0 & 0 & 36 & 0 & 56 & 66
\end{bmatrix}
\]

the arrays are:

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

\[ IDU = (1, 2, 4, 7, 10, 12, 16) \text{ where } nu=15 \]

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

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 \( A \):

```
|  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 |
```

the arrays are:

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

- **\( IDU \)** = \( (1, 3, 6, 9, 11, 15, 16) \) where \( nu=15 \)

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

- **\( IDL \)** = \( (1, 3, 6, 10, 12, 16, 17) \) where \( nl=16 \)

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

For a symmetric sparse matrix of order \( n \), profile-in skyline storage mode uses the same storage technique as for the upper triangle and diagonal elements of the general sparse matrix; therefore, only the \( AU \) and \( IDU \) arrays are needed.
Consider the following as an example of a symmetric sparse matrix $\mathbf{A}$ of order 6 and how it is stored in arrays $\mathbf{AU}$ and $\mathbf{IDU}$.

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

\[
\begin{array}{cccccc}
0 & 12 & 13 & 0 & 0 & 0 \\
12 & 22 & 0 & 24 & 0 & 0 \\
13 & 0 & 33 & 34 & 0 & 36 \\
0 & 24 & 34 & 44 & 45 & 0 \\
0 & 0 & 0 & 45 & 55 & 56 \\
0 & 0 & 36 & 0 & 56 & 66 \\
\end{array}
\]

the arrays are:

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

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

where $\mathbf{nu}=15$

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

For general sparse matrices and symmetric sparse matrices:

For each $a_{ij}$ for $j = 1, n$ and $i = k, j$,

where $a_{kj}$ is the topmost $a_{ij} \neq 0$ in each column $j$,

there exists $m$, where $1 \leq m \leq \mathbf{nu}$, such that

\[
\mathbf{AU}(m-j+i) = a_{ij}
\]

\[
\mathbf{IDU}(j) = m\text{ for each } a_{ij}
\]

\[
\mathbf{IDU}(n+1) = \mathbf{nu}+1
\]

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 $1 \leq m \leq \mathbf{nl}$, such that

\[
\mathbf{AL}(m-i+j) = a_{ij}
\]

\[
\mathbf{IDL}(i) = m\text{ for each } a_{ij}
\]

\[
\mathbf{IDL}(n+1) = \mathbf{nl}+1
\]

where:

$a_{ij}$ are the elements of sparse matrix $\mathbf{A}$, of order $n$.

Array $\mathbf{AU}$ has $\mathbf{nu}$ elements.

Array $\mathbf{AL}$ has $\mathbf{nl}$ elements.

Arrays $\mathbf{IDU}$ and $\mathbf{IDL}$ 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 47 for a description of real and complex numbers, and “How Do You Set Up Your Arrays?” on page 47 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 73.

Two-Dimensional Sequences
A two-dimensional sequence appears symbolically as a series of columns of elements. (They are represented in the same way as a matrix without the square brackets.) The two subscripts indicate the element positions in the first and second dimensions, respectively:

\[
\begin{array}{cccc}
    a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\
    a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\
    \vdots & \vdots & \ddots & \vdots \\
    a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\
\end{array}
\]

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 78. (In the area of Fourier transforms, a two-dimensional sequence may be stored in transposed form in an array. In this case, the stride for the second dimension is 1, and the stride for the first dimension is the leading dimension of the array.)

Three-Dimensional Sequences
A three-dimensional sequence is represented as a series of blocks of elements. Each block is equivalent to a two-dimensional sequence. The number of blocks indicates the length of the third dimension. The three subscripts indicate the element positions in the first, second, and third dimensions, respectively:
**Plane 0:**

\[
\begin{array}{cccc}
  a_{0,0,0} & \cdots & a_{0,n-1,0} \\
  a_{1,0,0} & \cdots & a_{1,n-1,0} \\
  \vdots & & \vdots \\
  \vdots & & \vdots \\
  a_{m-1,0,0} & \cdots & a_{m-1,n-1,0} \\
\end{array}
\]

**Plane 1:**

\[
\begin{array}{cccc}
  a_{0,0,1} & \cdots & a_{0,n-1,1} \\
  a_{1,0,1} & \cdots & a_{1,n-1,1} \\
  \vdots & & \vdots \\
  \vdots & & \vdots \\
  a_{m-1,0,1} & \cdots & a_{m-1,n-1,1} \\
  \vdots & & \vdots \\
\end{array}
\]

**Plane \((p-1)\):**

\[
\begin{array}{cccc}
  a_{0,0,p-1} & \cdots & a_{0,n-1,p-1} \\
  a_{1,0,p-1} & \cdots & a_{1,n-1,p-1} \\
  \vdots & & \vdots \\
  \vdots & & \vdots \\
  a_{m-1,0,p-1} & \cdots & a_{m-1,n-1,p-1} \\
\end{array}
\]
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,32,2056,16,20,1,1.0,aux,30000)```

See “How Stride Is Used for Three-Dimensional Sequences” on page 123 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 78, 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, $inc_1$, 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, $inc_2$, of a three-dimensional sequence is equivalent to the leading dimension for a matrix.

The stride for the third dimension, $inc_3$, 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, $inc_3$, 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, $inc_3 \geq (inc_2)(n)$.

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:

$inc_1 = 1$
$inc_2 = (E2-E1+1)$
$inc_3 = (E2-E1+1)(F2-F1+1)$

Given an array $A$, declared as $A(1:7,1:3,0:3)$, where the lengths of the first, second, and third dimensions are 7, 3, and 4, respectively, the resulting strides are $inc_1 = 1$, $inc_2 = 7$, and $inc_3 = 21$.

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>
</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, \( \text{inc3} \), is added to the starting point \( p-1 \) times. In this example, the stride for the third dimension is 21, and the number of blocks of elements, \( p \), to be processed is 3, so the starting points in array A are \( A(2,2,1) \), \( A(2,2,2) \), and \( A(2,2,3) \). These are elements 30.0, 51.0, and 72.0. These array elements then correspond to the sequence elements \( a_{111} \), \( a_{112} \), and \( a_{113} \), respectively.

In general terms, this results in the following starting positions for the blocks of elements in the array:

\[
\begin{align*}
A(\text{BEGINI}, \text{BEGINJ}, \text{BEGINK})
\\ A(\text{BEGINI}, \text{BEGINJ}, \text{BEGINK}+1)
\\ A(\text{BEGINI}, \text{BEGINJ}, \text{BEGINK}+2)
\\ \vdots
\\ A(\text{BEGINI}, \text{BEGINJ}, \text{BEGINK}+p-1)
\end{align*}
\]

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:

\[
\begin{align*}
\text{Plane 0:} & \\
& a_{000} \ a_{010} \\
& a_{100} \ a_{110} \\
& a_{200} \ a_{210} \\
& a_{300} \ a_{310}
\\
\text{Plane 1:} & \\
& a_{001} \ a_{011} \\
& a_{101} \ a_{111} \\
& a_{201} \ a_{211} \\
& a_{301} \ a_{311}
\\
\text{Plane 2:} & \\
& a_{002} \ a_{012} \\
& a_{102} \ a_{112} \\
& a_{202} \ a_{212} \\
& a_{302} \ a_{312}
\end{align*}
\]

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 126 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 \( \text{inc1} \), \( \text{inc2} \), and \( \text{inc3} \) and declaring your arrays to have certain number of dimensions, you achieve optimal performance in these subroutines. For details, see “Setting Up Your Data” on page 959 for each subroutine.
Chapter 4. Coding programs

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:

```
CALL subroutine-name (argument-1, . . . , argument-n)
```

An example of a calling sequence for the SAXPY subroutine might be:

```
CALL SAXPY (5,A,X,J+INC,Y,1)
```

The remaining ESSL subroutines are invoked as functions by coding a function reference. You first declare the type of value returned by the function: short- or long-precision real, short- or long-precision complex, or integer. Then you code the function reference as part of an expression in a statement. An example of declaring and invoking the DASUM function might be:

```
DOUBLE PRECISION DASUM,SUM,X

SUM = DASUM (N,X,INCX)
```

Values are returned differently for ESSL subroutines and functions. For subroutines, the results of the computation are returned in an argument specified in the calling sequence. In the CALL statement above, the result is returned in argument Y. For functions, the result is returned as the value of the function. In the assignment statement above, the result is assigned to SUM.

See the 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, subf, identified in the ESSL calling sequence. If your program that calls the numerical quadrature subroutines is coded in Fortran, there are some coding rules you must follow:

- You must declare subf as EXTERNAL in your program.
- You should code the subf subroutine to the specifications given in “Programming Considerations for the SUBF Subroutine” on page 1144. For examples of coding a subf subroutine in Fortran, see the subroutine descriptions there.

Setting Up Scalar Data in Fortran

Table 41 on page 126 lists the scalar data types in Fortran that are used for ESSL. Only those types and lengths used by ESSL are listed.
### Table 41: 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(^1)</td>
<td>CHARACTER*1</td>
</tr>
<tr>
<td>32-bit logical item (^4)</td>
<td>LOGICAL or LOGICAL*4</td>
</tr>
<tr>
<td>.TRUE., .FALSE.</td>
<td>.TRUE., .FALSE.</td>
</tr>
<tr>
<td>64-bit logical item (^4)</td>
<td>LOGICAL or LOGICAL*8</td>
</tr>
<tr>
<td>.TRUE., .FALSE.</td>
<td>.TRUE., .FALSE.</td>
</tr>
<tr>
<td>32-bit integer (^2, 4)</td>
<td>INTEGER or INTEGER*4</td>
</tr>
<tr>
<td>12345, -12345</td>
<td>12345, -12345</td>
</tr>
<tr>
<td>64-bit integer (^4)</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(^3)</td>
<td>REAL or REAL*4</td>
</tr>
<tr>
<td>12.345</td>
<td>0.12345E2</td>
</tr>
<tr>
<td>Long-precision real number(^3)</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(^3)</td>
<td>COMPLEX or COMPLEX*8</td>
</tr>
<tr>
<td>(123.45, -54321.0)</td>
<td>(123.45E0, -543.21E2)</td>
</tr>
<tr>
<td>Long-precision complex number(^3)</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.
precision. See “How Do You Set Up Your Scalar Data?” on page 47 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:

\[
C(i) \leftarrow (a_i, b_i) \text{ for a one-dimensional array} \\
C(i, j) \leftarrow (a_{ij}, b_{ij}) \text{ for a two-dimensional array} \\
C(i, j, k) \leftarrow (a_{ijk}, b_{ijk}) \text{ for a three-dimensional array}
\]

**One-Dimensional Array**

For a one-dimensional array in Fortran 77, you can code:

```fortran
DIMENSION A(E1:E2)
```

where \( A \) is the name of the array, \( E1 \) is the lower bound, and \( E2 \) is the upper bound of the single dimension in the array. If the lower bound is not specified, such as in \( A(E2) \), the value is assumed to be 1. The upper bound is required.

A one-dimensional array is stored in ascending storage locations (relative to some base storage address) in the following order:

**Relative Location**

<table>
<thead>
<tr>
<th>Array Element</th>
<th>Value</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>( 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:

**Relative Location**

<table>
<thead>
<tr>
<th>Array Element</th>
<th>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>
</tbody>
</table>
Two-Dimensional Array

For a two-dimensional array in Fortran 77, you can code:

```fortran
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 295.

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:

<table>
<thead>
<tr>
<th>Relative Location</th>
<th>Array Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A(E1,F1) (starting column 1)</td>
</tr>
<tr>
<td>2</td>
<td>A(E1+1,F1)</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td>E2–E1+1</td>
<td>A(E2,F1)</td>
</tr>
<tr>
<td>(E2–E1+1)+1</td>
<td>A(E1,F1+1) (starting column 2)</td>
</tr>
<tr>
<td>(E2–E1+1)+2</td>
<td>A(E1+1,F1+1)</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td>(E2–E1+1)(2)</td>
<td>A(E2,F1+1)</td>
</tr>
<tr>
<td>(E2–E1+1)(2)+1</td>
<td>A(E1,F1+2) (starting column 3)</td>
</tr>
<tr>
<td>(E2–E1+1)(2)+2</td>
<td>A(E1+1,F1+2)</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
</tbody>
</table>
\((E_2 - E_1 + 1)(F_2 - F_1)\)
A\((E_2, F_2 - 1)\)

\((E_2 - E_1 + 1)(F_2 - F_1) + 1\)
A\((E_1, F_2)\) (starting column \(F_2 - F_1 + 1\))

\((E_2 - E_1 + 1)(F_2 - F_1) + 2\)
A\((E_1 + 1, F_2)\)

For example, the 3 by 4 array \(A\) specified in the DIMENSION statement as \(A(2:4,1:4)\) and containing the following elements:

\[
\begin{pmatrix}
11 & 12 & 13 & 14 \\
21 & 22 & 23 & 24 \\
31 & 32 & 33 & 34 \\
\end{pmatrix}
\]

has its elements arranged in storage as follows:

**Relative Location**

**Array Element Value**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11 (starting column 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>12 (starting column 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>13 (starting column 3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>14 (starting column 4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>24</td>
<td></td>
<td></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 1041.

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**

**Array Element**

1. $A(E1,F1,G1)$ (starting the first set)

2. $A(E1+1,F1,G1)$
   
   ..
   
   ..
   
   ..
   
   ..

$$(E2-E1+1)(F2-F1+1)$$

$A(E2,F2,G1)$

$$(E2-E1+1)(F2-F1+1)+1$$

$A(E1,F1,G1+1)$ (starting the second set)

$$(E2-E1+1)(F2-F1+1)+2$$

$A(E1+1,F1,G1+1)$

..
\[(E_2 - E_1 + 1)(F_2 - F_1 + 1)(G_2 - G_1 + 1)\]
A\((E_2, F_2, G_1 + 1)\)

\[(E_2 - E_1 + 1)(F_2 - F_1 + 1)(G_2 - G_1)\]
A\((E_2, F_2, G_2)\)

\[(E_2 - E_1 + 1)(F_2 - F_1 + 1)(G_2 - G_1 + 1)\]
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:

\[
\begin{array}{cccccc}
  & 111 & 121 &  & 112 & 122 &  & 113 & 123 &  & 114 & 124 &  \\
 11 & 211 & 221 &  & 212 & 222 &  & 213 & 223 &  & 214 & 224 &  \\
 311 & 321 &  & 312 & 322 &  & 313 & 323 &  & 314 & 324 &  \\
\end{array}
\]

has its elements arranged in storage as follows:

Relative Location
Array Element Value

1
111 (starting the first set)

2
211

3
311

4
121
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.

```fortran
program matinv_example
  implicit none
  ! program to invert m n x n random matrices
  !
  !  real(8), allocatable :: A(:, :, :), det(:, :), rcond(:)
  !  real(8)              :: dummy_aux, seed=1998, sd
  !  integer              :: rc, i, m=8, n=500, iopt=3, naux=0
  ! allocate storage
  !
  allocate(A(n, n, m), stat=rc)
  call error_exit(rc,"Allocation of matrix A")
  allocate(det(2, m), stat=rc)
  call error_exit(rc,"Allocation of det")
  allocate(rcond(m), stat=rc)
  call error_exit(rc,"Allocation of rcond")
  ! Calculate inverses in parallel
  !
  !SMP$ parallel do private(i, sd), schedule(static),
  !SMP$& share(n, a, iopt, rcond, det, dummy_aux, naux)
  do i=1, m
    sd = seed + 100*i
    call durand(sd, n*n, A(1,1,i))
    call dgeicd(A(1,1,i), n, n, iopt, rcond(i), det(1,i),
               dummy_aux, naux)
  enddo
  write(*,*)'Reciprocal condition numbers of the matrices are:'
  write(*,'(4E12.4)') rcond
  
  deallocate(A, stat=rc)
  deallocate(det, stat=rc)
  deallocate(rcond, stat=rc)
  call error_exit(rc,"Deallocation of rcond")

contains

subroutine error_exit(error_code, string)
  character(*)    :: string
  integer         :: error_code
  if(error_code .eq. 0 ) return
  write(0,*)("",error_code
  write(0,*)(")", failing return code was ", error_code
  stop 1
end subroutine error_exit
end
```

**Handling Errors in Your Fortran Program**

ESSL provides you with flexibilities in handling both input-argument errors and computational errors:

- For input-argument errors 2015, 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 Fortran” on page 134 and “Computational Errors in Fortran” on page 136 explain how to use these facilities by describing the additional statements you must code in your program.

For multithreaded application programs, if you want to initialize the error option table and change the default settings for input-argument and computational errors, you need to implement the steps shown in “Input-Argument Errors in Fortran” on page 134 and “Computational Errors in Fortran” on page 136 on each thread that calls ESSL. An example is shown in “Example of Handling Errors in a Multithreaded Application Program” on page 141.

**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 1188.

**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 69. For an example, see “Computational Errors in Fortran Example 3” on page 141, 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 67. For a description of the input-argument errors, see “Input-Argument Error Messages(2001-2099)” on page 187. For a description of ERRSET, see Chapter 17, “Utilities subroutines,” on page 1187.

**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 69. For an example, see “Computational Errors in Fortran Example 3” on page 141, 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.

```fortran
C                    DECLARE ENOTRM AS EXTERNAL 
EXTERNAL ENOTRM

C                    INITIALIZE THE ESSL ERROR
C                    OPTION TABLE
CALL EINFO(0)

C                    MAKE ERROR CODE 2015 A RECOVERABLE
C                    ERROR AND SUPPRESS PRINTING ALL
C                    ERROR MESSAGES FOR IT
CALL ERRSET(2015,0,-1,0,ENOTRM,2015)

C                    CALL ESSL ROUTINE SWLEV.
C                    IF THE NAUX INPUT
C                    ARGUMENT IS TOO SMALL, ERROR
C                    2015 OCCURS. THE MINIMUM VALUE
C                    REQUIRED IS STORED IN THE NAUX
C                    INPUT ARGUMENT AND CONTROL GOES
C                    TO LABEL 400.
CALL SWLEV(X,INCX,U,INCU,Y,INCY,N,AUX,NAUX,*400)

C                    CHECK THE RESULTING INPUT ARGUMENT
C                    VALUE IN NAUX AND TAKE THE
C                    DESIRED ACTION

400    .
```

**Computational Errors in Fortran**

To obtain information about an ESSL computational error in a Fortran program, add the statements in the following steps to your program. Steps 2 and 7 for ERRSAV and ERRSTR, respectively, are optional. Adding these steps makes the effect of the call to ERRSET temporary. For a list of those computational errors that return information and to which these steps apply, see “EINFO (ESSL Error Information-Handler Subroutine)” on page 1188.
**Step 1. 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 1188.

**Step 2. 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 69.

**Step 3. Call ERRSET**

```fortran
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 1188. 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 67. For a description of the computational errors, see “Computational Error Messages(2100-2199)” on page 191. For a description of ERRSET, see Chapter 17, “Utilities subroutines,” on page 1187.

**Step 4. Call ESSL**

```fortran
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**

```
  nmbr  CALL EINFO (icode,inf1)
  -or-
  nmbr  CALL EINFO (icode,inf1,inf2)
```

This calls the EINFO subroutine, which returns information about certain computational errors, where:

- **nmbr** is the statement number yyy, zzz, or any of the other statement numbers preceded by an “*” in the CALL statement in Step 4, corresponding to the return code value for this error code.
- **icode** is the error code of interest.
- **inf1** and **inf2** are the integer variables used to receive the information, where **inf1** is assigned a value for all errors, and **inf2** is assigned a value for some errors. For a description of EINFO, see “EINFO (ESSL Error Information-Handler Subroutine)” on page 1188.

**Step 6. Check the Values in the Information Receivers**

These statements check the values returned in the output argument information receivers, **inf1** and **inf2**, which contain the information about the computational error.

**Step 7. Call ERRSTR**

```
  CALL  ERRSTR (ierno,tabent)
```

(This is an optional step.) This calls the ERRSTR subroutine, which stores an entry in the error option table for error number **ierno** from an 8-byte storage area, **tabent**, which is accessible to your program. ERRSTR must be called for each entry you want to store. This step is used, along with step 2, for ERRSAV. For information on whether you should use ERRSAV and ERRSTR, see “How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?” on page 69.
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 (I1).

```
C                    INITIALIZE THE ESSL ERROR
C                    OPTION TABLE
CALL EINFO(0)

C                    ALLOW 100 ERRORS FOR CODE 2104
CALL ERRSET(2104,100,0,0,0_8,2104)

C                    CALL ESSL ROUTINE DPPF.
C                    IF THE INPUT MATRIX IS NOT
C                    POSITIVE DEFINITE, CONTROL GOES TO
C                    LABEL 400
IOPT=0
CALL DPPF(APP,N,IOPT,*400)

C                    CALL THE INFORMATION-HANDLER
C                    ROUTINE FOR ERROR CODE 2104 TO
C                    RETURN ONE PIECE OF INFORMATION
C                    IN VARIABLE I1, THE INDEX OF THE
C                    LAST NONPOSITIVE DIAGONAL FOUND
C                    BY ROUTINE DPPF
400    CALL EINFO (2104,I1)
```
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.

```fortran
C                    INITIALIZE THE ESSL ERROR
C                    OPTION TABLE
C                    CALL EINFO(0)
C                    ALLOW 100 ERRORS FOR CODE 2103
C                    CALL ERRSET(2103,100,0,0,0_8,2103)
C                    CALL ESSL SUBROUTINE DGEF.
C                    IF THE INPUT MATRIX IS
C                    SINGULAR, CONTROL GOES TO
C                    LABEL 400
C                    CALL DGEF(A,LDA,N,IPVT,*400)
C                    CALL THE INFORMATION-HANDLER
C                    ROUTINE FOR ERROR CODE 2103 TO
C                    RETURN ONE PIECE OF INFORMATION
C                    IN VARIABLE I1, THE INDEX OF THE
C                    LAST ZERO DIAGONAL FOUND BY
C                    SUBROUTINE DGEF
C                    CALL EINFO (2103,I1)
```
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 \( I_1 \) and the upper range \( I_2 \). It uses ERRSAV and ERRSTR to insulate the effects of the error handling for error 2100 by this program.

```fortran
C                   DECLARE AN AREA TO SAVE THE
C                   ERROR OPTION TABLE INFORMATION
C                   FOR ERROR CODE 2100
CHARACTER*8 SAV2100
C                   INITIALIZE THE ESSL ERROR
CALL EINFO(0)
C                   OPTION TABLE
C                   SAVE THE EXISTING ERROR OPTION
CALL ERRSAV(2100,SAV2100)
C                   ALLOW 255 ERRORS FOR CODE 2100
CALL ERRSET(2100,255,0,0,0_8,2100)
C                   CALL ESSL SUBROUTINE DQINT.
C                   IF AN INVALID INDEX IS
CALL DQINT(S,G,OMEGA,X,INCX,N,T,INCT,Y,INCY,M,*400)
C                   COMPUTED, CONTROL GOES TO LABEL 400
CALL EINFO (2100,I1,I2)
C                   CALL THE INFORMATION-HANDLER
C                   ROUTINE FOR ERROR CODE 2100 TO
C                   RETURN TWO PIECES OF INFORMATION.
C                   VARIABLE I1 CONTAINS THE LOWER RANGE
C                   FOR THE COMPUTED INDEX.
C                   VARIABLE I2 CONTAINS THE UPPER RANGE
C                   FOR THE COMPUTED INDEX.
400   CALL EINFO (2100,I1,I2)
C                   RESTORE THE PREVIOUS ERROR OPTION
CALL ERRSTR(2100,SAV2100)
```

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 133 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.

```fortran
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, 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,i18,r20)
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 errrst(2015,SAV2015(i))
  call errrst(2105,SAV2105(i))
enddo

write(*,*)'Reciprocal condition numbers of the matrices are:'
write(*, '(4E12.4)') rcond

deallocate(A, stat=rc)
call error_exit(rc,"Deallocation of matrix A")
deallocate(det, stat=rc)
call error_exit(rc,"Deallocation of det")
deallocate(rcond, stat=rc)
call error_exit(rc,"Deallocation of rcond")
stop contains
subroutine error_exit(error_code,string)
character(*)    :: string
integer         :: error_code
if(error_code .eq. 0 ) return
write(*,*) 'ERROR: ',string
write(0,*) string,": failing return code was ",error_code
stop 1
end subroutine error_exit
end
```

**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 146.

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
saxpy (5,a,x,incx,y,1);
```

The ESSL subroutines returning a function value are invoked with the following type of statement:

```c
function-value-name=subroutine-name (argument-1, . . . , argument-n);
```

An example of invoking DASUM might be:

```c
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 142.

No Optional Arguments

In the ESSL calling sequences for C, there are no optional arguments, as for some programming languages. You must code all the arguments listed in the syntax.

Arguments That Must Be Passed by Value

All scalar arguments that are not modified must be passed by value in the ESSL calling sequence. (This refers to input-only scalar arguments, such as incx, m, and lda.)

Arguments That Must Be Passed by Reference

Following are the instances in which you pass your arguments by reference (as a pointer) in the ESSL calling sequence:

Arrays

Arguments that are arrays are passed by reference, as usual.
**Subroutine Names**

Some ESSL subroutines call a user-supplied subroutine. The name is part of the ESSL calling sequence. It must be passed by reference.

**Output Scalar Arguments**

When an output argument is a scalar data item, it must be passed by reference. This is true for all scalar data types: real, complex, and so forth. When this occurs, it is listed in the notes of each subroutine description.

**Character Arguments**

Character arguments must be passed as strings, by reference. You specify the character, in upper- or lowercase, in the ESSL calling sequence with double quotation marks around it, as in "t". Following is an example of how you can call SGEADD, specifying the transa and transb arguments as strings n and t, respectively:

```
sgeadd(a,5,"n",b,3,"t",c,4,4,3);
```

**Altered Arguments When Using Error Handling**

If you use ESSL error handling in your C program, as described in “Handling Errors in Your C Program” on page 147, 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 147 and in step 1 in “Computational Errors in C” on page 151. The two types of ESSL arguments are:

- naux arguments for auxiliary storage
- n arguments for transform lengths

**Setting Up a User-Supplied Subroutine for ESSL in C**

Some ESSL numerical quadrature subroutines call a user-supplied subroutine, subf, identified in the ESSL calling sequence. If your program that calls the numerical quadrature subroutines is coded in C, there are some coding rules you must follow for the subf subroutine:

- You can code the subf subroutine using only C or Fortran.
- You must declare subf as an external subroutine in your application program.
- You should code the subf subroutine to the specifications given in “Programming Considerations for the SUBF Subroutine” on page 1144. For an example of coding a subf subroutine in C, see Example 1.

**Setting Up Scalar Data Types in C**

Table 42 on page 144 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>&quot;n&quot;, &quot;t&quot;, &quot;c&quot;</td>
</tr>
<tr>
<td>32-bit logical item</td>
<td>int</td>
</tr>
<tr>
<td>.TRUE., .FALSE.</td>
<td>For additional information, see “Using Logical Data in C” on page 145.2</td>
</tr>
</tbody>
</table>

Table 42: Scalar Data Types in C Programs.
### Table 42: Scalar Data Types in C Programs. (continued)

<table>
<thead>
<tr>
<th>Terminology Used by ESSL</th>
<th>C Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-bit logical item(^5)</td>
<td>long</td>
</tr>
<tr>
<td>.TRUE., .FALSE.</td>
<td>For additional information, see “Using Logical Data in C” on page 145.(^2)</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(^5)</td>
<td>long</td>
</tr>
<tr>
<td>12345L, -12345L</td>
<td></td>
</tr>
<tr>
<td>Short-precision real number(^4)</td>
<td>float</td>
</tr>
<tr>
<td>12.345</td>
<td></td>
</tr>
<tr>
<td>Long-precision real number(^4)</td>
<td>double</td>
</tr>
<tr>
<td>12.345</td>
<td></td>
</tr>
<tr>
<td>Short-precision complex number(^4)</td>
<td>Specify it as described in “Setting Up Complex Data Types in C” on page 145.</td>
</tr>
<tr>
<td>(123.45, -54321.0)</td>
<td></td>
</tr>
<tr>
<td>Long-precision complex number(^4)</td>
<td>Specify it as described in “Setting Up Complex Data Types in C” on page 145.</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 145.
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 ERSET.
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

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:

```csharp
#define FALSE 0
#define TRUE 1
```
For 64-bit logical arguments
Use this macro definition:

```c
#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 126. To pass an array from your C program to ESSL, to have ESSL process the data correctly, and to get a result that is in the proper form for your C program, you can do any of the following:

- Build and process the matrix, logically transposed from the outset, and transpose the results as necessary.
- Before the ESSL call, transpose the input arrays. Then, following the ESSL call, transpose any arrays updated as output.
- If there are arguments in the ESSL calling sequence indicating whether the arrays are to be processed in normal or transposed form, such as the `transa` and `transb` arguments in the `_GEMM` subroutines, use these arguments in combination with the matrix equivalence rules to avoid having to transpose your data in separate operations. For further detail, see “SGEMMS, DGEMMS, CGEMMS, and ZGEMMS (Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes Using Winograd’s Variation of Strassen’s Algorithm)” on page 408.

### Creating Multiple Threads and Calling ESSL from Your C Program

The 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 specify the `pthread.h` header file as the first include file in your program.
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” on page 147 and “Computational Errors in C” on page 151 explain how to use these facilities by describing the additional statements you must code in your program.

For multithreaded application programs, if you want to initialize the error option table and change the default settings for input-argument and computational errors, you need to implement the steps shown in “Input-Argument Errors in C” on page 147 and “Computational Errors in C” on page 151 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
/* 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**

```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 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;
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 1188. 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 69. For an example, see “Computational Errors in C Example” on page 155, as the use is the same as for computational errors.
Step 5. Call ERRSET

```
errset (&ierno,&inoal,&inomes,&itrace,&iusadr,&irange);
```

This calls the ERRSET subroutine, which allows you to dynamically modify the action taken when an error occurs. For optionally-recoverable ESSL input-argument errors, you need to call ERRSET only if you want to avoid terminating your program and you want the input arguments associated with this error to be assigned correct values in your program when the error occurs. For one error (ierno) or a range of errors (irange), you can specify:

- How many times each error can occur before execution terminates (inoal)
- How many times each error message can be printed (inomes)
- The ESSL exit routine ENOTRM, to be invoked for the error indicated (iusadr)

ERRSET must be called for each error code you want to indicate as being recoverable. For ESSL, ierno should have a value of 2015, 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 67. For a description of the input-argument errors, see “Input-Argument Error Messages(2001-2099)” on page 187. For a description of ERRSET, see Chapter 17, “Utilities subroutines,” on page 1187.

Step 6. Call ESSL

```
irc = name (arg1,...,argn);
if irc == rc1 {
    ...
}
```

This calls the ESSL subroutine and specifies a branch on one or more return code values, where:

- `name` specifies the ESSL subroutine.
- `arg1,...,argn` are the input and output arguments. As explained in step 1, all arguments that can potentially be altered by error handling must be coded by reference.
- `irc` is the integer variable containing the return code resulting from the computation performed by the ESSL subroutine.
- `rc1, rc2`, and so forth are the possible return code values that can be passed back from the ESSL subroutine to C. The values can be 0, 1, 2, and so forth. Return code values are described under “Error Conditions” in each ESSL subroutine description.

Step 7. Perform the Desired Action

These are the statements following the test for each value of the return code, returned in irc in step 6. These statements perform whatever action is desired when the recoverable error occurs. These statements may check the new values set in the input arguments to determine whether adequate program storage is available, and then decide whether to continue or terminate the program. Otherwise, these statements may check that the size of the working storage arrays or the length of the transform agrees with other data in the program. The program may also store this corrected input argument value for future reference.
Step 8. Call ERRSTR

errstr (&ierno, storarea);

(This is an optional step.) This calls the ERRSTR subroutine, which stores an entry in the error option table for error number `ierno` from an 8-byte storage area, `storarea`, which is accessible to your program. ERRSTR must be called for each entry you want to store. This step is used, along with step 4, for ERRSAV. For information on whether you should use ERRSAV and ERRSTR, see “How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?” on page 69. For an example, see “Computational Errors in C Example” on page 155, 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.

```c
/*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...
Step 1. Code the Global Statements for ESSL Error Handling

```c
/* Code two underscores */
/* before the letters ESVERR */
#define __ESVERR
#include <essl.h>
```

These statements are coded with your global declares in the front of your program. The `#define` must be coded before the `#include` statement for the ESSL header file. After the point where you code these statements in your program, you must pass by reference all ESSL calling sequence arguments that can potentially be altered by ESSL error handling. This applies to all your ESSL call statements. The two types of arguments are:

- `naux` arguments for auxiliary storage
- `n` arguments for transform lengths

Step 2. Declare the Variables

```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 1188. 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 69. For an example, see “Computational Errors in C Example” on page 155.
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 1188. 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 67. For a description of the computational errors, see “Computational Error Messages(2100-2199)” on page 191. For a description of ERRSET, see Chapter 17, “Utilities subroutines,” on page 1187.

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. As explained in step 1, all arguments that can potentially be altered by error handling must be coded by reference.
- `irc` is the integer variable containing the return code resulting from the computation performed by the ESSL subroutine.
- `rc1, rc2`, and so forth are the possible return code values that can be passed back from the ESSL subroutine to C. The values can be 0, 1, 2, and so forth. Return code values are described under “Error Conditions” in each ESSL subroutine description.

The statements following each test of the return code can perform any desired action. This includes calling EINFO for more information about the error, as described in step 7.

Step 7. Call EINFO for Information

```c
einfo (ierno,&inf1,&inf2);
```

This calls the EINFO subroutine, which returns information about certain computational errors, where:
• `ierno` is the error code of interest.
• `inf1` and `inf2` are the integer variables used to receive the information, where `inf1` is assigned a value for all errors, and `inf2` is assigned a value for some errors. You must specify both arguments, as there are no optional arguments for C. Both arguments must be passed by reference, because they are output scalar arguments. For a description of EINFO, see “EINFO (ESSL Error Information-Handler Subroutine)” on page 1188.

**Step 8. Check the Values in the Information Receivers**

These statements check the values returned in the output argument information receivers, `inf1` and `inf2`, which contain the information about the computational error.

**Step 9. Call ERRSTR**

```
errstr (&ierno, storarea);
```

(This is an optional step.) This calls the ERRSTR subroutine, which stores an entry in the error option table for error number `ierno` from an 8-byte storage area, `storarea`, which is accessible to your program. ERRSTR must be called for each entry you want to store. This step is used, along with step 4, for ERRSAV. For information on whether you should use ERRSAV and ERRSTR, see “How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?” on page 69. For an example, see “Computational Errors in C Example” on page 155.)
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
/*GLOBAL STATEMENTS FOR ESSL ERROR HANDLING*/
#define __ESVERR
#include <essl.h>

/*DECLARE THE VARIABLES*/
main ()
{
  int ierno,inoal,inomes,itrace,irange,irc;
  long int iusadr;
  int inf1,inf2,dummy;
  char sav2105[8];
  .
  /*INITIALIZE THE ESSL ERROR OPTION TABLE*/
einfo (0,&dummy,&dummy);
  /*SAVE THE EXISTING ERROR OPTION TABLE ENTRY
   FOR ERROR CODE 2105*/
  ierno = 2105;
  errsav (&ierno,sav2105);
  .
  /*MAKE ERROR CODES 2101 THROUGH 2105 RECOVERABLE
   ERRORS AND SUPPRESS PRINTING ALL ERROR MESSAGES
   FOR THEM. THIS SHOWS HOW YOU CODE THE
   ERRSET ARGUMENTS FOR A RANGE OF ERRORS. */
  ierno = 2101;
  inoal = 0;
  inomes = 0; /*A DUMMY ARGUMENT*/
  itrace = 0; /*A DUMMY ARGUMENT*/
  iusadr = 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*/
    .
    }
  .
  /*RESTORE THE PREVIOUS ERROR OPTION TABLE ENTRY
   FOR ERROR CODE 2105. ERROR PROCESSING
   RETURNS TO HOW IT WAS BEFORE IT WAS ALTERED BY
   THE ABOVE ERRSAV STATEMENT*/
  ierno = 2105;
  errstr (&ierno,sav2105);
  .
}
```

C++ Programs

This 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 essler.h.

In the beginning of your program, before you call any of the ESSL subroutines, you must code the following statement for the ESSL header file:

```
#include <essler.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 160.

Coding the Calling Sequences

In C++ programs, the ESSL subroutines, not returning a function value, are invoked with the following type of statement:

```
subroutine-name (argument-1, . . . , argument-n);
```

An example of a calling sequence for SAXPY might be:

```
saxpy (5,a,x,incx,y,1);
```

The ESSL subroutines returning a function value are invoked with the following type of statement:

```
function-value-name = subroutine-name (argument-1, . . . , argument-n);
```

An example of invoking DASUM might be:

```
sum = dasum (n,x,incx);
```

See the 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 155.

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 _ESVCPTR using one of the following methods:

  1. Define _ESVCPTR in your program using a `#define` statement, as shown below:

     ```
     #define _ESVCPTR
     ```

     This statement is coded with your global declares and must be coded before the `#include` statement for the ESSL header file.

  2. Define _ESVCPTR at compile time by using the job processing procedure described in “C++ Program Procedures” on page 174.

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 1144. For an example of coding a subf subroutine in C++, see Example 1.

**Setting Up Scalar Data in C++**
Table 43 on page 158 lists the scalar data types in C++ that are used for ESSL. Only those types and lengths used by ESSL are listed.
### Table 43: 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 1</td>
<td>char *</td>
</tr>
<tr>
<td>'N', 'T', 'C' or 'n', 't', 'c'</td>
<td>&quot;n&quot;, &quot;t&quot;, &quot;c&quot;</td>
</tr>
<tr>
<td>32-bit logical item 7</td>
<td>int</td>
</tr>
<tr>
<td>.TRUE., .FALSE.</td>
<td>For additional information, see “Using Logical Data in C++” on page 159.</td>
</tr>
<tr>
<td>64-bit logical item 7</td>
<td>long</td>
</tr>
<tr>
<td>.TRUE., .FALSE.</td>
<td>For additional information, see “Using Logical Data in C++” on page 159.</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 7</td>
<td>long</td>
</tr>
<tr>
<td>12345l, -12345l</td>
<td></td>
</tr>
<tr>
<td>Short-precision real number 4</td>
<td>float</td>
</tr>
<tr>
<td>12.345</td>
<td></td>
</tr>
<tr>
<td>Long-precision real number 4</td>
<td>double</td>
</tr>
<tr>
<td>12.345</td>
<td></td>
</tr>
<tr>
<td>Short-precision complex number 4</td>
<td>complex &lt;float&gt;</td>
</tr>
<tr>
<td>(123.45, -54321.0)</td>
<td>&quot;5&quot; on page 158, float_Complex</td>
</tr>
<tr>
<td>Long-precision complex number 4</td>
<td>complex &lt;double&gt;</td>
</tr>
<tr>
<td>(123.45, -54321.0)</td>
<td>&quot;5&quot; on page 158, double_Complex</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 159.
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 <complex.h>.
7. 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++

The ESSL header file supports both the Standard Numerics Library and C99 floating-point types for complex arithmetic (<complex.h>).
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 perform one of the following steps:

- 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 169.

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 126. To pass an array from your C++ program to ESSL, to have ESSL process the data correctly, and to get a result that is in the proper form for your C++ program, you can do any of the following:

- Build and process the matrix, logically transposed from the outset, and transpose the results as necessary.
- Before the ESSL call, transpose the input arrays. Then, following the ESSL call, transpose any arrays updated as output.
- If there are arguments in the ESSL calling sequence indicating whether the arrays are to be processed in normal or transposed form, such as the `transa` and `transb` arguments in the `_GEMM` subroutines, use
these arguments in combination with the matrix equivalence rules to avoid having to transpose your
data in separate operations. For further detail, see “SGEMMS, DGEMMS, CGEMMS, and ZGEMMS (Matrix
Multiplication for General Matrices, Their Transposes, or Conjugate Transposes Using Winograd’s
Variation of Strassen’s Algorithm)” on page 408.

Creating Multiple Threads and Calling ESSL from Your C++ Program

The 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 x1C_r command.

```c
#include "essl.h"
#include <iostream>
/* 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>
#pragma once
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()
{
    float sx2[8] = { 1.,12., 7., -8., -5., -10., -9., 19.};
    pthread_t first_th;
    pthread_t second_th;
    int rc;
    struct arg_list a_l,b_l;
    a_l.n = 9;
    a_l.incx = 1;
    a_l.x = sx1;
    b_l.n = 8;
    b_l.incx = 1;
    b_l.x = sx2;
    rc = pthread_create(&first_th, NULL, Thread, (void *) &a_l);
    if (rc) exit(-1);
    rc = pthread_create(&second_th, NULL, Thread, (void *) &b_l);
    if (rc) exit(-1);
    sleep(20);
    exit(0);
}
/* Thread routine which calls the ESSL subroutine ISAMAX */
void* Thread(void *v)
{
    struct arg_list *al;
    float *t;
    int n,incx;
    int i;
    al = (struct arg_list *)(v);
    t = al->x;
    n = al->n;
    incx = al->incx;
    /* Calling the ESSL subroutine ISAMAX */
    i = isamax(n,t,incx);
    if (i == 8)
        cout << "max for sx2 should be 8 = " << i << "\n";
    else
        cout << "max for sx1 should be 6 = " << i << "\n";
    return NULL;
}
```

Handling Errors in Your C++ Program

ESSL provides you with flexibilities in handling both input-argument errors and computational errors:

160 IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and Reference
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++” on page 161 and “Computational Errors in C++” on page 165 explain how to use these facilities by describing the additional statements you must code in your program.

For multithreaded application programs, if you want to initialize the error option table and change the default settings for input-argument and computational errors, you need to implement the steps shown in “Input-Argument Errors in C++” on page 161 and “Computational Errors in C++” on page 165 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**

```
/* Code one underscore */
/* before the letters ESVERR */
#define _ESVERR
#ifdef __linux
#include <iostream>
#else
#include <iostream.h>
#endif
#include <stdio.h>
#include <essl.h>
extern "C" int enotrm(int &,int &);
extern "C" typedef int (*FN) (int &,int &);
```

These statements are coded with your global declares in the front of your program. The `#define` must be coded before the `#include` statements for the ESSL header file. The `extern` statements are required to call the ESSL error exit routine ENOTRM as an external reference in your program.

**Step 2. Declare the Variables**

```
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**

```
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 1188. These statements should be coded only once in the beginning of your program before calls to ERRSET.

**Step 4. Call ERRSAV**

```
errsav (ierno,storarea);
```

(This is an optional step.) This calls the ERRSAV subroutine, which stores the error option table entry for error number `ierno` in an 8-byte storage area, `storarea`, which is accessible to your program. ERRSAV must be called for each entry you want to save. This step is used, along with step 8, for ERRSTR. For information on whether you should use ERRSAV and ERRSTR, see “How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?” on page 69. For an example, see “Computational Errors in C++ Example” on page 168, as the use is the same as for computational errors.

**Step 5. Call ERRSET**

```
erreset (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 67. For a description of the input-argument errors, see “Input-Argument Error Messages(2001-2099)” on page 187. For a description of ERRSET, see Chapter 17, “Utilities subroutines,” on page 1187.

**Step 6. Call ESSL**

```
irc = name (arg1,...,argn);
if irc == rc1
{
    ...
    ...
}
if irc == rc2
{
    ...
    ...
}
```
This calls the ESSL subroutine and specifies a branch on one or more return code values, where:

- **name** specifies the ESSL subroutine.
- **arg1, arg2, ..., 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**

```plaintext
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 69. For an example, see “Computational Errors in C++ Example” on page 168, 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.

```c
/*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*/
ieren = 2015;
inoal = 0;
inomes = -1;
itrace = 0;
irange = 2015;
errset (ieren,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 1188.

**Step 1. Code the Global Statements for ESSL Error Handling**

```cpp
/* Code one underscore */
/* before the letters ESVERR */
#define _ESVERR
#ifdef __linux
#include <iostream>
#else
#include <iostream.h>
#endif
#include <stdio.h>
#include <essl.h>
```

These statements are coded with your global declares in the front of your program. The `#define` must be coded before the `#include` statement for the ESSL header file.

**Step 2. Declare the Variables**

```cpp
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**

```cpp
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 1188. These statements should be coded only once in the beginning of your program before calls to ERRSET.

**Step 4. Call ERRSAV**

```cpp
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 69. For an example, see “Computational Errors in C++ Example” on page 168.
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 (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 1188. 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 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 67. For a description of the computational errors, see “Computational Error Messages(2100-2199)” on page 191. For a description of ERRSET, see Chapter 17, “Utilities subroutines,” on page 1187.

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.

The statements following each test of the return code can perform any desired action. This includes calling EINFO for more information about the error, as described in step 7.

Step 7. Call EINFO for Information

```c
einfo (ierno, inf1, inf2);
```

This calls the EINFO subroutine, which returns information about certain computational errors, where:
• `ierno` is the error code of interest.
• `inf1` and `inf2` are the integer variables used to receive the information, where `inf1` is assigned a value for all errors, and `inf2` is assigned a value for some errors. You must specify both arguments, as there are no optional arguments for C. For a description of EINFO, see “EINFO (ESSL Error Information-Handler Subroutine)” on page 1188.

**Step 8. Check the Values in the Information Receivers**

These statements check the values returned in the output argument information receivers, `inf1` and `inf2`, which contain the information about the computational error.

**Step 9. Call ERRSTR**

```
errstr (ierno, storarea);
```

(This is an optional step.) This calls the ERRSTR subroutine, which stores an entry in the error option table for error number `ierno` from an 8-byte storage area, `storarea`, which is accessible to your program. ERRSTR must be called for each entry you want to store. This step is used, along with step 4, for ERRSAV. For information on whether you should use ERRSAV and ERRSTR, see “How Can You Control Error Handling in Large Applications by Saving and Restoring Entries in the Error Option Table?” on page 69. For an example, see “Computational Errors in C++ Example” on page 168.
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.
Chapter 5. Processing Your Program

You must make ESSL-specific changes 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.

Consider the following notes when you process your program:

**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/lib</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 man page 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 1239), then you must specify `-L/usr/local/lib` instead of `-L/usr/local/lib64` in the commands in Table 45 on page 171, Table 47 on page 172, Table 48 on page 173, and Table 51 on page 175.

4. For the ESSL SMP and SMP CUDA Libraries, you can use the XLSPORTS 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 Software requirements.

### Fortran Program Procedures

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></td>
<td><code>xlf_r -O -qnosave xyz.f -lesslsmp -lxlsmp</code></td>
</tr>
<tr>
<td></td>
<td>64-bit integer, 64-bit pointer</td>
<td><code>xlf_r -O -qnosave -qsmp xyz.f -lesslsmp6464</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>xlf_r -O -qnosave xyz.f -lesslsmp6464 -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/lib64 -R/usr/local/cuda/lib64</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>xlf_r -O -qnosave xyz.f -lesslsmpcuda -lxlsmp -lcublas -lcudart -L/usr/local/cuda/lib64 -R/usr/local/cuda/lib64</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 `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

```fortran
#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 45: Fortran Compile Commands on Linux 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, 64-bit pointer</td>
<td><code>xlf_r -0 -qnosave xyz.f -lessl</code> <code>-l/usr/local/include -lfftw3_essl -L/usr/local/lib64</code></td>
</tr>
<tr>
<td></td>
<td>32-bit integer, 64-bit pointer</td>
<td><code>xlf_r -0 -qnosave xyz.f -lesslomp</code> <code>-l/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>xlf_r -0 -qnosave xyz.f -lesslomp</code> <code>-l/usr/local/include -lfftw3_essl -L/usr/local/lib64</code></td>
</tr>
<tr>
<td></td>
<td>32-bit integer, 64-bit pointer</td>
<td><code>xlf_r -0 -qnosave xyz.f -lesslomp</code> <code>-l/usr/local/include -lfftw3_essl -L/usr/local/lib64</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 1239.

### C Program Procedures

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 46: C Compile and Link Commands.

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>C Compile Command</th>
</tr>
</thead>
</table>
Table 46: C Compile and Link Commands. (continued)

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>C Compile Command</th>
</tr>
</thead>
</table>

**Note:** In the commands listed in the table above, you must specify the following values:

- `xlf_version.release`
  
  > 15.1.6 or 16.1.0

- `xlsmp_version.release`
  
  > 4.1.6 or 5.1.0

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 47: C Compile and Link Commands for Use with FFTW Wrapper Libraries.

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>C Compile Command</th>
</tr>
</thead>
</table>

**Note:** In the commands listed in the table above, you must specify the following values:

- `xlf_version.release`
  
  > 15.1.6 or 16.1.0

- `xlsmp_version.release`
  
  > 4.1.6 or 5.1.0

For additional information on the FFTW Wrapper libraries, see Appendix C, “FFTW Version 3.1.2 to ESSL Wrapper Libraries,” on page 1239.

If you want to use gcc compile and link commands, use the commands shown in Table 48 on page 173.
Table 48: gcc Compile and Link Commands.

<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 48 on page 173, you must specify the following values:

- **xlf_version.release**
  > 15.1.6 or 16.1.0

- **xlsmp_version.release**
  > 4.1.6 or 5.1.0

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 49: gcc Compile and Link Commands for Use with FFTW Wrapper Libraries.

<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 49 on page 174, you must specify the following values:

- `xlf_version.release`  
  > 15.1.6 or 16.1.0<

- `xlsmp_version.release`  
  > 4.1.6 or 5.1.0<

C++ Program Procedures

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 50: C++ Compile and Link Commands.

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>C++ Compile Command</th>
</tr>
</thead>
</table>
### Table 50: C++ Compile and Link Commands. (continued)

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>C++ Compile Command</th>
</tr>
</thead>
</table>

| SMP CUDA     | 32-bit integer, 64-bit pointer | `xlC_r -O xyz.C -lessl -lxlf90_r -lxlf90 -lxlfmath -lcublas -lcudart -lm -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`
  - `15.1.6` or `16.1.0`<
- `xlsmp_version.release`
  - `4.1.6` or `5.1.0`<

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 51: C++ Compile and Link Commands for Use with FFTW Wrapper Libraries.

<table>
<thead>
<tr>
<th>ESSL Library</th>
<th>Environment</th>
<th>C++ Compile Command</th>
</tr>
</thead>
</table>

**Note:** In the commands listed in the table above, you must specify the following values:

- `xlf_version.release`
  - `15.1.6` or `16.1.0`<
- `xlsmp_version.release`
  - `4.1.6` or `5.1.0`<
For additional information on the FFTW Wrapper libraries, see Appendix C, “FFTW Version 3.1.2 to ESSL Wrapper Libraries,” on page 1239.

If you want to use g++ compile and link commands, use the commands shown in Table 52 on page 176.

<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++  
-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 | g++  
-xyz.C  
- 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++  
-xyz.C  
- lesslsmmp -lxlf90_r -lxlsmp -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++  
-xyz.C  
- lesslsmmp6464 -lxlf90_r -lxlsmp -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++  
-xyz.C  
- lesslsmmpcuda -lxlf90_r -lxlsmp -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 52 on page 176, you must specify the following values:

- **xlf_version.release**
  - >15.1.6 or 16.1.0<

- **xlsmp_version.release**
  - >14.1.6 or 5.1.0<

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 53: g++ Compile and Link Commands for Use with FFTW Wrapper Libraries.

<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 53 on page 177, you must specify the following values:

- `xlf_version.release`
  >15.1.6 or 16.1.0<

- `xlsmp_version.release`
  >4.1.6 or 5.1.0<
Chapter 6. Migrating programs to ESSL

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 5 to Version 6 Release 1

The calling sequences for the subroutines in ESSL Version 5 Release 5 and ESSL Version 6 Release 1 are identical. Therefore, your application programs do not require any changes.

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 54 on page 179, changes are required. As shown in Table 54 on page 179, CBLAS enumerated types have changed to the following type definitions:

<table>
<thead>
<tr>
<th>Table 54: CBLAS enumerated types changing to type definitions for ESSL Version 5 Release 5.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ESSL V5R4 enumerated type</strong></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>

⁴ 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.
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 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 1239.

**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. Troubleshooting ESSL

This provides the following information for your use when dealing with errors.

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 Linux
   >15765-L61<

   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. 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. Enter the following command:

   `rpm -q package` where the appropriate values of `package` are listed in Table 55 on page 183.

5. The message that is returned when an error is detected.
6. Any error message relating to core dumps.

<table>
<thead>
<tr>
<th>Descriptive Name</th>
<th>Product Packages</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESSL</td>
<td>essl.rte</td>
</tr>
<tr>
<td></td>
<td>essl.3264.rte</td>
</tr>
<tr>
<td></td>
<td>essl.3264.rtecuda</td>
</tr>
<tr>
<td></td>
<td>essl.6464.rte</td>
</tr>
<tr>
<td>XL Fortran Runtime Environment</td>
<td>libxlf</td>
</tr>
<tr>
<td>SMP Runtime Environment</td>
<td>libxlsmp</td>
</tr>
<tr>
<td>XL Fortran compiler</td>
<td>&gt;xlf.15.1.6, or 16.1.0&lt;</td>
</tr>
<tr>
<td>XL C compiler</td>
<td>&gt;xlc.13.1.6, or 16.1.0&lt;</td>
</tr>
<tr>
<td>XL C++ compiler</td>
<td>&gt;xlc.13.1.6, or 16.1.0&lt;</td>
</tr>
</tbody>
</table>
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.

Globalization

For globalization, 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:

```
/opt/ibmmath/essl/6.1/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.

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 programs,” on page 125.
- 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 187 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 191 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 programs,” on page 125.

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 an 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.
- Check the setting of your user ID's user limit (ulimit).

**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 185.

**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 184.

**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'.

186 IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and Reference
**Message Format**

The ESSL messages are issued in your output in the following format:

\[ \text{rtn-name : 2538-mmnn} \]

\[ \text{message-text} \]

*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 193

<table>
<thead>
<tr>
<th>Code</th>
<th>Message 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>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>
</tbody>
</table>
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. _).

The transform length (ARG NO. _) must be less than or equal to (\).

The transform length (ARG NO. _) must be greater than or equal to (\).

The routine must be initialized with the present value of (ARG NO. _).

The number of elements (ARG NO. _) in a work array must be greater than or equal to (\).

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'.

The dimension (ARG NO. _) of the matrices must be greater than or equal to zero.

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. _).

The number of sequences (ARG NO. _) must be greater than zero.

(ARG NO. _) must be nonzero.

The storage control switch (ARG NO. _) must be 1, 2, 3, or 4.

(ARG NO. _) must be less than (\).

The outer loop increment (ARG NO. _) must be greater than or equal to zero.

The stride (ARG NO. _) for a vector must be greater than zero.

The stride (ARG NO. _) for a vector must be greater than zero.

The stride (ARG NO. _) for a vector must be greater than or equal to (\).

The order (ARG NO. _) of a matrix must be greater than or equal to zero.

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, 11, 20, 21, 30, or 31; 0, 1, 2, 3, or 4).

The job option argument (ARG NO. _) must be 0 or 1.

The transform length (ARG NO. _) is not an allowed value. The next higher allowed value is (\).

The resulting convolution length obtained from ARG NO. 10 = (\), ARG NO. 11 = (\), ARG NO. 13 = (\), and ARG NO. 14 = (\) must be less than (\).

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.

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. _).
TRANSA (ARG NO. _) specifies whether an input matrix (ARG NO. _), its transpose, or its conjugate transpose should be used. TRANSA must be 'N', 'T', or 'C'.

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; 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 1.

The data type parameter (ARG NO. _) must be 'S', 'D', 'C', or 'Z'.

(ARG NO. _) must be greater than or equal to (ARG NO. _) and smaller than (ARG NO. _).

The matrix is singular. Column (_) 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 (ARG NO. _); greater than or equal to element (_) and less than or equal to (ARG NO. _); greater than or equal to 1 and less than or equal to (ARG NO. _); 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 1.

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 'I' or 'S'.

Element (_) in (ARG NO. _) must be [one of the following: greater than or equal to 0; greater than or equal to (ARG NO. _); greater than (ARG NO. _); and less than or equal to (ARG NO. _)].
Computational Error Messages (2100-2199)

2538-2100 The computed index of a vector is out of the range (_) to (_).

2538-2101 Eigenvalue (_) failed to converge after (_) iterations.

2538-2102 Eigenvector (_) failed to converge after (_) iterations.

2538-2103 The matrix (ARG NO. _) is singular. Zero diagonal element (_) has been detected.

2538-2104 The matrix (ARG NO. _) is not positive definite. The last diagonal element with nonpositive value is (_).

2538-2105 Factorization failed due to near zero pivot number (_).

2538-2106 Vector boundary misalignment detected in ESSL scalar library.

2538-2107 Singular value (_) failed to converge after (_) iterations.

2538-2108 The matrix specified by (ARG NO. _) and (ARG NO. _) is not definite because the diagonal is not of constant sign.

2538-2109 The matrix specified by (ARG NO. _) and (ARG NO. _) is not definite and the iterative process is stopped at iteration number (_).

2538-2110 The maximum allowed number of iterations, element number (_) of (ARG NO. _), were performed but the iterative process did not converge to a solution according to the stopping procedure.

2538-2111 The factorization matrix (ARG NO. _) is not consistent with the sparse matrix specified by (ARG NO. _) and (ARG NO. _).

2538-2112 The incomplete factorization of the sparse matrix specified by (ARG NO. _) and (ARG NO. _) is not stable.

2538-2113 Unexpected nonzero vector mask detected in ESSL scalar routine. Contact your IBM Service Representative.

2538-2114 Eigenvalue (_) failed to converge after (_) iterations.

2538-2115 The matrix (ARG NO. _) is not positive definite. The leading minor of order (_) has a nonpositive determinant.

2538-2116 The matrix specified by (ARG NO. _) and (ARG NO. _) is singular.

2538-2117 The pivot element in column (_) is smaller than the first element in (ARG NO. _).

2538-2118 The pivot element in row (_) is smaller than the first element in (ARG NO. _).
The storage space, specified by (ARG NO. _), is insufficient.

The matrix is singular. The last row processed in the matrix was row (_).

The matrix is singular. The last column processed was column (_).

The factorization failed. No pivot element was found in the active submatrix.

Performance can be improved by specifying a larger value for (ARG NO. _). (_ ) compressions were performed.

The data contained in AUX1, (ARG NO. _), was computed for a different algorithm.

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.

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 Gerschgorin interval initially used was incorrect.

(_ ) eigenvectors failed to converge after ( _ ) iterations. The indices are stored in IFAIL (ARG NO. _).

Eigenvalue ( _ ) failed to converge in the QZ iteration. Arrays ALPHA (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 ALPHA (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 ( _ ).

Eigenvalue ( _ ) failed to converge in the QZ iteration. Arrays ALPHA (ARG NO. _) and BETA (ARG NO. _) contain the eigenvalues successfully computed. For more information, refer to Engineering and Scientific Subroutine Library Guide and Reference.

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.

The dimension (ARG NO. _) of the array (ARG NO. _) must be greater 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.

2538-2207 The number of elements in the array (ARG NO. _) must be less than or equal to (_).

2538-2208 ANORM (ARG NO. _) must be equal to zero or greater than or equal to (_) and less than or equal to (_).

2538-2209 NORM (ARG NO. _), which specifies the computation to be performed, must be 'M', '1', 'O', 'I', 'F', or 'E'.

2538-2210 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'.

2538-2211 The alignment of (ARG NO. _) changed after initialization.

2538-2212 JOBZ (ARG NO. _), which specifies whether or not to compute eigenvectors, must be 'N' or 'V'.

2538-2213 RANGE (ARG NO. _), which specifies which eigenvalues to find, must be 'A', 'V', or 'I'.

2538-2214 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.

2538-2215 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. _).

2538-2216 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.

2538-2217 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'.

2538-2218 JOBVL (ARG NO. _), which specifies whether or not to compute left eigenvectors, must be 'N' or 'V'.

2538-2219 JOBVR (ARG NO. _), which specifies whether or not to compute right eigenvectors, must be 'N' or 'V'.

2538-2220 SENSE (ARG NO. _), which specifies which reciprocal condition numbers are to be computed, must be 'N', 'E', 'V', or 'B'.

2538-2221 JOBVL (ARG NO. _) and JOBVR (ARG NO. _) must be 'V' if SENSE (ARG. NO _) is 'E' or 'B'.

2538-2222 ITYPE (ARG NO. _), which specifies the problem type, must be 1, 2, or 3.

2538-2223 The routine must be initialized with the present value of element (_) of (ARG NO. _).

2538-2224 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'.

2538-2225 The lower bandwidth (ARG NO. _) must be less than the number of rows (ARG NO. _) of the matrix.

2538-2226 The upper bandwidth (ARG NO. _) must be less than the number of columns (ARG NO. _) of the matrix.

2538-2227 JOBU (ARG NO. _), which specifies whether or not to compute left singular vectors, must be 'N', 'A', 'S', or 'O'.

2538-2228 JOBVT (ARG NO. _), which specifies whether or not to compute left singular vectors, must be 'N', 'A', 'S', or 'O'.

2538-2229 JOBU (ARG NO. _) and JOBVT (ARG NO. _) cannot both be 'O'.

2538-2230 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. _).

2538-2231 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 CblasTrans.

The form (ARG NO. _) of a matrix must be CblasNoTrans, CblasTrans, or CblasConjTrans.

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 'B'.

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.

Miscellaneous Error Messages(2700-2799)
2538-2700 Internal ESSL error number ( ). Contact your IBM service representative.

Miscellaneous Error Messages(2700-2799)
2538-2700 Internal ESSL error number ( ). Contact your IBM service representative.

Performance may be improved by specifying an even value for (ARG NO. )

2538-2703 Internal ESSL error: message number requested ( ) is outside of the valid range. Contact your IBM service representative.

Performance may be improved by specifying a multiple of four for (ARG NO. ).

ESSL chose to use GPUs during the previous initialization call, but GPUs may not be used by the current computational call because it was made from an OpenMP parallel region.

Unable to locate message number ( ). Please refer to 'Using Error Handling' in the ESSL Guide and Reference for the full message text.
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” on page 197)
- Sparse vector-scalar linear algebra subprograms (“Sparse Vector-Scalar Linear Algebra Subprograms” on page 200)
- Matrix-vector linear algebra subprograms (“Matrix-Vector Linear Algebra Subprograms” on page 200)
- Sparse matrix-vector linear algebra subprograms (“Sparse Matrix-Vector Linear Algebra Subprograms” on page 202)

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 [“95” on page 1258]. 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 56: 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†</td>
<td>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 204</td>
</tr>
<tr>
<td>ICAMAX†</td>
<td>ICAMAX†</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†</td>
<td>IDAMIN†</td>
<td>“ISAMIN and IDMIN (Position of the First or Last Occurrence of the Vector Element Having Minimum Absolute Value)” on page 206</td>
</tr>
<tr>
<td>cblas_isamin</td>
<td>cblas_idamin</td>
<td></td>
</tr>
<tr>
<td>ISMAX†</td>
<td>IDMAX†</td>
<td>“ISMAX and IDMAX (Position of the First or Last Occurrence of the Vector Element Having the Maximum Value)” on page 209</td>
</tr>
<tr>
<td>cblas_ismax</td>
<td>cblas_idmax</td>
<td></td>
</tr>
<tr>
<td>ISMIN†</td>
<td>IDMIN†</td>
<td>“ISMIN and IDMIN (Position of the First or Last Occurrence of the Vector Element Having Minimum Value)” on page 211</td>
</tr>
<tr>
<td>cblas_ismin</td>
<td>cblas_idmin</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>SASUM(^\d)</td>
<td>DASUM(^\d)</td>
<td>“SASUM, DASUM, SCASUM, and DZASUM (Sum of the Magnitudes of the Elements in a Vector)” on page 214</td>
</tr>
<tr>
<td>SCASUM(^\d)</td>
<td>DZASUM(^\d)</td>
<td></td>
</tr>
<tr>
<td>cblas_sasum(^\d)</td>
<td>cblas_dasum(^\d)</td>
<td></td>
</tr>
<tr>
<td>cblas_scasum(^\d)</td>
<td>cblas_dzasum(^\d)</td>
<td></td>
</tr>
<tr>
<td>&gt; 1</td>
<td>DAXPBY(^\d)</td>
<td>“SAXPBY and DAXPBY (Scales Vector X and Vector Y by two Scalars, Adds them to one another, and Stores in the Vector Y)” on page 219</td>
</tr>
<tr>
<td>SAXPBY(^\d)</td>
<td>cblas_daxpby(^\d)</td>
<td></td>
</tr>
<tr>
<td>SCOPY(^\d)</td>
<td>DCOPY(^\d)</td>
<td>“SCOPY, DCOPY, CCOPY, and ZCOPY (Copy a Vector)” on page 223</td>
</tr>
<tr>
<td>CCOPY(^\d)</td>
<td>ZCOPY(^\d)</td>
<td></td>
</tr>
<tr>
<td>cblas_scopy(^\d)</td>
<td>cblas_dcopy(^\d)</td>
<td></td>
</tr>
<tr>
<td>cblas_ccopy(^\d)</td>
<td>cblas_zcopy(^\d)</td>
<td></td>
</tr>
<tr>
<td>SDOT(^\d)</td>
<td>DDOT(^\d)</td>
<td>“SDOT, DDOT, CDOTU, ZDOTU, CDOTC, and ZDTC (Dot Product of Two Vectors)” on page 225</td>
</tr>
<tr>
<td>CDOTU(^\d)</td>
<td>ZDOTU(^\d)</td>
<td></td>
</tr>
<tr>
<td>CDOTC(^\d)</td>
<td>ZDOTC(^\d)</td>
<td></td>
</tr>
<tr>
<td>cblas_sdot(^\d)</td>
<td>cblas_ddot(^\d)</td>
<td></td>
</tr>
<tr>
<td>cblas_cdotu(^\d)</td>
<td>cblas_zdotu(^\d)</td>
<td></td>
</tr>
<tr>
<td>cblas_cdotc(^\d)</td>
<td>cblas_zdotc(^\d)</td>
<td></td>
</tr>
<tr>
<td>SNDOT</td>
<td>DNDOT</td>
<td>“SNDOT and DNDOT (Compute Special Dot Products N Times)” on page 233</td>
</tr>
<tr>
<td>SNRM2(^\d)</td>
<td>DNRM2(^\d)</td>
<td>“SNRM2, DNRM2, SCNRM2, and DZNRM2 (Euclidean Length of a Vector with Scaling of Input to Avoid Destructive Underflow and Overflow)” on page 238</td>
</tr>
<tr>
<td>SCNRM2(^\d)</td>
<td>DZNRM2(^\d)</td>
<td></td>
</tr>
<tr>
<td>cblas_snrm2(^\d)</td>
<td>cblas_dnrn2(^\d)</td>
<td></td>
</tr>
<tr>
<td>cblas_scnrn2(^\d)</td>
<td>cblas_dznrm2(^\d)</td>
<td></td>
</tr>
<tr>
<td>SNORM2(^\d)</td>
<td>DNM2(^\d)</td>
<td>“SNORM2, DNORM2, CNORM2, and ZNORM2 (Euclidean Length of a Vector with No Scaling of Input)” on page 241</td>
</tr>
<tr>
<td>CNORM2(^\d)</td>
<td>ZNM2(^\d)</td>
<td></td>
</tr>
<tr>
<td>cblas_srotg(^\d)</td>
<td>cblas_drotg(^\d)</td>
<td></td>
</tr>
<tr>
<td>cblas_crotg(^\d)</td>
<td>cblas_zrotg(^\d)</td>
<td></td>
</tr>
<tr>
<td>SROTG(^\d)</td>
<td>DROTG(^\d)</td>
<td>“SROTG, DROTG, CROTG, and ZROTG (Construct a Givens Plane Rotation)” on page 244</td>
</tr>
<tr>
<td>CROTG(^\d)</td>
<td>ZROTG(^\d)</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>SROT*</td>
<td>DROT*</td>
<td>“SROT, DROT, CROT, ZROT, CSROT, and ZDROT (Apply a Plane Rotation)” on page 250</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>cblas_csrot*</td>
<td>cblas_zdrot*</td>
<td></td>
</tr>
<tr>
<td>SROTMG*</td>
<td>DROTMG*</td>
<td>“SROTMG and DROTMG (Construct a modified Givens Transformation)” on page 254</td>
</tr>
<tr>
<td>cblas_srotmg*</td>
<td>cblas_drotnmg*</td>
<td></td>
</tr>
<tr>
<td>SROTM*</td>
<td>DROTM*</td>
<td>“SROTM and DROTM (Apply a modified Givens Transformation)” on page 256</td>
</tr>
<tr>
<td>cblas_srotm*</td>
<td>cblas_drotnm*</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 259</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_scskal*</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 262</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 265</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 269</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_cvces</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 273</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 276</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>
</tbody>
</table>
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 ["39" on page 1255]. Both real and complex versions of the subprograms are provided.

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 ["44" on page 1255] and ["45" on page 1255]. Both real and complex versions of the subprograms are provided.
<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 295</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_dgemv*</td>
<td></td>
</tr>
<tr>
<td>cblas_zgemv*</td>
<td>cblas_dgemv*</td>
<td></td>
</tr>
<tr>
<td>cblas_zgemv*</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 306</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>SSPMV*</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 313</td>
</tr>
<tr>
<td>CHPMV*</td>
<td>ZHPMV*</td>
<td></td>
</tr>
<tr>
<td>SSYMV*</td>
<td>ZSYMV*</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_zhpvm*</td>
<td>cblas_dsymv*</td>
<td></td>
</tr>
<tr>
<td>cblas_zhemv*</td>
<td>cblas_zhemv*</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 321</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>cblas_dspr*</td>
<td></td>
</tr>
<tr>
<td>cblas_zhpr*</td>
<td>cblas_dsymr*</td>
<td></td>
</tr>
<tr>
<td>cblas_zher*</td>
<td>cblas_dsymr*</td>
<td></td>
</tr>
<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 328</td>
</tr>
<tr>
<td>CHPR2*</td>
<td>ZHPR2*</td>
<td></td>
</tr>
<tr>
<td>SSYR2*</td>
<td>DSYR2*</td>
<td></td>
</tr>
<tr>
<td>CHER2*</td>
<td>ZHER2*</td>
<td></td>
</tr>
<tr>
<td>SSLR2§</td>
<td>cblas_dspr2*</td>
<td></td>
</tr>
<tr>
<td>cblas_zhpr2*</td>
<td>cblas_dsymr2*</td>
<td></td>
</tr>
<tr>
<td>cblas_zher2*</td>
<td>cblas_dsymr2*</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 336</td>
</tr>
<tr>
<td>CGBMV*</td>
<td>ZGBMV*</td>
<td></td>
</tr>
<tr>
<td>cblas_sgbmv*</td>
<td>cblas_dgbmv*</td>
<td></td>
</tr>
<tr>
<td>cblas_zgbmv*</td>
<td>cblas_dgbmv*</td>
<td></td>
</tr>
<tr>
<td>cblas_zgbmv*</td>
<td>cblas_zgbmv*</td>
<td></td>
</tr>
</tbody>
</table>
## 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 59: 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 371</td>
</tr>
</tbody>
</table>
Table 59: List of Sparse Matrix-Vector Linear Algebra Subprograms. (continued)

<table>
<thead>
<tr>
<th>Long-Precision Subprogram</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSMTM</td>
<td>“DSMTM (Transpose a Sparse Matrix in Compressed-Matrix Storage Mode)” on page 378</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 374</td>
</tr>
</tbody>
</table>

Use Considerations

If your program uses a sparse matrix stored by rows, as defined in “Storage-by-Rows” on page 115, 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 1212). 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 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 110.

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 61.
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 60: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td></td>
</tr>
<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 34.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>ISAMAX</td>
</tr>
<tr>
<td>C and C++</td>
<td>isamax</td>
</tr>
<tr>
<td>CBLAS</td>
<td>clblas_isamax</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 60 on page 204.

\( incx \)

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 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 \( \text{incx} \geq 0 \), \( i = k \)

For \( \text{incx} < 0 \), \( i = n-k+1 \)

See reference [“95” on page 1258]. 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>IMAX = ISAMAX( 9, X, 1 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( X = (1.0, 2.0, 7.0, -8.0, -5.0, -10.0, -9.0, 10.0, 6.0) \)

Output:

IMAX = 6

**Example 2**

This example shows a vector, \( x \), with a stride greater than 1.
Function Reference and Input:

\[
\begin{aligned}
N & \quad X & \quad \text{INCX} \\
\text{IMAX} & = \text{ISAMAX}( 5 \ , \ X \ , \ 2 ) \\
X & = (1.0, \ . \ , 7.0, \ . \ , -5.0, \ . \ , -9.0, \ . \ , 6.0) \\
\end{aligned}
\]

Output:

\[
\begin{aligned}
\text{IMAX} & = 4 \\
\end{aligned}
\]

**Example 3**
This example shows a vector, \( x \), with a stride of 0.

Function Reference and Input:

\[
\begin{aligned}
N & \quad X & \quad \text{INCX} \\
\text{IMAX} & = \text{ISAMAX}( 9 \ , \ X \ , \ 0 ) \\
X & = (1.0, \ . \ , \ . \ , \ . \ , \ . \ , \ . \ , \ . \ , \ . \ , \ . \ , \ . \ , \ . \) \\
\end{aligned}
\]

Output:

\[
\begin{aligned}
\text{IMAX} & = 1 \\
\end{aligned}
\]

**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:

\[
\begin{aligned}
N & \quad X & \quad \text{INCX} \\
\text{IMAX} & = \text{ISAMAX}( 8 \ , \ X \ , \ -2 ) \\
X & = (3.0, \ . \ , 5.0, \ . \ , -8.0, \ . \ , 6.0, \ . \ , 8.0, \ . \ , 4.0, \ . \ , 8.0, \ . \ , 2.0) \\
\end{aligned}
\]

Output:

\[
\begin{aligned}
\text{IMAX} & = 7 \\
\end{aligned}
\]

**Example 5**
This example shows a vector, \( x \), containing complex numbers and having a stride of 1.

Function Reference and Input:

\[
\begin{aligned}
N & \quad X & \quad \text{INCX} \\
\text{IMAX} & = \text{ICAMAX}( 5 \ , \ X \ , \ 1 ) \\
X & = ((9.0 , 2.0) , (7.0 , -8.0) , (-5.0 , -10.0) , (-4.0 , 10.0) , (6.0 , 3.0)) \\
\end{aligned}
\]

Output:

\[
\begin{aligned}
\text{IMAX} & = 2 \\
\end{aligned}
\]

**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>
<thead>
<tr>
<th>Table 61: Data Types.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Types</strong></td>
</tr>
<tr>
<td><strong>x</strong></td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>ISAMIN</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
<tr>
<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 34.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>ISAMIN</td>
</tr>
<tr>
<td>C and C++</td>
<td>isamin</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_isamin</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 61 on page 207.

\[
incx\]

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 \( \text{incx} \), \( i \) has the following values:

For \( \text{incx} \geq 0 \), \( i = k \)
For \( \text{incx} < 0 \), \( i = n-k+1 \)

See reference [“95” on page 1258]. 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:

\[
\begin{array}{|c|c|c|}
\hline
N & X & \text{INCX} \\
\hline
\text{IMIN} = \text{ISAMIN}(6, X, 1) \\
X = (3.0, 4.0, 1.0, 8.0, 1.0, 3.0) \\
\hline
\end{array}
\]

Output:

\[
\begin{array}{c}
\text{IMIN} = 3 \\
\end{array}
\]

**Example 2**
This example shows a vector, \( x \), with a stride greater than 1.

Function Reference and Input:

\[
\begin{array}{|c|c|c|}
\hline
N & X & \text{INCX} \\
\hline
\text{IMIN} = \text{ISAMIN}(4, X, 2) \\
X = (-3.0, . , -9.0, . , -8.0, . , 3.0) \\
\hline
\end{array}
\]

Output:

\[
\begin{array}{c}
\text{IMIN} = 1 \\
\end{array}
\]

**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:

\[
\begin{array}{|c|c|c|}
\hline
N & X & \text{INCX} \\
\hline
\text{IMIN} = \text{ISAMIN}(4, X, 2) \\
X = (2.0, . , -1.0, . , 4.0, . , 1.0) \\
\hline
\end{array}
\]

Output:

\[
\begin{array}{c}
\text{IMIN} = 2 \\
\end{array}
\]
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:

\[
\begin{array}{|c|c|c|}
\hline
N & X & \text{INCX} \\
\hline\hline
\text{IMIN} &= \text{ISAMIN}(4, X, -2) \\
X &= (2.0, \ , \ -1.0, \ , \ 4.0, \ , \ 1.0) \\
\hline
\end{array}
\]

Output:

\[
\text{IMIN} = 4
\]

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>
<thead>
<tr>
<th>Table 62: Data Types.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Types</strong></td>
</tr>
<tr>
<td><strong>x</strong></td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
<tr>
<td><strong>Subprogram</strong></td>
</tr>
<tr>
<td>ISMAX</td>
</tr>
<tr>
<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 34.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>ISMAX</td>
</tr>
<tr>
<td>C and C++</td>
<td>ismax</td>
</tr>
<tr>
<td>CBLAS</td>
<td>blas_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 \( 1+(n-1)|\text{incx}| \), containing numbers of the data type indicated in Table 62 on page 209.

\( \text{incx} \)

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 \text{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 \( \text{incx} \), \( i \) has the following values:

For \( \text{incx} \geq 0 \), \( i = k \)
For \( \text{incx} < 0 \), \( i = n-k+1 \)

See reference [\cite{95} on page 1258]. 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>IMAX = ISMAX(6, X, 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X = (3.0, 4.0, 1.0, 8.0, 1.0, 8.0)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output:

| IMAX | = 4 |

Example 2

This example shows a vector, \( x \), with a stride greater than 1.
Function Reference and Input:

\[
\begin{array}{c|c|c}
N & X & INCX \\
\hline
IMAX & = & ISMAX( 4, X, 2 ) \\
X & = & (-3.0, . , 9.0, . , -8.0, . , 3.0) \\
\end{array}
\]

Output:

\[
\begin{array}{c}
IMAX & = & 2 \\
\end{array}
\]

**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:

\[
\begin{array}{c|c|c}
N & X & INCX \\
\hline
IMAX & = & ISMAX( 4, X, 2 ) \\
X & = & (2.0, . , 4.0, . , 4.0, . , 1.0) \\
\end{array}
\]

Output:

\[
\begin{array}{c}
IMAX & = & 2 \\
\end{array}
\]

**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:

\[
\begin{array}{c|c|c}
N & X & INCX \\
\hline
IMAX & = & ISMAX( 4, X, -2 ) \\
X & = & (2.0, . , 4.0, . , 4.0, . , 1.0) \\
\end{array}
\]

Output:

\[
\begin{array}{c}
IMAX & = & 3 \\
\end{array}
\]

**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>
<thead>
<tr>
<th>Table 63: Data Types.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Types</strong></td>
</tr>
<tr>
<td>( x )</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td><strong>Subprogram</strong></td>
</tr>
<tr>
<td>ISMIN</td>
</tr>
</tbody>
</table>

Linear algebra subprograms 211
Table 63: Data Types. (continued)

<table>
<thead>
<tr>
<th>Data Types</th>
<th>IDMIN</th>
</tr>
</thead>
<tbody>
<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 34.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</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 \( 1+(n-1)|incx| \), containing numbers of the data type indicated in Table 63 on page 211.

- **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 [“95” on page 1258]. 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:

\[
\begin{array}{|c|c|c|}
\hline
N & X & INCX \\
\hline
IMIN &= ISMIN( 6 , X , 1 ) \\
X &= (3.0, 4.0, 1.0, 8.0, 1.0, 3.0) \\
\hline
\end{array}
\]

Output:

\[
\begin{array}{c}
IMIN = 3 \\
\end{array}
\]

**Example 2**

This example shows a vector, \( x \), with a stride greater than 1.

Function Reference and Input:

\[
\begin{array}{|c|c|c|}
\hline
N & X & INCX \\
\hline
IMIN &= ISMIN( 4 , X , 2 ) \\
X &= (-3.0, . \ , -9.0, . \ , -8.0, . \ , 3.0) \\
\hline
\end{array}
\]

Output:

\[
\begin{array}{c}
IMIN = 2 \\
\end{array}
\]

**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:

\[
\begin{array}{|c|c|c|}
\hline
N & X & INCX \\
\hline
IMIN &= ISMIN( 4 , X , 2 ) \\
X &= (2.0, . \ , 1.0, . \ , 4.0, . \ , 1.0) \\
\hline
\end{array}
\]

Output:

\[
\begin{array}{c}
IMIN = 2 \\
\end{array}
\]

**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:

\[
\begin{array}{|c|c|c|}
\hline
N & X & INCX \\
\hline
\end{array}
\]

Linear algebra subprograms 213
MIN = 4, X = (2.0, 1.0, 4.0)

Output:
MIN = 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 64: Data Types.

<table>
<thead>
<tr>
<th>Data Types</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 34.

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 64 on page 214.
- **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 64 on page 214.
**Note:** Declare this function in your program as returning a value of the type indicated in Table 64 on page 214.

**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 [“95” on page 1258]. 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 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:

\[
\begin{array}{ccc}
N & X & INCX \\
SUMM = SASUM( 7 , X , 1 ) \\
X = (1.0, -3.0, -6.0, 7.0, 5.0, 2.0, -4.0)
\end{array}
\]

Output:

\[
SUMM = 28.0
\]

**Example 2**

This example shows a vector, \( x \), with a stride greater than 1.

Function Reference and Input:

\[
\begin{array}{ccc}
N & X & INCX \\
SUMM = SASUM( 4 , X , 2 )
\end{array}
\]
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:

\[
\begin{array}{c|c|c|c|c|c|c|c}
N & X & INCX \\
---&---&---
SUMM = SASUM( 4, X, -2 )
\end{array}
\]

\[
X = (1.0, \ , -6.0, \ , 5.0, \ , -4.0)
\]

Output:

\[
SUMM = 16.0
\]

Example 4
This example shows a vector, \( x \), with a stride of 0. The result in \( \text{SUMM} \) is \( nx_1 \).

Function Reference and Input:

\[
\begin{array}{c|c|c|c|c|c|c|c}
N & X & INCX \\
---&---&---
SUMM = SASUM( 7, X, 0 )
\end{array}
\]

\[
X = (-2.0, \ , \ , \ , \ , \ , \ , \ )
\]

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:

\[
\begin{array}{c|c|c|c|c|c|c|c}
N & X & INCX \\
---&---&---
SUMM = SCASUM( 5, X, 1 )
\end{array}
\]

\[
X = ((1.0, 2.0), (-3.0, 4.0), (5.0, -6.0), (-7.0, -8.0), (9.0, 10.0))
\]

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 65: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha, x, y$</td>
<td></td>
</tr>
<tr>
<td>Short-precision real</td>
<td>SAXPY</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DAXPY</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CAXPY</td>
</tr>
<tr>
<td>Long-precision complex</td>
<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 34.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SAXPY</td>
</tr>
<tr>
<td>C and C++</td>
<td>saxpy</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_saxpy</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 65 on page 217.

$x$

is the vector $x$ of length $n$. Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 65 on page 217.

$incx$

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 $1+(n-1)|incy|$, containing numbers of the data type indicated in Table 65 on page 217.

$incy$

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 65 on page 217.

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 “Vector concepts” on page 71.

**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 [“95” on page 1258]. If \( \text{alpha} \) or \( n \) is zero, no computation is performed. For CAXPY, intermediate results are accumulated in long precision when the 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
CALL  SAXPY( 5 , 2.0 , X , 1  , Y , 2  )
```

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

\( Y = (3.0, 5.0, 7.0, 9.0, 11.0) \)

**Example 2**

This example shows vectors \( x \) and \( y \) having strides of opposite signs. For \( y \), which has negative stride, processing begins at element \( Y(5) \), which is 1.0.

Call Statement and Input:

```
N  ALPHA  X  INCX  Y  INCY
CALL  SAXPY( 5 , 2.0 , X , 1  , Y , -1  )
```

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

\( Y = (15.0, 12.0, 9.0, 6.0, 3.0) \)
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:
\[
\begin{array}{c|c|c|c|c|c|c}
N & \text{ALPHA} & \text{INCX} & \text{Y} & \text{INCY} \\
\hline
5 & \alpha & 0 & (5.0, 4.0, 3.0, 2.0, 1.0) \\
\end{array}
\]

Output:
\[
\begin{array}{c}
Y = (7.0, 6.0, 5.0, 4.0, 3.0) \\
\end{array}
\]

Example 4
This example shows how \textsc{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:
\[
\begin{array}{c|c|c|c|c|c|c}
N & \text{ALPHA} & \text{INCX} & \text{Y} & \text{INCY} \\
\hline
1 & \alpha & 0 & (5.0) \\
\end{array}
\]

Output:
\[
\begin{array}{c}
Y = (7.0) \\
\end{array}
\]

Example 5
This example shows how to use \textsc{caxpy}, where vectors \( x \) and \( y \) contain complex numbers. In this case, vectors \( x \) and \( y \) have positive strides.

Call Statement and Input:
\[
\begin{array}{c|c|c|c|c|c|c}
N & \text{ALPHA} & \text{INCX} & \text{Y} & \text{INCY} \\
\hline
3 & \alpha & 1 & (5.0, 4.0, 3.0) \\
\end{array}
\]

\textsc{saxpby} and \textsc{daxpby} (Scales Vector X and Vector Y by Two Scalars, Adds them to one another, and Stores in the Vector Y)

Purpose
These subprograms perform the following computation, by using the scalars \( \alpha \) and \( \beta \), and vectors \( x \) and \( y \):
\[
y \leftarrow \alpha x + \beta y
\]
Table 66: Data Types.

<table>
<thead>
<tr>
<th>alpha, beta, x, y</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SAXPBY</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DAXPBY</td>
</tr>
</tbody>
</table>

**Note:** On certain processors, SIMD algorithms can be used if alignment requirements are met. For more information, see “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 34.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SAXPBY</td>
</tr>
<tr>
<td>C and C++</td>
<td>saxpby</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_saxpby</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 66 on page 220.

- **x**
  - is the vector x of length n.
  - Specified as: a one-dimensional array of (at least) length \( 1+(n-1)|incx| \), which contains numbers of the data type indicated in Table 66 on page 220.

- **incx**
  - is the stride for vector x.
  - Specified as: an integer. The incx argument can have any value.

- **beta**
  - is the scalar beta.
  - Specified as: a number of the data type indicated in Table 66 on page 220.

- **y**
  - is the vector y of length n.
  - Specified as: a one-dimensional array of (at least) length \( 1+(n-1)|incy| \), which contains numbers of the data type indicated in Table 66 on page 220.

- **incy**
  - is the stride for vector y.
  - Specified as: an integer. The incy argument can have any value.

**On Return**

- **y**
  - is the vector y, which contains the results of the computation \( \alpha x + \beta y \).
  - Returned as: a one-dimensional array, which contains numbers of the data type indicated in Table 66 on page 220.
Notes
1. If you specify same values for vectors x and y, the incx and incy arguments must be equal; otherwise, results are unpredictable.
2. If you specify different values for vectors x and y, the vectors must not have common elements; otherwise, results are unpredictable. See “Vector concepts” on page 71.

Function
The computation is expressed as follows:

\[
\begin{bmatrix}
  y_1 \\
  \vdots \\
  y_n \\
\end{bmatrix}
\begin{bmatrix}
  \alpha \\
  \vdots \\
  \beta \\
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  \vdots \\
  x_n \\
\end{bmatrix}
\]

See reference [“95” on page 1258]. Computation is not performed in the following cases:
• n is zero.
• alpha is zero and beta is one.

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:

\[
\begin{array}{c}
| \text{N} | \text{ALPHA} | \text{X} | \text{INCX} | \text{BETA} | \text{Y} | \text{INCY} |
|---|---|---|---|---|---|
\end{array}
\]

CALL SAXPBY ( 5 , 2.0 , X , 1 , 1.0, Y , 2 )

\[
X = (1.0, 2.0, 3.0, 4.0, 5.0)
\]

\[
Y = (1.0, \ldots, 1.0, \ldots, 1.0, \ldots, 1.0)
\]

Output:

\[
Y = (3.0, \ldots, 5.0, \ldots, 7.0, \ldots, 9.0, \ldots, 11.0)
\]

Example 2
This example shows vectors x and y with strides of opposite signs. For vector y, which has negative stride, processing begins at element Y(5), which is 1.0.

Call Statement and Input:

\[
\begin{array}{c}
<table>
<thead>
<tr>
<th>\text{N}</th>
<th>\text{ALPHA}</th>
<th>\text{X}</th>
<th>\text{INCX}</th>
<th>\text{BETA}</th>
<th>\text{Y}</th>
<th>\text{INCY}</th>
</tr>
</thead>
</table>
\end{array}
\]

Output:
CALL SAXPBY ( 5 , 2.0 , X , 1 , 1.0,  Y , -1 )

\[
X = (1.0, 2.0, 3.0, 4.0, 5.0) \\
Y = (5.0, 4.0, 3.0, 2.0, 1.0)
\]

Output:

\[
Y = (15.0, 12.0, 9.0, 6.0, 3.0)
\]

**Example 3**

This example shows a vector, \( x \), with 0 stride. Vector \( x \) is treated like a vector of length \( n \), all of whose elements are the same as the single element in vector \( x \).

Call Statement and Input:

\[
\begin{align*}
\text{N} & \quad \text{ALPHA} & \quad \text{X} & \quad \text{INCX} & \quad \text{BETA} & \quad \text{Y} & \quad \text{INCY} \\
\text{CALL SAXPBY} & \quad (5 , 2.0 , X , 0 , 1.0, Y , 0 )
\end{align*}
\]

\[
X = (1.0) \\
Y = (5.0, 4.0, 3.0, 2.0, 1.0)
\]

Output:

\[
Y = (7.0, 6.0, 5.0, 4.0, 3.0)
\]

**Example 4**

This example shows a vector, \( x \) that has a positive stride, a vector, \( y \) that has a zero stride, and a vector count that is greater than 1.

Call Statement and Input:

\[
\begin{align*}
\text{N} & \quad \text{ALPHA} & \quad \text{X} & \quad \text{INCX} & \quad \text{BETA} & \quad \text{Y} & \quad \text{INCY} \\
\text{CALL SAXPBY} & \quad (5 , 2.0 , X , 1 , 1.0, Y , 0 )
\end{align*}
\]

\[
X = (1.0, 2.0, 3.0, 4.0, 5.0) \\
Y = (5.0)
\]

Output:

\[
Y = (15.0)
\]

**Example 5**

This example shows a vector, \( x \) that has a negative stride, a vector, \( y \) that has a zero stride, and a vector count that is greater than 1.

Call Statement and Input:

\[
\begin{align*}
\text{N} & \quad \text{ALPHA} & \quad \text{X} & \quad \text{INCX} & \quad \text{BETA} & \quad \text{Y} & \quad \text{INCY} \\
\text{CALL SAXPBY} & \quad (5 , 2.0 , X , -1 , 1.0, Y , 0 )
\end{align*}
\]

\[
X = (1.0, 2.0, 3.0, 4.0, 5.0) \\
Y = (5.0)
\]

Output:

\[
Y = (7.0)
\]

**Example 6**

This example shows how SAXPBY 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:

```
<table>
<thead>
<tr>
<th>N</th>
<th>ALPHA</th>
<th>X</th>
<th>INCX</th>
<th>BETA</th>
<th>Y</th>
<th>INCY</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALL SAXPBY ( 5 , 2.0 , X, 0 , 1.0, Y , 0 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

X = (1.0)
Y = (5.0)

Output:

Y = (7.0)

**SCOPY, DCOPY, CCOPY, and ZCOPY (Copy a Vector)**

**Purpose**

These subprograms copy vector \( x \) to another vector, \( y \):

\( y \leftarrow x \)

**Table 67: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x, y )</td>
<td>SCOPY</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>DCOPY</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>CCOPY</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>ZCOPY</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td></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 34.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SCOPY</td>
</tr>
<tr>
<td>C and C++</td>
<td>`scopy</td>
</tr>
<tr>
<td>CBLAS</td>
<td>`cblas_scopy</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 \( 1+(n-1)|incx| \), containing numbers of the data type indicated in Table 67 on page 223.

- \( incx \)
  - is the stride for vector \( x \).
Specified as: an integer. It can have any value.

\( y \)

See On Return.

\( incy \)

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 \( 1+(n-1)|incy| \), containing numbers of the data type indicated in Table 67 on page 223.

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 “Vector concepts” on page 71.

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 [“95” on page 1258]. 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:

\[
\begin{array}{cccccc}
N & X & INCX & Y & INCY \\
\hline
5 & X & 1 & Y & 2 \\
\end{array}
\]

\[
\text{CALL SCOPY( 5 , X , 1 , Y , 2 )}
\]

\[
X = (1.0, 2.0, 3.0, 4.0, 5.0)
\]

Output:

\[
Y = (1.0, . , 2.0, . , 3.0, . , 4.0, . , 5.0)
\]
Example 2
This example shows how to obtain a reverse copy of the input vector $x$ by specifying strides with the same absolute value, but with opposite signs, for input vector $x$ and output vector $y$. For $y$, which has a negative stride, results are stored beginning at element $Y(5)$.

Call Statement and Input:

```
N   X  INCX  Y  INCY
  |   |    |    |
CALL SCOPY( 5 , X , 1  , Y , -1  )
X     = (1.0, 2.0, 3.0, 4.0, 5.0)
```

Output:

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

Example 3
This example shows an input vector, $x$, with 0 stride. Vector $x$ is treated like a vector of length $n$, all of whose elements are the same as the single element in $x$. This is a technique for replicating an element of a vector.

Call Statement and Input:

```
N   X  INCX  Y  INCY
  |   |    |    |
CALL SCOPY( 5 , X , 0  , Y , 1  )
X     = (13.0)
```

Output:

```
Y     = (13.0, 13.0, 13.0, 13.0, 13.0)
```

Example 4
This example shows input vector $x$ and output vector $y$, containing complex numbers and having positive strides.

Call Statement and Input:

```
N   X  INCX  Y  INCY
  |   |    |    |
CALL CCOPY( 4 , X , 1  , Y , 2  )
X     = ((1.0, 1.0), (2.0, 2.0), (3.0, 3.0), (4.0, 4.0))
```

Output:

```
Y     = ((1.0, 1.0), . , (2.0, 2.0), . , (3.0, 3.0), . ,
         (4.0, 4.0))
```

SDOT, DDOT, CDOTU, ZDOTU, CDOTC, and ZDOTC (Dot Product of Two Vectors)

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 $y$:

$$ \overline{x} \cdot y $$
Table 68: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x, y, \text{dotu}, \text{dotc}, \text{Result}$</td>
<td></td>
</tr>
<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 34.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>SDOT</td>
</tr>
<tr>
<td>C and C++</td>
<td>sdot</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_sdot</td>
</tr>
<tr>
<td></td>
<td>cblas_cdotu_sub</td>
</tr>
<tr>
<td></td>
<td>cblas_cdotc_sub</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 68 on page 226.

$\text{incx}$

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 68 on page 226.

$\text{incy}$

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 68 on page 226.

$\text{dotu}$

is the result of the dot product computation.

Returned as: a number of the data type indicated in Table 68 on page 226.

$\text{dotc}$

is the result of the dot product computation.
Notes
Declare this function in your program as returning a value of the data type indicated in Table 68 on page 226.

Function
SDOT, DDOT, CDOTU, and ZDOTU compute the dot product of the vectors \( \mathbf{x} \) and \( \mathbf{y} \), which is expressed as follows:

\[
\mathbf{x} \cdot \mathbf{y} = x_1 y_1 + x_2 y_2 + \ldots + x_n y_n
\]

CDOTC and ZDOTC compute the dot product of the complex conjugate of vector \( \mathbf{x} \) with vector \( \mathbf{y} \), which is expressed as follows:

\[
\overline{\mathbf{x}} \cdot \mathbf{y} = \overline{x_1} y_1 + \overline{x_2} y_2 + \ldots + \overline{x_n} y_n
\]

See reference [“95” on page 1258]. 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 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, \( \mathbf{x} \) and \( \mathbf{y} \), having strides of 1.

Function Reference and Input:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( N )</td>
<td>( X )</td>
<td>( \text{INCX} )</td>
<td>( Y )</td>
<td>( \text{INCY} )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{DOTT} = \text{SDOT}( 5, X, 1, Y, 1 ) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

\( Y = (9.0, 8.0, 7.0, -6.0, 5.0) \)

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, \( \mathbf{x} \), with a stride of 1, and a vector, \( \mathbf{y} \), with a stride greater than 1.

Function Reference and Input:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( N )</td>
<td>( X )</td>
<td>( \text{INCX} )</td>
<td>( Y )</td>
<td>( \text{INCY} )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{DOTT} = \text{SDOT}( 5, X, 1, Y, 2 ) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

\( Y = (9.0, . , 7.0, . , 5.0, . , -3.0, . , 1.0) \)
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:

\[
\begin{array}{c|c|c|c|c|}
N & X & INCX & Y & INCY \\
\hline
5 & X & -1 & Y & 2 \\
\end{array}
\]

\begin{align*}
X &= (1.0, 2.0, -3.0, 4.0, 5.0) \\
Y &= (9.0, 7.0, 5.0, -3.0, 1.0)
\end{align*}

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 \(x_1 y_1 + \ldots + x_n y_n\).

Function Reference and Input:

\[
\begin{array}{c|c|c|c|c|}
N & X & INCX & Y & INCY \\
\hline
5 & X & 0 & Y & 1 \\
\end{array}
\]

\begin{align*}
X &= (1.0, \ldots, \ldots) \\
Y &= (9.0, 8.0, 7.0, -6.0, 5.0)
\end{align*}

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 \(nx_1 y_1\).

Function Reference and Input:

\[
\begin{array}{c|c|c|c|c|}
N & X & INCX & Y & INCY \\
\hline
5 & X & 0 & Y & 0 \\
\end{array}
\]

\begin{align*}
X &= (1.0, \ldots, \ldots) \\
Y &= (9.0, \ldots, \ldots)
\end{align*}

Output:

\[
\text{DOTT} = (5) \times (1.0) \times (9.0) = 45.0
\]

Example 6
This example shows how to compute the dot product of two vectors, \(x\) and \(y\), containing complex numbers, where \(x\) has a stride of 1, and \(y\) has a stride greater than 1.

Function Reference and Input:

\[
\begin{array}{c|c|c|c|c|}
N & X & INCX & Y & INCY \\
\hline
3 & X & 1 & Y & 2 \\
\end{array}
\]

\begin{align*}
X &= ((1.0, 2.0), (3.0, -4.0), (-5.0, 6.0)) \\
Y &= ((10.0, 9.0), \ldots, (2.0, 1.0))
\end{align*}

Output:

\[
\text{DOTT} = (1.0 + 2.0 - 4.0 - 5.0 + 6.0) = 2.0
\]
Example 7
This example shows how to compute the dot product of the conjugate of a vector, $x$, with vector $y$, both containing complex numbers, where $x$ has a stride of 1, and $y$ has a stride greater than 1.

Function Reference and Input:

<table>
<thead>
<tr>
<th>N</th>
<th>X</th>
<th>INCX</th>
<th>Y</th>
<th>INCY</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOTT = CDOTC ( 3 , X , 1 , Y , 2 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$X = ((1.0, 2.0), (3.0, -4.0), (-5.0, 6.0))$

$Y = ((10.0, 9.0), . , (-6.0, 5.0), . , (2.0, 1.0))$

Output:

$DOTT = ((10.0 - 18.0 - 10.0) + (18.0 - 20.0 + 6.0),$

$(9.0 + 15.0 - 5.0) + (20.0 + 24.0 + 12.0))$

$= (-14.0, -37.0)$

SNAXPY and DNAXPY (Compute SAXPY or DAXPY N Times)

Purpose
These subprograms compute SAXPY or DAXPY, respectively, $n$ times:

$y_i \leftarrow y_i + \alpha x_i$ 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. 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 216.

Table 69: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$, $x$, $y$</td>
<td>SNAXPY</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>SNAXPY</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DNAXPY</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SNAXPY</td>
</tr>
<tr>
<td>C and C++</td>
<td>snaxpy</td>
</tr>
</tbody>
</table>

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 $\alpha_i$, used in each computation of $y_i + \alpha_i x_i$.
Specified as: a one-dimensional array of (at least) length $1+(n-1)|inca|$, containing numbers of the data type indicated in Table 69 on page 229.

$inca$ 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 $y_i + \alpha_i x_i$.
Specified as: a one- or two-dimensional array of (at least) length $1+(n-1)(incxo) + (m-1)|incxi|$, containing numbers of the data type indicated in Table 69 on page 229.

$incxi$ 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 $n$ computations of $y_i + \alpha_i x_i$.
Specified as: a one- or two-dimensional array of (at least) length $1+(n-1)(incyo) + (m-1)|incyi|$, containing numbers of the data type indicated in Table 69 on page 229.

$incyi$ 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; $incyi > 0$ or $incyi < 0$.

$incyo$ 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
$y$ 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 69 on page 229.

Notes
Vector $y$ must have no common elements with vector $a$ or vector $x$; otherwise, results are unpredictable. See “Vector concepts” on page 71.

Function
The SAXPY or DAXPY computations:

$$y \leftarrow y + \alpha x$$

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 “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 216) uses the length of the $x_i$ and $y_i$ vectors, $m$, for its input argument, $n$. It also uses the strides for the inner loop, $incx$ and $incy$, for its parameters $incx$ and $incy$, respectively. See “Function” on page 218 for a description of how the computation is done.

The outer loop of the SNAXPY or DNAXPY computation uses the strides of $inca$, $incxo$, and $incyo$ to locate the elements in $a$ and vectors in $x$ and $y$ for each $i$-th computation. These are:

For $i = 1, n$:

- $\alpha_{((i-1)\text{inca}+1)}$ for $inca \geq 0$
- $\alpha_{((i-n)\text{inca}+1)}$ for $inca < 0$
- $x_{((i-1)\text{incxo}+1)}$
- $y_{((i-1)\text{incyo}+1)}$

If $m$ or $n$ is 0, no computation is performed.

**Error conditions**

**Computational Errors**

None

**Input-Argument Errors**

1. $n < 0$
2. $m < 0$
3. $incxo < 0$
4. $incy_i = 0$
5. $incyo < 0$

**Examples**

**Example 1**

This example shows vectors, contained in matrices, with the stride of the inner loops $incx$ and $incy$ 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)
```

```
X = | 1.0 4.0 3.0 |
   | 2.0 3.0 4.0 |
   | 3.0 2.0 2.0 |
   | 4.0 1.0 1.0 |
   | . . .      |
   | . . .      |
   | . . .      |
   | . . .      |
   | . . .      |

Y = | 4.0 1.0 3.0 |
   | 3.0 2.0 4.0 |
   | 2.0 3.0 2.0 |
   | 1.0 4.0 1.0 |
```
Example 2

This example shows vectors, contained in matrices, with a stride of the inner loop incxi greater than 1.

Call Statement and Input:

\[
\begin{array}{ccccccccccc}
N & M & A & INCA & X & INCXI & INCXO & Y & INCYI & INCYO \\
\hline
3 & 4 & A & 1 & X & 2 & 10 & Y & 1 & 5 \\
\end{array}
\]

A = (3.0, 2.0, 4.0)

\[
\begin{array}{cccc}
1.0 & 4.0 & 3.0 \\
2.0 & 3.0 & 4.0 \\
3.0 & 2.0 & 2.0 \\
4.0 & 1.0 & 1.0 \\
\end{array}
\]

\[
\begin{array}{cccc}
4.0 & 1.0 & 3.0 \\
3.0 & 2.0 & 4.0 \\
2.0 & 3.0 & 2.0 \\
1.0 & 4.0 & 1.0 \\
\end{array}
\]

Y = (same as output Y in Example 1)

Example 3

This example shows vectors, contained in matrices, with a negative stride, incyi, for the inner loop.

Call Statement and Input:

\[
\begin{array}{ccccccccccc}
N & M & A & INCA & X & INCXI & INCXO & Y & INCYI & INCYO \\
\hline
3 & 4 & A & 1 & X & 1 & 10 & Y & -1 & 5 \\
\end{array}
\]

A = (3.0, 2.0, 4.0)

\[
\begin{array}{cccc}
1.0 & 4.0 & 3.0 \\
2.0 & 3.0 & 4.0 \\
3.0 & 2.0 & 2.0 \\
4.0 & 1.0 & 1.0 \\
\end{array}
\]
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 A(5), which is 3.0.

Call Statement and Input:

```plaintext
CALL SNAXPY( 3 , 4 , A , -2 , X ,  1  , 10  , Y ,  1  ,  5  )
```

\text{A} = (4.0, . , 2.0, . , 3.0)

\begin{align*}
\text{X} &= \begin{pmatrix} 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{pmatrix} \\
\text{Y} &= \begin{pmatrix} 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{pmatrix}
\end{align*}

Output:

\begin{align*}
\text{Y} &= \begin{pmatrix} 13.0 & 6.0 & 5.0 \\ 11.0 & 7.0 & 10.0 \\ 9.0 & 8.0 & 20.0 \\ 7.0 & 9.0 & 15.0 \\ . & . & . \end{pmatrix}
\end{align*}

Y = (same as output Y in Example 1)

**SNDOT and DNDOT (Compute Special Dot Products N Times)**

**Purpose**

These subprograms compute one of the following special dot products \( n \) times:

**Special Dot Products**

\[ s_i \leftarrow x_i \cdot y_i \quad \text{Store positive dot product} \]

\[ s_i \leftarrow -x_i \cdot y_i \quad \text{Store negative dot product} \]
Special Dot Products

\[ s_i \leftarrow \begin{cases} s_i + x_i \cdot y_i & \text{Accumulate positive dot product} \\ s_i - x_i \cdot y_i & \text{Accumulate negative dot product} \end{cases} \]

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>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s, x, y )</td>
<td>SNDOT</td>
</tr>
<tr>
<td>Short-precision real</td>
<td></td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DNDOT</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SNDOT</td>
</tr>
<tr>
<td>C and C++</td>
<td>sdot</td>
</tr>
</tbody>
</table>

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 \( 1+(n-1)|incs| \), containing numbers of the data type indicated in Table 70 on page 234.

- **\( incs \)**
  - 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 \( isw = 1 \), \( s_i \leftarrow x_i \cdot y_i \)
    - 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 $i = 1, n$

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 70 on page 234.

$incxi$

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 $(1+(n-1)(incy)) + (m-1)\|incyi\|$, containing numbers of the data type indicated in Table 70 on page 234.

$incyi$

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.

$incy$

is the stride for $y$ in the outer loop—that is, the stride identifying each vector $y_i$ in $y$.

Specified as: an integer; $incy \geq 0$.

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 70 on page 234.

Function

The four possible computations that can be performed by these subprograms are:

**Four Possible Computations Performed by These Subroutines**

$s_i \leftarrow x_i \cdot y_i$  
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, \( inc_x \) and \( inc_y \), to locate the elements in vectors \( x \) and \( y \), respectively.
- The strides for the outer loops, \( incs \), \( inc_xo \), and \( inc_yo \), 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 VSX unit is not used.

### Error conditions

#### Computational Errors

None

#### Input-Argument Errors

1. \( n < 0 \)
2. \( m < 0 \)
3. \( incs = 0 \)
4. \( isw < 1 \) or \( isw > 4 \)
5. \( incxo < 0 \)
6. \( incyo < 0 \)

### Examples

#### Example 1

This example shows a store positive dot product computation using vectors with positive strides.

Call Statement and Input:

```
CALL SNDOT( 3 , 4 , S , 1 , 1 , X , 1 , 4 , Y , 1 , 4 )
```

\[
\begin{align*}
X & = \begin{bmatrix}
1.0 & 2.0 & 3.0 \\
2.0 & 3.0 & 4.0 \\
3.0 & 4.0 & 5.0 \\
4.0 & 5.0 & 6.0 \\
\end{bmatrix} \\
Y & = \begin{bmatrix}
4.0 & 3.0 & 2.0 \\
3.0 & 2.0 & 1.0 \\
2.0 & 1.0 & 4.0 \\
1.0 & 4.0 & 3.0 \\
\end{bmatrix}
\end{align*}
\]

Output:

\[
S = (20.0, 36.0, 48.0)
\]

#### Example 2

This example shows a store negative dot product computation using vectors with positive and negative strides.
Call Statement and Input:

\[
\begin{align*}
N & \quad M & \quad S & \quad \text{INCS} & \quad \text{ISW} & \quad X & \quad \text{INCXI} & \quad \text{INCXO} & \quad Y & \quad \text{INCYI} & \quad \text{INCYO} \\
& & & & & & & & & & \\
\text{CALL SNDOT(} & 3 & , & 4 & , & S & , & -1 & , & 2 & , & X & , & 2 & , & 10 & , & Y & , & -1 & , & 6 & )
\end{align*}
\]

\[
X = \begin{bmatrix}
1.0 & 2.0 & 3.0 \\
2.0 & 3.0 & 4.0 \\
\vdots & \vdots & \vdots \\
3.0 & 4.0 & 5.0 \\
4.0 & 5.0 & 6.0 \\
\vdots & \vdots & \vdots \\
\end{bmatrix}
\]

\[
Y = \begin{bmatrix}
4.0 & 3.0 & 2.0 \\
3.0 & 2.0 & 1.0 \\
2.0 & 1.0 & 4.0 \\
1.0 & 4.0 & 3.0 \\
\vdots & \vdots & \vdots \\
\end{bmatrix}
\]

Output:

\[
S = (-42.0, -34.0, -30.0)
\]

**Example 3**

This example shows an accumulative positive dot product using vectors with positive and negative strides.

Call Statement and Input:

\[
\begin{align*}
N & \quad M & \quad S & \quad \text{INCS} & \quad \text{ISW} & \quad X & \quad \text{INCXI} & \quad \text{INCXO} & \quad Y & \quad \text{INCYI} & \quad \text{INCYO} \\
& & & & & & & & & & \\
\text{CALL SNDOT(} & 3 & , & 4 & , & S & , & 1 & , & 3 & , & X & , & -2 & , & 10 & , & Y & , & 2 & , & 10 & )
\end{align*}
\]

\[
S = (2.0, 5.0, 8.0)
\]
Example 4

This example shows an accumulative negative dot product using vectors with positive and negative strides.

Call Statement and Input:

```
CALL SNDOT( 3 , 4 , S , -1  , 4 , X ,  1  ,  6  , Y ,  2  , 10  )
```

```
S        =  (3.0, 6.0, 9.0)
X    =  | 1.0  2.0  3.0 |
       | 2.0  3.0  4.0 |
       | 3.0  4.0  5.0 |
       | 4.0  5.0  6.0 |
Y    =  | 4.0  3.0  2.0 |
       | 3.0  2.0  1.0 |
       | 2.0  1.0  4.0 |
       | 1.0  4.0  3.0 |
       | .    .    . |
       | .    .    . |
       | .    .    . |
```

Output:

```
S        =  (-45.0, -30.0, -11.0)
```

SNRM2, DNRM2, SCNRM2, and DZNRM2 (Euclidean Length of a Vector with Scaling of Input to Avoid Destructive Underflow and Overflow)

Purpose

These subprograms compute the Euclidean length ($l_2$ norm) of vector $x$, with scaling of input to avoid destructive underflow and overflow.

Table 71: Data Types.

<table>
<thead>
<tr>
<th>Data Types</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 242 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 34.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>SNRM2</td>
</tr>
<tr>
<td>C and C++</td>
<td>snrm2</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_snrm2</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 \( n \), whose Euclidean length is to be computed. Specified as: a one-dimensional array of (at least) length \( 1+(n-1)|\text{incx}| \), containing numbers of the data type indicated in Table 71 on page 238.
- \( \text{incx} \) 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 71 on page 238.

Notes

Declare this function in your program as returning a value of the data type indicated in Table 71 on page 238.

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 [“95” on page 1258]. 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: \(3.37 \times 10^{-38} \) to \(3.37 \times 10^{38}\)
- For long-precision: \(1.67 \times 10^{-308} \) to \(1.67 \times 10^{308}\)

**Example 1**

This example shows a vector, \( x \), whose elements must be scaled to prevent overflow.

<table>
<thead>
<tr>
<th>N</th>
<th>X</th>
<th>INCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNORM = DNRM2( 6 , X , 1 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ X = (0.68056D+200, 0.25521D+200, 0.34028D+200, 0.85071D+200, 0.25521D+200, 0.85071D+200) \]

Output:

\[ DNORM = 0.1469D+201 \]

**Example 2**

This example shows a vector, \( x \), whose elements must be scaled to prevent destructive underflow.

Function Reference and Input:

<table>
<thead>
<tr>
<th>N</th>
<th>X</th>
<th>INCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNORM = DNRM2( 4 , X , 2 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ X = (0.10795D-200, \ , 0.10795D-200, \ , 0.10795D-200, \ , 0.10795D-200, \ , 0.10795D-200) \]

Output:

\[ DNORM = 0.21590D-200 \]

**Example 3**

This example shows a vector, \( x \), with a stride of 0. The result in \( SNORM \) is:

\[ \sqrt{\sum_{i=1}^{n} x_i^2} \]

Function Reference and Input:

<table>
<thead>
<tr>
<th>N</th>
<th>X</th>
<th>INCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNORM = SNRM2( 4 , X , 0 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ X = (4.0) \]
Example 4
This example shows a vector, \( x \), containing complex numbers, and whose elements must be scaled to prevent overflow.

Function Reference and Input:

\[
\begin{array}{c|c|c}
N & X & INCX \\
\hline
DZNORM & DZNRM2(3, X, 1) \\
\end{array}
\]

\[
X = ((0.68056D+200, 0.25521D+200), (0.34028D+200, 0.85071D+200), \\
(0.25521D+200, 0.85071D+200))
\]

Output:

\[
DZNORM = 0.1469D+201
\]

Example 5
This example shows a vector, \( x \), containing complex numbers, and whose elements must be scaled to prevent destructive underflow.

Function Reference and Input:

\[
\begin{array}{c|c|c}
N & X & INCX \\
\hline
DZNORM & DZNRM2(2, X, 2) \\
\end{array}
\]

\[
X = ((0.10795D-200, 0.10795D-200), . , \\
(0.10795D-200, 0.10795D-200))
\]

Output:

\[
DZNORM = 0.2159D-200
\]

**SNORM2, DNORM2, CNORM2, and ZNORM2 (Euclidean Length of a Vector with No Scaling of Input)**

**Purpose**
These subprograms compute the euclidean length (\( l_2 \) norm) of vector \( x \) with no scaling of input.

<table>
<thead>
<tr>
<th>Data Types</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 34.
Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>SNORM2</td>
</tr>
<tr>
<td>C and C++</td>
<td>snorm2</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, whose euclidean length is to be computed.
  - 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 on page 241.
- **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 72 on page 241.

**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 \( 0.10010\times10^{-145} < |x_i| < 0.13408\times10^{155} \) for \( i = 1, n \)
2. Declare this function in your program as returning a value of the data type indicated in Table 72 on page 241.

**Function**

The euclidean length (\( l_2 \) norm) of vector x is expressed as follows with no scaling of input:

\[
\sqrt{|x_1|^2 + |x_2|^2 + \ldots + |x_n|^2}
\]

See reference ["95" on page 1258]. 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 242.
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:

\[
\begin{align*}
N & \quad X & \quad \text{INCX} \\
\text{SNORM} & = \text{SNORM2}(6, X, 1) \\
X & = (3.0, 4.0, 1.0, 8.0, 1.0, 3.0)
\end{align*}
\]

Output:
SNORM = 10.0

Example 2
This example shows a vector, x, with a stride greater than 1.

Function Reference and Input:

\[
\begin{align*}
N & \quad X & \quad \text{INCX} \\
\text{SNORM} & = \text{SNORM2}(6, X, 2) \\
X & = (3.0, \ldots, 4.0, \ldots, 1.0, \ldots, 8.0, \ldots, 1.0, \ldots, 3.0)
\end{align*}
\]

Output:
SNORM = 10.0

Example 3
This example shows a vector, x, with a stride of 0. The result in SNORM is:

\[
\sqrt{\sum_{i=1}^{n} x_i^2}
\]

Function Reference and Input:

\[
\begin{align*}
N & \quad X & \quad \text{INCX} \\
\text{SNORM} & = \text{SNORM2}(4, X, 0) \\
X & = (4.0)
\end{align*}
\]

Output:
SNORM = 8.0

Example 4
This example shows a vector, x, containing complex numbers and having a stride of 1.

Function Reference and Input:

\[
\begin{align*}
N & \quad X & \quad \text{INCX} \\
\text{CNORM} & = \text{CNORM2}(3, X, 1) \\
X & = ((3.0, 4.0), (1.0, 8.0), (-1.0, 3.0))
\end{align*}
\]
Output:

CNORM = 10.0

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 73: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$, $b$, $r$, $s$</td>
<td>c</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 complex</td>
<td>Short-precision real</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SROTG</td>
</tr>
<tr>
<td>C and C++</td>
<td>srotg</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_srotg</td>
</tr>
</tbody>
</table>

On Entry

$a$
is the rotational elimination parameter $a$.
Specified as: a number of the data type indicated in Table 73 on page 244.

$b$
is the rotational elimination parameter $b$.
Specified as: a number of the data type indicated in Table 73 on page 244.

$c$
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 73 on page 244.

\( b \)
is the value computed for \( z \).

For SROTG and DROTG:
\[ z = s \text{ if } |a| > |b| \]
\[ z = 1/c \text{ if } |a| \leq |b| \text{ and } c \neq 0 \text{ and } r \neq 0 \]
\[ z = 1 \text{ if } |a| \leq |b| \text{ and } c = 0 \text{ and } r \neq 0 \]
\[ z = 0 \text{ if } r = 0 \]

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 73 on page 244.

\( c \)
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:
\[ c = \frac{|a|}{\sqrt{|a|^2 + |b|^2}} \quad \text{if } |a| \neq 0 \]
\[ c = 0 \quad \text{if } |a| = 0 \]

 Returned as: a number of the data type indicated in Table 73 on page 244.

\( s \)
is the sine \( s \) of the angle of (Givens) rotation.

For SROTG and DROTG:
For CROTG and ZROTG:

\[
s = \frac{\psi \overline{b}}{\sqrt{|a|^2 + |b|^2}} \quad \text{if } |a| \neq 0
\]

\[
s = (1.0, 0.0) \quad \text{if } |a| = 0
\]

where \( \psi = \frac{a}{|a|} \)

Returned as: a number of the data type indicated in Table 73 on page 244.

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:

\[
\sigma = \text{SIGN}(a) \quad \text{if } |a| > |b|
\]
\[
\sigma = \text{SIGN}(b) \quad \text{if } |a| \leq |b|
\]

\[
c = a/r \quad \text{if } r \neq 0
\]
\[
c = 1 \quad \text{if } r = 0
\]

\[
s = b/r \quad \text{if } r \neq 0
\]
\[
s = 0 \quad \text{if } r = 0
\]

\[
z = s \quad \text{if } |a| > |b|
\]
\[
z = 1/c \quad \text{if } |a| \leq |b| \text{ and } c \neq 0 \text{ and } r \neq 0
\]
\[
z = 1 \quad \text{if } |a| \leq |b| \text{ and } c = 0 \text{ and } r \neq 0
\]
\[
z = 0 \quad \text{if } r = 0
\]

See reference [“95” on page 1258].

Following are some important points about the computation:
1. The numbers for \( c, s, \) and \( r \) satisfy:
2. Where necessary, scaling is used to avoid overflow and destructive underflow in the computation of \( r \), which is expressed as follows:

\[
\begin{bmatrix}
  c & s \\
  -s & c
\end{bmatrix}
\begin{bmatrix}
  a \\
  b
\end{bmatrix} =
\begin{bmatrix}
  r \\
  0
\end{bmatrix}
\]

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 [“112” on page 1259]. \( c \) 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:

\[
r = \psi \sqrt{|a|^2 + |b|^2} \quad \text{if} \quad |a| \neq 0
\]

\[
r = b \quad \text{if} \quad |a| = 0
\]

where:

\[
\psi = a/|a|
\]

\[
c = \frac{|a|}{\sqrt{|a|^2 + |b|^2}} \quad \text{if} \quad |a| \neq 0
\]

\[
c = 0 \quad \text{if} \quad |a| = 0
\]
\[ s = \frac{\psi \bar{b}}{\sqrt{|a|^2 + |b|^2}} \quad \text{if } |a| \neq 0 \]

\[ s = (1.0, 0.0) \quad \text{if } |a| = 0 \]

See reference [“95” on page 1258].

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:

\[
r = \psi \left( |a| + |b| \right) \sqrt{ \left| \frac{a}{|a| + |b|} \right|^2 + \left| \frac{b}{|a| + |b|} \right|^2 } \]

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:

\[
\begin{array}{cccc}
A & B & C & S \\
| & | & | & | \\
\end{array}
\]

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:

\[
\begin{array}{cccc}
A & B & C & S \\
\hline \\
\text{CALL SROTG}( & 0.0 & , 2.0 & , C & , S ) \\
\end{array}
\]

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:

\[
\begin{array}{cccc}
A & B & C & S \\
\hline \\
\text{CALL SROTG}( & 6.0 & , -8.0 & , C & , S ) \\
\end{array}
\]

Output:

\[
\begin{array}{c}
A = -10.0 \\
B = -1.666 \\
C = -0.6 \\
S = 0.8 \\
\end{array}
\]

Example 4
This example shows the construction of a real Givens plane rotation, where \(|a| > |b|\).

Call Statement and Input:

\[
\begin{array}{cccc}
A & B & C & S \\
\hline \\
\text{CALL SROTG}( & 8.0 & , 6.0 & , C & , S ) \\
\end{array}
\]

Output:

\[
\begin{array}{c}
A = 10.0 \\
B = 0.6 \\
C = 0.8 \\
S = 0.6 \\
\end{array}
\]

Example 5
This example shows the construction of a complex Givens plane rotation, where \(|a| = 0\).

Call Statement and Input:

\[
\begin{array}{cccc}
A & B & C & S \\
\hline \\
\text{CALL CROTG}( & A & , B & , C & , S ) \\
\end{array}
\]

Output:

\[
\begin{array}{c}
A = (0.0, 0.0) \\
B = (1.0, 0.0) \\
\end{array}
\]

Example 6
This example shows the construction of a complex Givens plane rotation, where \(|a| \neq 0\).
Call Statement and Input:

A   B   C   S
|   |   |   |
CALL CROTG( A , B , C , S )
A = (3.0, 4.0)
B = (4.0, 6.0)

Output:
A = (5.26, 7.02)
C = 0.57
S = (0.82, -0.05)

SROT, DROT, CROT, ZROT, CSROT, and ZDROT (Apply a Plane Rotation)

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 74: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SROT</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DROT</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CROT</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZROT</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CSROT</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZDROT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Types</th>
<th>c</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>x, y</td>
<td>Short-precision real</td>
<td>Short-precision real</td>
</tr>
<tr>
<td></td>
<td>Long-precision real</td>
<td>Long-precision real</td>
</tr>
<tr>
<td></td>
<td>Short-precision real</td>
<td>Short-precision complex</td>
</tr>
<tr>
<td></td>
<td>Long-precision real</td>
<td>Long-precision complex</td>
</tr>
<tr>
<td></td>
<td>Short-precision real</td>
<td>Short-precision real</td>
</tr>
<tr>
<td></td>
<td>Long-precision real</td>
<td>Long-precision real</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 34.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SROT</td>
</tr>
<tr>
<td>C and C++</td>
<td>srot</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_srot</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 \( x_i \) 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 74 on page 250.

**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 $y_i$ 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 74 on page 250.

**incy**

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 74 on page 250.

$s$

the sine, $s$, of the angle of rotation.

Specified as: a number of the data type indicated in Table 74 on page 250.

**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 74 on page 250.

$y$

is the vector $y$ of length $n$, containing the rotated $y_i$ coordinates, where:

For SROT, DROT, CSROT, and ZDROT:

$$y_i \leftarrow -sx_i + cy_i \quad \text{for } i = 1, n$$

For CROT and ZROT:

$$y_i \leftarrow -sx_i + cy_i \quad \text{for } i = 1, n$$

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 74 on page 250.

**Notes**

The vectors $x$ and $y$ must have no common elements; otherwise, results are unpredictable. See “Vector concepts” on page 71.

**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:
For CROT and ZROT:

\[
\begin{bmatrix}
  x_i \\
  y_i
\end{bmatrix}
\leftarrow
\begin{bmatrix}
  c & s \\
  -s & c
\end{bmatrix}
\begin{bmatrix}
  x_i \\
  y_i
\end{bmatrix}
\quad \text{for } i = 1, n
\]

For CROT and ZROT:

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

See references [“70” on page 1257] and [“95” on page 1258]. 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 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:

```plaintext
CALL SROT( 5 , X , 1 , Y , 2 , 0.5 , S )
```

\( X = (1.0, 2.0, 3.0, 4.0, 5.0) \)
\( Y = (-1.0, . , -2.0, . , -3.0, . , -4.0, . , -5.0) \)

\( S = \frac{\sqrt{3.0}}{2.0} \)

Output:

\( X = (-0.366, -0.732, -1.098, -1.464, -1.830) \)
\( Y = (-1.366, -2.732, -4.098, -5.464, -6.830) \)

Example 2
This example shows how to apply a real plane rotation to real vectors \( x \) and \( y \) having strides of opposite sign.

Call Statement and Input:

```plaintext
CALL SROT( 5 , X , 1 , Y , -1 , 0.5 , S )
```

\( X = (1.0, 2.0, 3.0, 4.0, 5.0) \)
\( Y = (-5.0, -4.0, -3.0, -2.0, -1.0) \)
\[ S = \frac{\sqrt{3.0}}{2.0} \]

Output:
\[ X = \text{(same as output } X \text{ in Example 1)} \]
\[ Y = (-6.830, -5.464, -4.098, -2.732, -1.366) \]

**Example 3**
This example shows how scalar values in vectors \( x \) and \( y \) can be processed by specifying 0 strides and the number of elements to be processed, \( n \), equal to 1.

Call Statement and Input:

<table>
<thead>
<tr>
<th>N</th>
<th>X</th>
<th>INCX</th>
<th>Y</th>
<th>INCY</th>
<th>C</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( X )</td>
<td>0</td>
<td>( Y )</td>
<td>0</td>
<td>0.5</td>
<td>S</td>
</tr>
</tbody>
</table>

\[ \text{CALL SROT( 1 , X , 0 , Y , 0 , 0.5 , S )} \]
\[ X = (1.0) \]
\[ Y = (-1.0) \]

\[ S = \frac{\sqrt{3.0}}{2.0} \]

Output:
\[ X = (-0.366) \]
\[ Y = (-1.366) \]

**Example 4**
This example shows how to apply a complex plane rotation to complex vectors \( x \) and \( y \) having positive strides.

Call Statement and Input:

<table>
<thead>
<tr>
<th>N</th>
<th>X</th>
<th>INCX</th>
<th>Y</th>
<th>INCY</th>
<th>C</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>( X )</td>
<td>1</td>
<td>( Y )</td>
<td>2</td>
<td>0.5</td>
<td>S</td>
</tr>
</tbody>
</table>

\[ \text{CALL CROT( 3 , X , 1 , Y , 2 , 0.5 , S )} \]
\[ X = ((1.0, 2.0), (2.0, 3.0), (3.0, 4.0)) \]
\[ Y = ((-1.0, 5.0), . , (-2.0, 4.0), . , (-3.0, 3.0)) \]
\[ S = (0.75, 0.50) \]

Output:
\[ X = ((-2.750, 4.250), (-2.500, 3.500), (-2.250, 2.750)) \]
\[ Y = ((-2.250, 1.500), . , (-4.000, 0.750), . , (-5.750, 0.000)) \]

**Example 5**
This example shows how to apply a real plane rotation to complex vectors \( x \) and \( y \) having positive strides.

Call Statement and Input:

<table>
<thead>
<tr>
<th>N</th>
<th>X</th>
<th>INCX</th>
<th>Y</th>
<th>INCY</th>
<th>C</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>( X )</td>
<td>1</td>
<td>( Y )</td>
<td>2</td>
<td>0.5</td>
<td>S</td>
</tr>
</tbody>
</table>

\[ \text{CALL CSROT( 3 , X , 1 , Y , 2 , 0.5 , S )} \]
\[ X = ((1.0, 2.0), (2.0, 3.0), (3.0, 4.0)) \]
\[ Y = ((-1.0, 5.0), . , (-2.0, 4.0), . , (-3.0, 3.0)) \]

Linear algebra subprograms 253
\[ S = \frac{\sqrt{3.0}}{2.0} \]

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:

\[
\begin{vmatrix}
\sqrt{d_1} x_1 \\
\sqrt{d_2} x_2
\end{vmatrix}
\]

**Figure 11: 2-Vector**

\( H \) has one of the following forms:

<table>
<thead>
<tr>
<th>PARAM</th>
<th>( H )</th>
</tr>
</thead>
</table>
| \(-1.0\) | \[
\begin{bmatrix}
h_{11} & h_{12} \\
h_{21} & h_{22}
\end{bmatrix}
\] |
| \(0.0\) | \[
\begin{bmatrix}
1.0 & h_{12} \\
h_{21} & 1.0
\end{bmatrix}
\] |
| \(1.0\) | \[
\begin{bmatrix}
h_{11} & 1.0 \\
-1.0 & h_{22}
\end{bmatrix}
\] |
| \(-2.0\) | \[
\begin{bmatrix}
1.0 & 0.0 \\
0.0 & 1.0
\end{bmatrix}
\] |

**Figure 12: The form of matrix \( H \)**

SROTM and DROTM uses the information in PARAM to apply the modified Givens transformation.

**Table 75: 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 76: Syntax for supported computing languages.**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SROTMG</td>
</tr>
<tr>
<td>C and C++</td>
<td>srotmg</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_srotmg</td>
</tr>
</tbody>
</table>

**On Entry**

- **d1**
  - is the value \( d_1 \) shown in **Figure 11 on page 254**.
  - Specified as: a number of the data type indicated in Table 75 on page 254.

- **d2**
  - is the value \( d_2 \) shown in **Figure 11 on page 254**.
  - Specified as: a number of the data type indicated in Table 75 on page 254.

- **x1**
  - is the value \( x_1 \) shown in **Figure 11 on page 254**.
  - Specified as: a number of the data type indicated in Table 75 on page 254.

- **x2**
  - is the value \( x_2 \) shown in **Figure 11 on page 254**.
  - Specified as: a number of the data type indicated in Table 75 on page 254.

**On Return**

- **d1**
  - \( d_1 \) is overwritten; that is, the original input is not preserved.
  - Returned as: a number of the data type indicated in Table 75 on page 254.

- **d2**
  - \( d_2 \) is overwritten; that is, the original input is not preserved.
  - Returned as: a number of the data type indicated in Table 75 on page 254.

- **x1**
  - \( x_1 \) is overwritten; that is, the original input is not preserved.
  - Returned as: a number of the data type indicated in Table 75 on page 254.

- **x2**
  - \( x_2 \) is overwritten; that is, the original input is not preserved.
  - Returned as: a number of the data type indicated in Table 75 on page 254.

**param**

- **param** is a vector of length 5 defining the form and element values of the modified Givens transformation matrix \( H \), where:
  - \( param_1 = \text{Form of } H \), either -1.0, 0.0, 1.0, or -2.0, as shown in **Figure 12 on page 254**.
  - \( param_2 = H_{11} \)
  - \( param_3 = H_{21} \)
  - \( param_4 = H_{12} \)
  - \( param_5 = H_{22} \)

The matrix \( H \) values of 1.0, -1.0, 0.0 implied when \( param_1 = 0, 1, -2 \) are assumed and are not stored in \( param_{2:5} \).

SROTM and DROTM use the information in **param** to apply the modified Givens transformation.
Returned as: a vector of length 5.

Notes:
1. In your C program, arguments \(d1\), \(d2\), \(x1\), and \(x2\) must be passed by reference.
2. In your C++ program, for cblas_srotmg and cblas_drotmg, arguments \(d1\), \(d2\), \(x1\), and \(x2\) 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:

\[
\begin{bmatrix}
\sqrt{d_1} x_1 \\
\sqrt{d_2} x_2
\end{bmatrix}
\]

\(H\) has one of the following forms:

- \(1.0 \ H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \)
- \(0.0 \ H = \begin{bmatrix} 1.0 & h_{12} \\ h_{21} & 1.0 \end{bmatrix} \)
- \(-1.0 \ H = \begin{bmatrix} -1.0 & h_{12} \\ h_{21} & -1.0 \end{bmatrix} \)
- \(-2.0 \ H = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix} \)

SROTM and DROTM uses the information in PARAM to apply the modified Givens transformation.

For more information, see references [“95” on page 1258].

Error conditions

Computational Errors
None

Input-Argument Errors
None

Examples

See the combined DROTMG and DROTM example shown in “Example 1” on page 259.

SROTM and DROTM (Apply a modified Givens Transformation)

Purpose

SROTM and DROTM apply a modified GIVENS matrix transformation. For \(I=1, n\):
Where $H$ is a modified Givens transformation returned by a preceding call to SROTMG or DROTMG respectively.

For $I = 1,n$:

\[
\begin{align*}
  x_i &= h_{11}x_i + h_{12}y_i \\
y_i &= h_{21}x_i + h_{22}y_i
\end{align*}
\]

### Table 77: 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

### Table 78: Syntax for supported computing languages.

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SROTM</td>
</tr>
<tr>
<td>C and C++</td>
<td>srotm</td>
</tr>
<tr>
<td>CBLAS</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 $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 77 on page 257.

$incx$ 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 $1+(n-1)|incy|$, containing numbers of the data type indicated in Table 77 on page 257.

$incy$ is the stride for vector $y$. 
Specified as: an integer. It can have any value.

*param*

*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:

\[
\begin{align*}
\text{param}_1 &= \text{Form of } H, \text{ either } -1.0, 0.0, 1.0, \text{ or } -2.0, \text{ as shown in Figure 12 on page 254}. \\
\text{param}_2 &= H_{11} \\
\text{param}_3 &= H_{21} \\
\text{param}_4 &= H_{12} \\
\text{param}_5 &= H_{22}
\end{align*}
\]

The matrix *H* values of 1.0, -1.0, 0.0 implied when *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 77 on page 257.

*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 77 on page 257.

**Notes:**

- The vectors *x* and *y* must have no common elements; otherwise, results are unpredictable. See “Vector concepts” on page 71.

- 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*:

\[
\begin{bmatrix}
x_i \\
y_i
\end{bmatrix}
\begin{bmatrix}
- & - & - & H & - & - \\
\end{bmatrix}
\begin{bmatrix}
x_i \\
y_i
\end{bmatrix}
\]

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 [“95” on page 1258].

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:**

```
D1   D2   X1   X2   PARAM
|    |    |    |     |
CALL DROTMG( D1, D2, X1, X2, PARAM)
```

\[D1 = 1000000000.0\]
\[D2 = 1.0\]
\[X1 = -2.0\]
\[X2 = 4.0\]

**Output:**

\[PARAM = (-1.0, 4096.0, 2.0, -0.8192D-5, 1.0)\]

On output, D1, D2, X1, and X2 are overwritten.

**Call Statement and Input:**

```
N   X INCX   Y INCY   PARAM
|   |   |   |     |
CALL DROT( 5, X, 1, Y, 1, PARAM)
```

\[X = (1.0, 2.0, 3.0, 4.0, 5.0)\]
\[Y = (-1.0, -2.0, -3.0, -4.0, -5.0)\]

\[PARAM = \text{output from the call to DROTMG}\]

**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 \(x\):
Table 79: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>α Short-precision real</td>
<td>SSCAL</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DSCAL</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CSCAL</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZSCAL</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>CSSCAL</td>
</tr>
<tr>
<td>Long-precision real</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 34.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SSCAL</td>
</tr>
<tr>
<td>C and C++</td>
<td>sscal</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_sscal</td>
</tr>
</tbody>
</table>

On Entry

\[ n \]

is the number of elements in vector \( x \). Specified as: an integer; \( n \geq 0 \).

\[ \text{alpha} \]

is the scalar \( \alpha \).

Specified as: a number of the data type indicated in Table 79 on page 260.

\[ x \]

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 on page 260.

\[ \text{incx} \]

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 79 on page 260.

Notes

The fastest way in ESSL to zero out contiguous (stride 1) arrays is to call SSCAL or DSCAL, specifying \( \text{incx} = 1 \) and \( \text{alpha} = 0 \).
Function
The computation is expressed as follows:

\[
\begin{bmatrix}
  x_1 \\
  \vdots \\
  \vdots \\
  x_n \\
\end{bmatrix} \leftarrow \alpha \begin{bmatrix}
  x_1 \\
  \vdots \\
  \vdots \\
  x_n \\
\end{bmatrix}
\]

See reference [“95” on page 1258]. If \( n = 0 \), no computation is performed. For CSCAL, intermediate results are accumulated in long precision when the 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:

```
N  ALPHA  X  INCX
|    |    |   |
CALL SSCAL( 5 , 2.0 , X , 1  )
X       = (1.0, 2.0, 3.0, 4.0, 5.0)
```

Output:

```
X       = (2.0, 4.0, 6.0, 8.0, 10.0)
```

Example 2
This example shows vector, \( x \), with a stride greater than 1.

Call Statement and Input:

```
N  ALPHA  X  INCX
|    |    |   |
CALL SSCAL( 5 , 2.0 , X , 2  )
X       = (1.0, . , 2.0, . , 3.0, . , 4.0, . , 5.0)
```

Output:

```
X       = (2.0, . , 4.0, . , 6.0, . , 8.0, . , 10.0)
```

Example 3
This example illustrates that when the strides for two similar computations (Example 1 and Example 3) have the same absolute value but have opposite signs, the output is the same. This example is the same as Example 1, except the stride for \( x \) is negative (-1). For performance reasons, it is better to specify the positive stride. For \( x \), processing begins at element \( X(5) \), which is 5.0, and results are stored beginning at the same element.
Call Statement and Input:

```
N  ALPHA  X   INCX
CALL SSCAL( 5 , 2.0 , X , -1  )
X        =  (1.0, 2.0, 3.0, 4.0, 5.0)
```

Output:

```
X        =  (2.0, 4.0, 6.0, 8.0, 10.0)
```

**Example 4**

This example shows how SSCAL can be used to compute a scalar value. In this case, input vector \( x \) contains a scalar value, and the stride is 0. The number of elements to be processed, \( n \), is 1.

Call Statement and Input:

```
N  ALPHA  X  INCX
CALL SSCAL( 1 , 2.0 , X , 0  )
X        =  (1.0)
```

Output:

```
X        =  (2.0)
```

**Example 5**

This example shows a scalar, \( \alpha \), and a vector, \( x \), containing complex numbers, where vector \( x \) has a stride of 1.

Call Statement and Input:

```
N  ALPHA  X  INCX
CALL CSCAL( 3 ,ALPHA, X , 1  )
ALPHA    =  (2.0, 3.0)
X        =  ((1.0, 2.0), (2.0, 0.0), (3.0, 5.0))
```

Output:

```
X        =  ((-4.0, 7.0), (4.0, 6.0), (-9.0, 19.0))
```

**Example 6**

This example shows a scalar, \( \alpha \), containing a real number, and a vector, \( x \), containing complex numbers, where vector \( x \) has a stride of 1.

Call Statement and Input:

```
N  ALPHA  X  INCX
CALL CSSCAL( 3 , 2.0 , X , 1  )
X        =  ((1.0, 2.0), (2.0, 0.0), (3.0, 5.0))
```

Output:

```
X        =  ((2.0, 4.0), (4.0, 0.0), (6.0, 10.0))
```

**SSWAP, DSWAP, CSWAP, and ZSWAP (Interchange the Elements of Two Vectors)**

**Purpose**

These subprograms interchange the elements of vectors \( x \) and \( y \).
\[ y \leftrightarrow x \]

### Table 80: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x, y )</td>
<td></td>
</tr>
<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 34.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>( \text{CALL SSWAP}</td>
</tr>
<tr>
<td>C and C++</td>
<td>( \text{sswap}</td>
</tr>
<tr>
<td>CBLAS</td>
<td>( \text{cblas_sswap}</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 80 on page 263.

- \( \text{incx} \) 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 80 on page 263.

- \( \text{incy} \) 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 80 on page 263.

- \( y \) 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 80 on page 263.
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 “Vector concepts” on page 71.

Function

The elements of vectors \( x \) and \( y \) are interchanged as follows:

\[
\begin{bmatrix}
  y_1 \\
  \vdots \\
  \vdots \\
  y_n \\
\end{bmatrix}
\leftrightarrow
\begin{bmatrix}
  x_1 \\
  \vdots \\
  \vdots \\
  x_n \\
\end{bmatrix}
\]

See reference ["95" on page 1258]. 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:

```fortran
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:

```fortran
CALL SSWAP( 5 , X , 1  , Y , -1  )
```

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

\( Y = (-1.0, -2.0, -3.0, -4.0, -5.0) \)
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:

\[
\begin{array}{c|c|c|c|c|c}
N & X & INCX & Y & INCY \\
\hline
1 & X & 0 & Y & 0 \\
\hline
\end{array}
\]

\[
\begin{align*}
X &= (1.0) \\
Y &= (-4.0)
\end{align*}
\]

Output:

\[
\begin{align*}
X &= (-4.0) \\
Y &= (1.0)
\end{align*}
\]

Example 4
This example shows vectors \( x \) and \( y \), containing complex numbers and having positive strides.

Call Statement and Input:

\[
\begin{array}{c|c|c|c|c|c|c}
N & X & INCX & Y & INCY \\
\hline
4 & X & 1 & Y & 2 \\
\hline
\end{array}
\]

\[
\begin{align*}
X &= ((1.0, 6.0), (2.0, 7.0), (3.0, 8.0), (4.0, 9.0)) \\
Y &= ((-1.0, -1.0), (2.0, -2.0), (3.0, -3.0), (4.0, -4.0))
\end{align*}
\]

Output:

\[
\begin{align*}
X &= ((-1.0, -1.0), (2.0, -2.0), (3.0, -3.0), (4.0, -4.0)) \\
Y &= ((1.0, 6.0), (2.0, 7.0), (3.0, 8.0), (4.0, 9.0))
\end{align*}
\]

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 \( z \):

\[ z \leftarrow x + y \]

**Table 81: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x, y, z )</td>
<td><strong>SVEA</strong></td>
</tr>
<tr>
<td>Short-precision real</td>
<td><strong>SVEA</strong></td>
</tr>
<tr>
<td>Long-precision real</td>
<td><strong>DVEA</strong></td>
</tr>
<tr>
<td>Short-precision complex</td>
<td><strong>CVEA</strong></td>
</tr>
<tr>
<td>Long-precision complex</td>
<td><strong>ZVEA</strong></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 34.
## Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SVEA</td>
</tr>
<tr>
<td>C and C++</td>
<td>svea</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_svea</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 \(1+(n-1)|\text{incx}|\), containing numbers of the data type indicated in Table 81 on page 265.

- **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 81 on page 265.

- **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. Returned as: a one-dimensional array of (at least) length \(1+(n-1)|\text{incz}|\), containing numbers of the data type indicated in Table 81 on page 265.

### 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 “Vector concepts” on page 71.

### Function

The computation is expressed as follows:
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
N   X  INCX  Y  INCY  Z  INCZ
|   |   |    |   |    |   |
CALL SVEA( 5 , X , 1  , Y , 2  , Z , 1  )
```

$x = (1.0, 2.0, 3.0, 4.0, 5.0)$

$y = (1.0, . , 1.0, . , 1.0, . , 1.0, . , 1.0)$

Output:

$z = (2.0, . , 4.0, . , 6.0, . , 8.0, . , 10.0)$

**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:

```plaintext
N   X  INCX  Y   INCY  Z  INCZ
|   |   |    |    |    |   |
CALL SVEA( 5 , X , 1  , Y , -1  , Z , 2  )
```

$x = (1.0, 2.0, 3.0, 4.0, 5.0)$

$y = (5.0, 4.0, 3.0, 2.0, 1.0)$

Output:

$z = (2.0, . , 4.0, . , 6.0, . , 8.0, . , 10.0)$

**Example 3**

This example shows a vector, $x$, with 0 stride and a vector, $z$, with negative stride. $x$ is treated like a vector of length $n$, all of whose elements are the same as the single element in $x$. For vector $z$, results are stored beginning in element $Z(5)$.

Call Statement and Input:

```plaintext
N   X  INCX  Y  INCY  Z  INCZ
|   |   |    |   |    |   |
```
CALL SVEA( 5 , X , 0 , Y , 1 , Z , -1 )

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

Output:
Z = (2.0, 3.0, 4.0, 5.0, 6.0)

Example 4

This example shows a vector, y, with 0 stride. y is treated like a vector of length n, all of whose elements are the same as the single element in y.

Call Statement and Input:

CALL SVEA( 5 , X , 1 , Y , 0 , Z , 1 )

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

Output:
Z = (6.0, 7.0, 8.0, 9.0, 10.0)

Example 5

This example shows the output vector, z, with 0 stride, where the vector x has positive stride, and the vector y has 0 stride. The number of elements to be processed, n, is greater than 1.

Call Statement and Input:

CALL SVEA( 5 , X , 1 , Y , 0 , Z , 0 )

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

Output:
Z = (10.0)

Example 6

This example shows the output vector z, with 0 stride, where the vector x has 0 stride, and the vector y has negative stride. The number of elements to be processed, n, is greater than 1.

Call Statement and Input:

CALL SVEA( 5 , X , 0 , Y , -1 , Z , 0 )

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

Output:
Z = (6.0)

Example 7

This example shows how SVEA can be used to compute a scalar value. In this case, vectors x and y contain scalar values. The strides of all vectors, x, y, and z, are 0. The number of elements to be processed, n, is 1.
Call Statement and Input:

```
  N   X  INCX  Y  INCY  Z  INCZ
  CALL SVEA( 1 , X , 0 , Y , 0 , Z , 0 )
  X   =  (1.0)
  Y   =  (5.0)
```

Output:
```
  Z   =  (6.0)
```

**Example 8**

This example shows vectors \( \mathbf{x} \) and \( \mathbf{y} \), containing complex numbers and having positive strides.

Call Statement and Input:

```
  N   X  INCX  Y  INCY  Z  INCZ
  CALL CVEA( 3 , X , 1 , Y , 2 , Z , 1 )
  X   =  ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
  Y   =  ((7.0, 8.0), (9.0, 10.0), (11.0, 12.0))
```

Output:
```
  Z   =  ((8.0, 10.0), (12.0, 14.0), (16.0, 18.0))
```

**SVES, DVES, CVES, and ZVES (Subtract a Vector Y from a Vector X and Store in a Vector Z)**

**Purpose**

These subprograms perform the following computation, using vectors \( \mathbf{x}, \mathbf{y}, \) and \( \mathbf{z} \):

\[
\mathbf{z} \leftarrow \mathbf{x} - \mathbf{y}
\]

**Table 82: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathbf{x}, \mathbf{y}, \mathbf{z} )</td>
<td>SVES</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>DVES</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>CVES</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>ZVES</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 34.

**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 \( 1 + (n-1) |\text{incx}| \), containing numbers of the data type indicated in Table 82 on page 269.

\( \text{incx} \)

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 82 on page 269.

\( \text{incy} \)

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 \( z \) of length \( n \), containing the result of the computation. Returned as: a one-dimensional array of (at least) length \( 1 + (n-1) |\text{incz}| \), containing numbers of the data type indicated in Table 82 on page 269.

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 “Vector concepts” on page 71.

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:

```
N  X  INCX  Y  INCY  Z  INCZ
CALL SVES( 5 , X , 1  , Y , 2  , Z , 1  )
```

$x = (1.0, 2.0, 3.0, 4.0, 5.0)$

$y = (1.0, . , 1.0, . , 1.0, . , 1.0, . , 1.0)$

Output:

$z = (0.0, 1.0, 2.0, 3.0, 4.0)$

Example 2
This example shows vectors $x$ and $y$ having strides of opposite sign, and an output vector $z$ having a positive stride. For $y$, which has negative stride, processing begins at element $Y(5)$, which is 1.0.

Call Statement and Input:

```
N  X  INCX  Y  INCY  Z  INCZ
CALL SVES( 5 , X , 1  , Y , -1  , Z , 2  )
```

$x = (1.0, 2.0, 3.0, 4.0, 5.0)$

$y = (5.0, 4.0, 3.0, 2.0, 1.0)$

Output:

$z = (0.0, . , 0.0, . , 0.0, . , 0.0, . , 0.0)$

Example 3
This example shows a vector, $x$, with 0 stride, and a vector, $z$, with negative stride. $x$ is treated like a vector of length $n$, all of whose elements are the same as the single element in $x$. For vector $z$, results are stored beginning in element $Z(5)$.

Call Statement and Input:

```
N  X  INCX  Y  INCY  Z  INCZ
CALL SVES( 5 , X , 0  , Y , 1  , Z , -1  )
```

$x = (1.0)$

$y = (5.0, 4.0, 3.0, 2.0, 1.0)$

Output:

$z = (0.0, -1.0, -2.0, -3.0, -4.0)$

Example 4
This example shows a vector, $y$, with 0 stride. $y$ is treated like a vector of length $n$, all of whose elements are the same as the single element in $y$. 

Linear algebra subprograms 271
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.

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.
Example 8

This example shows vectors $x$ and $y$, containing complex numbers and having positive strides.

Call Statement and Input:

```
N   X  INCX  Y  INCY  Z  INCZ
CALL CVES( 3 , X , 1  , Y , 2  , Z , 1 )
```

$X = ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))$

$Y = ((7.0, 8.0), . , (9.0, 10.0), . , (11.0, 12.0))$

Output:

$Z = ((-6.0, -6.0), (-6.0, -6.0), (-6.0, -6.0))$

SVEM, DVEM, CVEM, and ZVEM (Multiply a Vector X by a Vector Y and Store in a Vector Z)

Purpose

These subprograms perform the following computation, using vectors $x$, $y$, and $z$:

$z \leftarrow xy$

Table 83: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$, $y$, $z$</td>
<td></td>
</tr>
<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 34.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SVEM</td>
</tr>
<tr>
<td>C and C++</td>
<td><code>svem</code></td>
</tr>
<tr>
<td>CBLAS</td>
<td><code>cblas_svem</code></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 $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 83 on page 273.

$incx$

is the stride for vector $x$. 

Linear algebra subprograms 273
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 \( 1 + (n-1) |incy| \), containing numbers of the data type indicated in Table 83 on page 273.

\( incy \)
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. Returned as: a one-dimensional array of (at least) length \( 1 + (n-1) |incz| \), containing numbers of the data type indicated in Table 83 on page 273.

**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 “Vector concepts” on page 71.

**Function**

The computation is expressed as follows:

\[ z_i \leftarrow x_i y_i \quad \text{for} \quad i = 1, n \]

If \( n \) is 0, no computation is performed. For CVEM, intermediate results are accumulated in long precision when the 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:

```
N   X  INCX   Y  INCY   Z  INCZ
|   |    |    |    |    |    |
CALL  SVEM( 5 , X , 1  , Y , 2  , Z , 1  )
```

\( X = (1.0, 2.0, 3.0, 4.0, 5.0) \)
\( Y = (1.0, . , 1.0, . , 1.0, . , 1.0, . , 1.0) \)
Output:

\[ Z = (1.0, 2.0, 3.0, 4.0, 5.0) \]

**Example 2**

This example shows vectors \( x \) and \( y \) having strides of opposite sign, and an output vector \( z \) having a positive stride. For \( y \), which has negative stride, processing begins at element \( Y(5) \), which is 1.0.

Call Statement and Input:

\[
\begin{array}{cccccc}
N & X & INCX & Y & INCY & Z & INCZ \\
\mid & \mid & \mid & \mid & \mid & \mid & \mid \\
\text{CALL SVEM( 5, X, 1, Y, -1, Z, 2 )}
\end{array}
\]

\[
\begin{array}{l}
X = (1.0, 2.0, 3.0, 4.0, 5.0) \\
Y = (5.0, 4.0, 3.0, 2.0, 1.0)
\end{array}
\]

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:

\[
\begin{array}{cccccc}
N & X & INCX & Y & INCY & Z & INCZ \\
\mid & \mid & \mid & \mid & \mid & \mid & \mid \\
\text{CALL SVEM( 5, X, 0, Y, 1, Z, -1 )}
\end{array}
\]

\[
\begin{array}{l}
X = (1.0) \\
Y = (5.0, 4.0, 3.0, 2.0, 1.0)
\end{array}
\]

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:

\[
\begin{array}{cccccc}
N & X & INCX & Y & INCY & Z & INCZ \\
\mid & \mid & \mid & \mid & \mid & \mid & \mid \\
\text{CALL SVEM( 5, X, 1, Y, 0, Z, 1 )}
\end{array}
\]

\[
\begin{array}{l}
X = (1.0, 2.0, 3.0, 4.0, 5.0) \\
Y = (5.0)
\end{array}
\]

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:

\[
\begin{array}{cccccc}
N & X & INCX & Y & INCY & Z & INCZ \\
\mid & \mid & \mid & \mid & \mid & \mid & \mid \\
\text{CALL SVEM( 5, X, 1, Y, 0, Z, 0 )}
\end{array}
\]

\[
\begin{array}{l}
X = (1.0, 2.0, 3.0, 4.0, 5.0) \\
Y = (5.0)
\end{array}
\]

Output:

\[ Z = (5.0, 10.0, 15.0, 20.0, 25.0) \]
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:

\[
\begin{array}{ccccccc}
N & X & INCX & Y & INCY & Z & INCZ \\
\mid & \mid & \mid & \mid & \mid & \mid & \mid \\
\text{CALL SVEM(} & 5 & , X , 0 , & Y , -1 , & Z , 0 )
\end{array}
\]

\[
\begin{array}{ccc}
X & = & (1.0) \\
Y & = & (5.0, 4.0, 3.0, 2.0, 1.0)
\end{array}
\]

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:

\[
\begin{array}{ccccccc}
N & X & INCX & Y & INCY & Z & INCZ \\
\mid & \mid & \mid & \mid & \mid & \mid & \mid \\
\text{CALL SVEM(} & 1 & , X , 0 , & Y , 0 , & Z , 0 )
\end{array}
\]

\[
\begin{array}{ccc}
X & = & (1.0) \\
Y & = & (5.0)
\end{array}
\]

Output:

\[
Z = (5.0)
\]

Example 8
This example shows vectors \( x \) and \( y \), containing complex numbers and having positive strides.

Call Statement and Input:

\[
\begin{array}{ccccccc}
N & X & INCX & Y & INCY & Z & INCZ \\
\mid & \mid & \mid & \mid & \mid & \mid & \mid \\
\text{CALL CVEM(} & 3 & , X , 1 , & Y , 2 , & Z , 1 )
\end{array}
\]

\[
\begin{array}{ccc}
X & = & ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0)) \\
Y & = & ((7.0, 8.0), \ldots, (9.0, 10.0), \ldots, (11.0, 12.0))
\end{array}
\]

Output:

\[
Z = ((-9.0, 22.0), (-13.0, 66.0), (-17.0, 126.0))
\]

SYAX, DYAX, CYAX, ZYAX, CSYAX, and ZDYAX (Multiply a Vector X by a Scalar and Store in a Vector Y)

Purpose
These subprograms perform the following computation, using the scalar \( \alpha \) and vectors \( x \) and \( y \):

\[ y \leftarrow \alpha x \]
Table 84: Data Types.

<table>
<thead>
<tr>
<th>Data Types</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 34.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SYAX</td>
</tr>
<tr>
<td>C and C++</td>
<td>syax</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 84 on page 277.

- **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 84 on page 277.

- **incx**
  - is the stride for vector x.
  - Specified as: an integer. It can have any value.

- **y**
  - See On Return.

- **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, 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 84 on page 277.
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 “Vector concepts” on page 71.

Function
The computation is expressed as follows:

\[
\begin{bmatrix}
  y_1 \\
  \cdot \\
  \cdot \\
  \cdot \\
  y_n \\
\end{bmatrix}
\leftarrow \alpha \begin{bmatrix}
  x_1 \\
  \cdot \\
  \cdot \\
  \cdot \\
  x_n \\
\end{bmatrix}
\]

See reference [“95” on page 1258]. If \( n = 0 \), no computation is performed. For CYAX, intermediate results are accumulated in long precision when the 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:

\[
\begin{array}{l}
N \quad \text{ALPHA} \quad X \quad \text{INCX} \quad Y \quad \text{INCY} \\
\text{CALL SYAX( 5 , 2.0 , X , 1 , Y , 2 )} \\
X \quad = \quad (1.0 , 2.0 , 3.0 , 4.0 , 5.0) \\
\end{array}
\]

Output:

\[
Y \quad = \quad (2.0 , . , 4.0 , . , 6.0 , . , 8.0 , . , 10.0) \\
\]

Example 2
This example shows vectors \( x \) and \( y \) that have strides of opposite signs. For \( y \), which has negative stride, results are stored beginning in element \( Y(5) \).

Call Statement and Input:

\[
\begin{array}{l}
N \quad \text{ALPHA} \quad X \quad \text{INCX} \quad Y \quad \text{INCY} \\
\text{CALL SYAX( 5 , 2.0 , X , 1 , Y , -1 )} \\
X \quad = \quad (1.0 , 2.0 , 3.0 , 4.0 , 5.0) \\
\end{array}
\]

Output:

\[
Y \quad = \quad (10.0 , 8.0 , 6.0 , 4.0 , 2.0) \\
\]
**Example 3**
This example shows a vector, \( \mathbf{x} \), with 0 stride. \( \mathbf{x} \) is treated like a vector of length \( n \), all of whose elements are the same as the single element in \( \mathbf{x} \).

Call Statement and Input:

```
 N  ALPHA  X  INCX  Y  INCY
CALL SYAX( 5 , 2.0 , X , 0  , Y , 1  )
X     = (1.0)
```

Output:
```
Y     = (2.0, 2.0, 2.0, 2.0, 2.0)
```

**Example 4**
This example shows how SYAX can be used to compute a scalar value. In this case both vectors \( \mathbf{x} \) and \( \mathbf{y} \) contain scalar values, and the strides for both vectors are 0. The number of elements to be processed, \( n \), is 1.

Call Statement and Input:

```
 N  ALPHA  X  INCX  Y  INCY
CALL SYAX( 1 , 2.0 , X , 0  , Y , 0  )
X     = (1.0)
```

Output:
```
Y     = (2.0)
```

**Example 5**
This example shows a scalar, \( \alpha \), and vectors \( \mathbf{x} \) and \( \mathbf{y} \), containing complex numbers, where both vectors have a stride of 1.

Call Statement and Input:

```
 N  ALPHA  X  INCX  Y  INCY
CALL CYAX( 3 ,ALPHA, X , 1  , Y , 1  )
ALPHA = (2.0, 3.0)
X     = ((1.0, 2.0), (2.0, 0.0), (3.0, 5.0))
```

Output:
```
Y     = ((-4.0, 7.0), (4.0, 6.0), (-9.0, 19.0))
```

**Example 6**
This example shows a scalar, \( \alpha \), containing a real number, and vectors \( \mathbf{x} \) and \( \mathbf{y} \), containing complex numbers, where both vectors have a stride of 1.

Call Statement and Input:

```
 N  ALPHA  X  INCX  Y  INCY
CALL CSYAX( 3 , 2.0 , X , 1  , Y , 1  )
X     = ((1.0, 2.0), (2.0, 0.0), (3.0, 5.0))
```

Output:
```
Y     = ((2.0, 4.0), (4.0, 0.0), (6.0, 10.0))
```
SZAXPY, DZAXPY, CZAXPY, and ZZAXPY (Multiply a Vector X by a Scalar, Add to a Vector Y, and Store in a Vector Z)

Purpose
These subprograms perform the following computation, using the scalar $\alpha$ and vectors $x$, $y$, and $z$:

$$z \leftarrow y + \alpha x$$

### Table 85: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha, x, y, z$</td>
<td>SZAXPY</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>DZAXPY</td>
</tr>
<tr>
<td>Long-precision real</td>
<td></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 34.

### Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SZAXPY</td>
</tr>
<tr>
<td>C and C++</td>
<td>szaxpy</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_szaxpy</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 85 on page 280.

$x$
- 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 85 on page 280.

$incx$
- 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 $1+(n-1)|incy|$, containing numbers of the data type indicated in Table 85 on page 280.

$incy$
- is the stride for vector $y$. 

280 IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and Reference
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 \( z \) of length \( n \), containing the result of the computation \( y + \alpha x \). Returned as: a one-dimensional array of (at least) length \( 1+(n-1)|\text{incz}| \), containing numbers of the data type indicated in Table 85 on page 280.

**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 “Vector concepts” on page 71.

**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 [“95” on page 1258]. If \( n \) is 0, no computation is performed. For CZAXPY, intermediate results are accumulated in long precision when the 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:

```fortran
  CALL SZAXPY( 5 , 2.0 , X , 1  , Y , 2  , Z , 1  )
```

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

\( Y = (1.0, . , 1.0, . , 1.0, . , 1.0, . , 1.0) \)
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:

\[
\begin{array}{cccccccc}
N & ALPHA & X & INCX & Y & INCY & Z & INCZ \\
5 & 2.0 & X & 1 & Y & -1 & Z & 2 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
X & = & (1.0, 2.0, 3.0, 4.0, 5.0) \\
Y & = & (5.0, 4.0, 3.0, 2.0, 1.0) \\
\end{array}
\]

Output:

\[
Z = (3.0, . , 6.0, . , 9.0, . , 12.0, . , 15.0)
\]

Example 3
This example shows a vector, \( x \), with 0 stride, and a vector, \( z \), with negative stride. \( x \) is treated like a vector of length \( n \), all of whose elements are the same as the single element in \( x \). For vector \( z \), results are stored beginning in element \( Z(5) \).

Call Statement and Input:

\[
\begin{array}{cccccccc}
N & ALPHA & X & INCX & Y & INCY & Z & INCZ \\
5 & 2.0 & X & 0 & Y & 1 & Z & -1 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
X & = & (1.0) \\
Y & = & (5.0, 4.0, 3.0, 2.0, 1.0) \\
Z & = & (3.0, 4.0, 5.0, 6.0, 7.0) \\
\end{array}
\]

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:

\[
\begin{array}{cccccccc}
N & ALPHA & X & INCX & Y & INCY & Z & INCZ \\
5 & 2.0 & X & 1 & Y & 0 & Z & 1 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
X & = & (1.0, 2.0, 3.0, 4.0, 5.0) \\
Y & = & (5.0) \\
\end{array}
\]

Output:

\[
Z = (7.0, 9.0, 11.0, 13.0, 15.0)
\]

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:

\[
\begin{array}{cccccccc}
N & ALPHA & X & INCX & Y & INCY & Z & INCZ \\
1 & 2.0 & X & 0 & Y & 0 & Z & 0 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
X & = & (1.0) \\
Y & = & (5.0) \\
\end{array}
\]
Example 6
This example shows vectors \(x\) and \(y\), containing complex numbers and having positive strides.

Call Statement and Input:

\[
\begin{array}{cccccccc}
N & \text{ALPHA} & \text{X} & \text{INCX} & \text{Y} & \text{INCY} & \text{Z} & \text{INCZ} \\
3 & (2.0, 3.0) & \left( (1.0, 2.0), (2.0, 0.0), (3.0, 5.0) \right) & \left( (1.0, 1.0), \ldots, (0.0, 2.0), \ldots, (5.0, 4.0) \right) \\
\end{array}
\]

Output:

\[
\begin{array}{c}
\text{Z} = \left( (-3.0, 8.0), (4.0, 8.0), (-4.0, 23.0) \right)
\end{array}
\]

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 86: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x, y)</td>
<td></td>
</tr>
<tr>
<td>Short-precision real</td>
<td>SSCTR</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DSCTR</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CSCTR</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZSCTR</td>
</tr>
</tbody>
</table>

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SSCTR</td>
</tr>
<tr>
<td>C and C++</td>
<td>ssctr</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 86 on page 283.

**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.

**y**
See On Return.

**On Return**

**y**
is the sparse vector \( y \), stored in full-vector storage mode, of (at least) length \( \max(\text{INDX}(i)) \) for \( i = 1, nz \), into which \( nz \) elements of vector \( x \) are copied at positions indicated by the indices array \( \text{INDX} \).

Returned 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 86 on page 283.

**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 “Vector concepts” on page 71.
3. For a description of how sparse vectors are stored, see “Sparse Vector” on page 75.

**Function**
The copy is expressed as follows:

\[
y_{\text{INDX}(i)} \leftarrow x_i \quad \text{for} \quad i = 1, nz
\]

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 [“39” on page 1255]. 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 \( \text{INDX} \) are in ascending order:

\[
Y = (6.0, 2.0, 4.0, 7.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0)
\]
Example 2
This example shows how to use SSCTR to copy a sparse vector \( \mathbf{x} \) of length 5 into the following vector \( \mathbf{y} \), where the elements of array \( \text{INDX} \) are in random order:

\[
\mathbf{y} = (6.0, 2.0, 4.0, 7.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0)
\]

Call Statement and Input:

\[
\begin{align*}
\text{NZ} & \quad \mathbf{X} & \quad \text{INDX} & \quad \mathbf{Y} \\
& \quad (5) & \quad (1.0, 2.0, 3.0, 4.0, 5.0) & \quad (1, 3, 4, 7, 10) \\
\text{CALL} & \quad \text{SSCTR} & \quad (5, \mathbf{X}, \text{INDX}, \mathbf{Y})
\end{align*}
\]

Output:

\[
\mathbf{y} = (1.0, 2.0, 2.0, 3.0, 6.0, 10.0, 4.0, 8.0, 9.0, 5.0)
\]

Example 3
This example shows how to use CSCTR to copy a sparse vector \( \mathbf{x} \) of length 3 into the following vector \( \mathbf{y} \), where the elements of array \( \text{INDX} \) are in random order:

\[
\mathbf{y} = ((6.0, 5.0), (-2.0, 3.0), (15.0, 4.0), (9.0, 0.0))
\]

Call Statement and Input:

\[
\begin{align*}
\text{NZ} & \quad \mathbf{X} & \quad \text{INDX} & \quad \mathbf{Y} \\
& \quad (3) & \quad ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0)) & \quad (4, 1, 3) \\
\text{CALL} & \quad \text{CSCTR} & \quad (3, \mathbf{X}, \text{INDX}, \mathbf{Y})
\end{align*}
\]

Output:

\[
\mathbf{y} = ((3.0, 4.0), (-2.0, 3.0), (5.0, 6.0), (1.0, 2.0))
\]

SGTHR, DGTHR, CGTHR, and ZGTHR (Gather Specified Elements of a Sparse Vector \( \mathbf{Y} \) in Full-Vector Storage Mode into a Sparse Vector \( \mathbf{X} \) in Compressed-Vector Storage Mode)

**Purpose**
These subprograms gather specified elements of vector \( \mathbf{y} \), stored in full-vector storage mode, into sparse vector \( \mathbf{x} \), stored in compressed-vector storage mode.

<table>
<thead>
<tr>
<th>Table 87: Data Types.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Types</strong></td>
</tr>
<tr>
<td>( \mathbf{x}, \mathbf{y} )</td>
</tr>
<tr>
<td>Data Types</td>
</tr>
<tr>
<td>---------------------------</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>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGTHR</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgthr</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 87 on page 285.

**$x$**

See On Return.

**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 87 on page 285.

### Notes

1. Vectors $x$ and $y$ must have no common elements; otherwise, results are unpredictable. See “Vector concepts” on page 71.
2. For a description of how sparse vectors are stored, see “Sparse Vector” on page 75.

### Function

The copy is expressed as follows:

$$x_i \leftarrow y_{\text{INDX}(i)} \quad \text{for } i = 1, nz$$

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 [“39” on page 1255]. 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:

\[
\begin{array}{llll}
\text{NZ} & y & x & \text{INDX} \\
\hline
5 & y & x & \text{INDX} \\
\end{array}
\]

CALL SGTHR( 5, y, x, \text{INDX} )

\[
\begin{array}{ll}
\text{Y} & = (6.0, 2.0, 4.0, 7.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0) \\
\text{INDX} & = (1, 3, 4, 7, 9) \\
\end{array}
\]

Output:

\[
\begin{array}{l}
x = (6.0, 4.0, 7.0, -2.0, 9.0) \\
\end{array}
\]

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:

\[
\begin{array}{llll}
\text{NZ} & y & x & \text{INDX} \\
\hline
5 & y & x & \text{INDX} \\
\end{array}
\]

CALL SGTHR( 5, y, x, \text{INDX} )

\[
\begin{array}{ll}
\text{Y} & = (6.0, 2.0, 4.0, 7.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0) \\
\text{INDX} & = (4, 3, 1, 10, 7) \\
\end{array}
\]

Output:

\[
\begin{array}{l}
x = (7.0, 4.0, 6.0, 0.0, -2.0) \\
\end{array}
\]

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 \( \text{INDX} \) are in random order.

Call Statement and Input:

\[
\begin{array}{llll}
\text{NZ} & y & x & \text{INDX} \\
\hline
3 & y & x & \text{INDX} \\
\end{array}
\]

CALL CGTHR( 3, y, x, \text{INDX} )

\[
\begin{array}{ll}
\text{Y} & = ((6.0, 5.0), (-2.0, 3.0), (15.0, 4.0), (9.0, 0.0)) \\
\text{INDX} & = (4, 1, 3) \\
\end{array}
\]

Output:

\[
\begin{array}{l}
x = ((9.0, 0.0), (6.0, 5.0), (15.0, 4.0)) \\
\end{array}
\]
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 88: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>x, y</td>
<td></td>
</tr>
<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>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGTHRZ</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgthrz</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 ≥ 0.

y

is the sparse vector y, stored in full-vector storage mode, of (at least) length max(INDX(i)) for i = 1, nz, from which nz elements are copied from positions indicated by the indices array INDX.

Specified as: a one-dimensional array of (at least) length max(INDX(i)) for i = 1, nz, containing numbers of the data type indicated in Table 88 on page 288.

x

See On Return.

indx

is the array, referred to as INDX, containing the nz indices that indicate the positions of the elements of the sparse vector x when in full-vector storage mode. They also indicate the positions in vector y from which elements are copied then set to zero.

Specified as: a one-dimensional array of (at least) length nz, containing integers.

On Return

y

is the sparse vector y, stored in full-vector storage mode, of (at least) length max(INDX(i)) for i = 1, nz, whose elements are set to zero at positions indicated by the indices array INDX.

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 88 on page 288.
\( \mathbf{x} \)
is the sparse vector \( \mathbf{x} \), containing \( nz \) elements stored in compressed-vector storage mode in an array, referred to as \( \mathbf{X} \), into which are copied the elements of vector \( \mathbf{y} \) 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 88 on page 288.

**Notes**

1. Each value specified in array \( \text{INDX} \) must be unique; otherwise, results are unpredictable.
2. Vectors \( \mathbf{x} \) and \( \mathbf{y} \) must have no common elements; otherwise, results are unpredictable. See “Vector concepts” on page 71.
3. For a description of how sparse vectors are stored, see “Sparse Vector” on page 75.

**Function**

The copy is expressed as follows:

\[
\begin{align*}
\mathbf{x}_i & \leftarrow \mathbf{y}(\text{INDX}(i)) \\
\mathbf{y}(\text{INDX}(i)) & \leftarrow 0.0 \quad \text{(for SGTHRZ and DGTHRZ)} \\
\mathbf{y}(\text{INDX}(i)) & \leftarrow (0.0,0.0) \quad \text{(for CGTHRZ and ZGTHRZ)}
\end{align*}
\]

where:

\( \mathbf{x} \) is a sparse vector, stored in compressed-vector storage mode.

\( \text{INDX} \) is the indices array for sparse vector \( \mathbf{x} \).

\( \mathbf{y} \) is a sparse vector, stored in full-vector storage mode.

See reference [“39” on page 1255]. 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 SGTHRZ to copy specified elements of a vector \( \mathbf{y} \) into a sparse vector \( \mathbf{x} \) of length 5, where the elements of array \( \text{INDX} \) are in ascending order.

Call Statement and Input:

\[
\begin{align*}
\text{NZ} & \quad \text{Y} & \quad \text{X} & \quad \text{INDX} \\
\text{CALL SGTHRZ( 5 , Y , X , INDX )}
\end{align*}
\]

\[
\begin{align*}
\text{Y} & = (6.0, 2.0, 4.0, 7.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0) \\
\text{INDX} & = (1, 3, 4, 7, 9)
\end{align*}
\]

Output:

\[
\begin{align*}
\text{Y} & = (0.0, 2.0, 0.0, 0.0, 6.0, 10.0, 0.0, 8.0, 0.0, 0.0) \\
\text{X} & = (6.0, 4.0, 7.0, -2.0, 9.0)
\end{align*}
\]

**Example 2**
This example shows how to use SGTHRZ to copy specified elements of a vector \( \mathbf{y} \) into a sparse vector \( \mathbf{x} \) of length 5, where the elements of array \( \text{INDX} \) are in random order. (Note that the element 0.0 occurs in output vector \( \mathbf{x} \). This does not produce an error.)
Example 3
This example shows how to use CGTHRZ to copy specified elements of a vector \( y \) into a sparse vector \( x \) of length 3, where the elements of array INDX are in random order.

Call Statement and Input:

\[
\begin{aligned}
\text{NZ Y X INDX} \\
\text{CALL CGTHRZ( 3 , Y , X , INDX )}
\end{aligned}
\]

\[
\begin{aligned}
Y &= (6.0, 5.0), (-2.0, 3.0), (15.0, 4.0), (9.0, 0.0) \\
\text{INDX} &= (4, 1, 3)
\end{aligned}
\]

Output:

\[
\begin{aligned}
Y &= ((0.0, 0.0), (-2.0, 3.0), (0.0, 0.0), (0.0, 0.0)) \\
X &= ((9.0, 0.0), (6.0, 5.0), (15.0, 4.0))
\end{aligned}
\]

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>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha, x, y )</td>
<td>SAXPYI</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>DAXPYI</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>CAXPYI</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>ZAXPYI</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZAXPYI</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SAXPYI</td>
</tr>
<tr>
<td>C and C++</td>
<td>saxpyi</td>
</tr>
</tbody>
</table>

On Entry

290 IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and Reference
\( nz \)

is the number of elements in sparse vector \( x \), stored in compressed-vector storage mode. Specified as: an integer; \( nz \geq 0 \).

\( \text{alpha} \)

is the scalar \( \alpha \). Specified as: a number of the data type indicated in Table 89 on page 290.

\( x \)

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 89 on page 290.

\( \text{indx} \)

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 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 \( \max(\text{INDX}(i)) \) for \( i = 1, nz \). 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 89 on page 290.

On Return

\( y \)

is the sparse vector \( y \), stored in full-vector storage mode, of (at least) length \( \max(\text{INDX}(i)) \) for \( i = 1, nz \) 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 89 on page 290.

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 “Vector concepts” on page 71.
3. For a description of how sparse vectors are stored, see “Sparse Vector” on page 75.

Function

The computation is expressed as follows:

\[ y_{\text{INDX}(i)}' = y_{\text{INDX}(i)} + \alpha x_i \quad \text{for } i = 1, nz \]

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 [“39” on page 1255]. If \( \alpha \) or \( nz \) is zero, no computation is performed. For SAXPYI and CAXPYI, intermediate results are accumulated in long-precision.

Error conditions

Computational Errors

None

Input-Argument Errors

\( nz < 0 \)
Examples

Example 1
This example shows how to use SAXPYI to perform a computation using a sparse vector \( x \) of length 5, where the elements of array \( \text{INDX} \) are in ascending order.

Call Statement and Input:

\begin{verbatim}
NZ ALPHA X INDX Y
CALL SAXPYI( 5 , 2.0 , X , INDX , Y )
X = (1.0, 2.0, 3.0, 4.0, 5.0)
INDX = (1, 3, 4, 7, 10)
Y = (1.0, 5.0, 4.0, 3.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0)
\end{verbatim}

Output:

\[ Y = (3.0, 5.0, 8.0, 9.0, 6.0, 10.0, 6.0, 8.0, 9.0, 10.0) \]

Example 2
This example shows how to use SAXPYI to perform a computation using a sparse vector \( x \) of length 5, where the elements of array \( \text{INDX} \) are in random order.

Call Statement and Input:

\begin{verbatim}
NZ ALPHA X INDX Y
CALL SAXPYI( 5 , 2.0 , X , INDX , Y )
X = (1.0, 2.0, 3.0, 4.0, 5.0)
INDX = (4, 3, 1, 10, 7)
Y = (1.0, 5.0, 4.0, 3.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0)
\end{verbatim}

Output:

\[ Y = (7.0, 5.0, 8.0, 9.0, 6.0, 10.0, 8.0, 9.0, 8.0, 8.0) \]

Example 3
This example shows how to use CAXPYI to perform a computation using a sparse vector \( x \) of length 3, where the elements of array \( \text{INDX} \) are in random order.

Call Statement and Input:

\begin{verbatim}
NZ ALPHA X INDX Y
CALL CAXPYI( 3 , ALPHA , X , INDX , Y )
ALPHA = (2.0, 3.0)
X = ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
INDX = (4, 1, 3)
Y = ((6.0, 5.0), (-2.0, 3.0), (15.0, 4.0), (9.0, 0.0))
\end{verbatim}

Output:

\[ Y = ((0.0, 22.0), (-2.0, 3.0), (7.0, 31.0), (5.0, 7.0)) \]

**SDOTI, DDOTI, CDOTUI, ZDOTUI, CDOTCI, and ZDOTCI (Dot Product of a Sparse Vector \( x \) in Compressed-Vector Storage Mode and a Sparse Vector \( y \) in Full-Vector Storage Mode)**

**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 90: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x, y$, Result</td>
<td></td>
</tr>
<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>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>SDOTI</td>
</tr>
<tr>
<td>C and C++</td>
<td>sdoti</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 90 on page 293.

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 $\text{max}(\text{INDX}(i))$ for $i = 1, nz$. Specified as: a one-dimensional array of (at least) length $\text{max}(\text{INDX}(i))$ for $i = 1, nz$, containing numbers of the data type indicated in Table 90 on page 293.

On Return

Function value

is the result of the dot product computation.

Returned as: a number of the data type indicated in Table 90 on page 293.

Notes

1. Declare this function in your program as returning a value of the data type indicated in Table 90 on page 293.

2. For a description of how sparse vectors are stored, see “Sparse Vector” on page 75.

Function

For SDOTI, DDOTI, CDOTUI, and ZDOTUI, the dot product computation is expressed as follows:
For CDOTCI and ZDOTCI, the dot product computation is expressed as follows:

\[
\sum_{i=1}^{nz} x_i y_{\text{INDX}(i)} = x_1 y_{\text{INDX}(1)} + x_2 y_{\text{INDX}(2)} + \ldots + x_{nz} y_{\text{INDX}(nz)}
\]

where:

- \( x \) is a sparse vector, stored in compressed-vector storage mode.
- \( y \) is a sparse vector, stored in full-vector storage mode.
- \( \bar{x} \) is the complex conjugate of a sparse vector, stored in compressed-vector storage mode.
- \( \text{INDX} \) is the indices array for sparse vector \( x \).
- See reference \[“39” on page 1255\]. 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:

\[
\text{NZ} \quad \text{X} \quad \text{INDX} \quad \text{Y} \\
\text{DOTT} = \text{SDOTI} ( 5 , \text{X} , \text{INDX} , \text{Y} )
\]

\[
\begin{align*}
\text{X} &= (1.0, 2.0, 3.0, 4.0, 5.0) \\
\text{INDX} &= (1, 3, 4, 7, 10) \\
\text{Y} &= (1.0, 5.0, 4.0, 3.0, 6.0, 10.0, -2.0, 8.0, 9.0, 0.0)
\end{align*}
\]

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.
**Example 3**
This example shows how to use CDOTUI to compute a dot product using a sparse vector \( x \) of length 3, where the elements of array INDX are in ascending order.

Function Reference and Input:

\[
\begin{align*}
\text{NZ} & \quad X & \quad \text{INDX} & \quad Y \\
\text{DOTT} & = \text{CDOTUI}( 3, X, \text{INDX}, Y ) \\
X & = ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0)) \\
\text{INDX} & = (1, 3, 4) \\
Y & = ((6.0, 5.0), (-2.0, 3.0), (15.0, 4.0), (9.0, 0.0))
\end{align*}
\]

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 INDX are in random order.

Function Reference and Input:

\[
\begin{align*}
\text{NZ} & \quad X & \quad \text{INDX} & \quad Y \\
\text{DOTT} & = \text{CDOTCI}( 3, X, \text{INDX}, Y ) \\
X & = ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0)) \\
\text{INDX} & = (4, 1, 3) \\
Y & = ((6.0, 5.0), (-2.0, 3.0), (15.0, 4.0), (9.0, 0.0))
\end{align*}
\]

Output:

\[
\text{DOTT} = (146.0, -97.0)
\]
\[ y \leftarrow \beta y + \alpha A^T x \]

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:

\[
\begin{align*}
y & \leftarrow \beta y + \alpha A x \\
y & \leftarrow \beta y + \alpha A^T x \\
y & \leftarrow \beta y + \alpha A^H x
\end{align*}
\]

SGEMX and DGEMX compute the matrix-vector product for a real general matrix, using the scalar \( \alpha \), vectors \( x \) and \( y \), and matrix \( A \):

\[ y \leftarrow \beta y + \alpha A x \]

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 \( A \):

\[ y \leftarrow \beta y + \alpha A^T x \]

### Table 91: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha, \beta, x, y, A )</td>
<td>( \text{SGEMV, SGEMX, and SGEMTX} )</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>( \text{SGEMV, SGEMX, and SGEMTX} )</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>( \text{DGEMV, DGEMX, and DGEMTX} )</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>( \text{CGEMV} )</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>( \text{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 34.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGEMV</td>
</tr>
<tr>
<td></td>
<td>CALL SGEMX</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgemv</td>
</tr>
<tr>
<td></td>
<td>sgemx</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_sgemv</td>
</tr>
</tbody>
</table>

**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 \textit{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.

\textit{transa}

indicates the form of matrix $A$ to use in the computation, where:

If $\textit{transa} = 'N'$, $A$ is used in the computation.

If $\textit{transa} = 'T'$, $A^T$ is used in the computation.

If $\textit{transa} = 'C'$, $A^H$ is used in the computation.

Specified as: a single character. It must be 'N', 'T', or 'C'.

\textit{cblas_transa}

indicates the form of matrix $A$ to use in the computation, where:

If \textit{cblas_transa} = CblasNoTrans, $A$ is used in the computation.

If \textit{cblas_transa} = CblasTrans, $A^T$ is used in the computation.

If \textit{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.

\textbf{m}

is the number of rows in matrix $A$, and:

For SGEMV, DGEMV, CGEMV, and ZGEMV:

If $\textit{transa} = 'N'$, it is the length of vector $y$.

If $\textit{transa} = 'T'$ 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; $0 \leq m \leq \text{lda}$.

\textbf{n}

is the number of columns in matrix $A$, and:

For SGEMV, DGEMV, CGEMV, and ZGEMV:

If $\textit{transa} = 'N'$, it is the length of vector $x$.

If $\textit{transa} = 'T'$ 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; $n \geq 0$.

\textbf{alpha}

is the scaling constant $\alpha$.

Specified as: a number of the data type indicated in \textit{Table 91 on page 296}.

\textbf{a}

is the $m$ by $n$ matrix $A$, where:

For SGEMV, DGEMV, CGEMV, and ZGEMV:

If $\textit{transa} = 'N'$, $A$ is used in the computation.

If $\textit{transa} = 'T'$, $A^T$ is used in the computation.

If $\textit{transa} = 'C'$, $A^H$ 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.

**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 91 on page 296.

$lda$

is the leading dimension of the array specified for $a$.

Specified as: an integer; $lda > 0$ and $lda \geq m$.

$x$

is the vector $x$, where:

For SGEMV, DGEMV, CGEMV, and ZGEMV:

If $transa = 'N'$, it has length $n$.
If $transa = 'T'$ or 'C', it has length $m$.

For SGEMX and DGEMX, it has length $n$.
For SGEMTX and DGEMTX, it has length $m$.

Specified as: a one-dimensional array, containing numbers of the data type indicated in Table 91 on page 296, where:

For SGEMV, DGEMV, CGEMV, and ZGEMV:

If $transa = 'N'$, it must have at least $1+(n-1)|incx|$ elements.
If $transa = 'T'$ or 'C', it must have at least $1+(m-1)|incx|$ elements.

For SGEMX and DGEMX, it must have at least $1+(n-1)|incx|$ elements.
For SGEMTX and DGEMTX, it must have at least $1+(m-1)|incx|$ elements.

$beta$

is the scaling constant $\beta$.

Specified as: a number of the data type indicated in Table 91 on page 296.

$incx$

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 91 on page 296, 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 $1+(n-1)|incy|$ 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 91 on page 296.

**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 ["44" on page 1255] and ["45" on page 1255].
4. Vector y must have no common elements with matrix A or vector x; otherwise, results are unpredictable. See "Vector concepts" on page 71.

**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 VSX unit is not used. Occasionally, for performance reasons, these intermediate results are stored.

See references ["44" on page 1255], ["45" on page 1255], ["48" on page 1255], ["56" on page 1256], and ["95" on page 1258]. No computation is performed if m or n is 0 or if α is zero and β is one.

**General Matrix**

For SGEMV, DGEMV, CGEMV, and ZGEMV, the matrix-vector product for a general matrix:

\[ y \leftarrow \beta y + \alpha Ax \]

is expressed as follows:

\[
\begin{bmatrix}
  y_1 \\
  . \\
  . \\
  y_m \\
\end{bmatrix}
\leftarrow 
\beta
\begin{bmatrix}
  y_1 \\
  . \\
  . \\
  y_m \\
\end{bmatrix}
+ 
\alpha
\begin{bmatrix}
  a_{11} & \cdots & a_{1n} \\
  . & . & . \\
  . & . & . \\
  a_{m1} & \cdots & a_{mn} \\
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  . \\
  . \\
  x_n \\
\end{bmatrix}
\]

For SGEMX and DGEMX, the matrix-vector product for a real general matrix:
\[ \mathbf{y} \leftarrow \mathbf{y} + \alpha \mathbf{A} \mathbf{x} \]

is expressed as follows:

\[
\begin{bmatrix}
  y_1 \\
  . \\
  . \\
  . \\
  y_m
\end{bmatrix}
\leftarrow
\begin{bmatrix}
  y_1 \\
  . \\
  . \\
  . \\
  y_m
\end{bmatrix}
+ \alpha
\begin{bmatrix}
  a_{11} & \cdots & a_{1n} \\
  . & \ddots & . \\
  . & . & . \\
  a_{m1} & \cdots & a_{mn}
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  . \\
  . \\
  . \\
  x_n
\end{bmatrix}
\]

In these expressions:

- \( \mathbf{y} \) is a vector of length \( m \).
- \( \alpha \) is a scalar.
- \( \beta \) is a scalar.
- \( \mathbf{A} \) is an \( m \times n \) matrix.
- \( \mathbf{x} \) 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:

\[ \mathbf{y} \leftarrow \beta \mathbf{y} + \alpha \mathbf{A}^T \mathbf{x} \]

is expressed as follows:

\[
\begin{bmatrix}
  y_1 \\
  . \\
  . \\
  . \\
  y_n
\end{bmatrix}
\leftarrow
\begin{bmatrix}
  y_1 \\
  . \\
  . \\
  . \\
  y_n
\end{bmatrix}
+ \alpha
\begin{bmatrix}
  a_{11} & \cdots & a_{m1} \\
  . & \ddots & . \\
  . & . & . \\
  a_{1n} & \cdots & a_{mn}
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  . \\
  . \\
  . \\
  x_m
\end{bmatrix}
\]

For SGEMTX and DGEMTX, the matrix-vector product for the transpose of a real general matrix:

\[ \mathbf{y} \leftarrow \mathbf{y} + \alpha \mathbf{A}^T \mathbf{x} \]

is expressed as follows:

\[
\begin{bmatrix}
  y_1 \\
  . \\
  . \\
  . \\
  y_n
\end{bmatrix}
\leftarrow
\begin{bmatrix}
  y_1 \\
  . \\
  . \\
  . \\
  y_n
\end{bmatrix}
+ \alpha
\begin{bmatrix}
  a_{11} & \cdots & a_{m1} \\
  . & \ddots & . \\
  . & . & . \\
  a_{1n} & \cdots & a_{mn}
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  . \\
  . \\
  . \\
  x_m
\end{bmatrix}
\]

In these expressions:

- \( \mathbf{y} \) is a vector of length \( n \).
- \( \alpha \) is a scalar.
- \( \beta \) is a scalar.
\( \mathbf{A}^T \) is the transpose of matrix \( \mathbf{A} \), where \( \mathbf{A} \) is an \( m \) by \( n \) matrix.

\( \mathbf{x} \) 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:

\[
\mathbf{y} \leftarrow \beta \mathbf{y} + \alpha \mathbf{A}^H \mathbf{x}
\]

is expressed as follows:

\[
\begin{bmatrix}
  y_1 \\
  \vdots \\
  y_n
\end{bmatrix}
\leftarrow \begin{bmatrix}
  \beta \\
  \vdots \\
  \beta
\end{bmatrix}
+ \begin{bmatrix}
  \alpha \\
  \vdots \\
  \alpha
\end{bmatrix}
\begin{bmatrix}
  \bar{a}_{11} & \cdots & \bar{a}_{m1} \\
  \vdots & \ddots & \vdots \\
  \bar{a}_{1n} & \cdots & \bar{a}_{mn}
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  \vdots \\
  x_m
\end{bmatrix}
\]

where:

- \( \mathbf{y} \) is a vector of length \( n \).
- \( \alpha \) is a scalar.
- \( \beta \) is a scalar.
- \( \mathbf{A}^H \) is the conjugate transpose of matrix \( \mathbf{A} \), where \( \mathbf{A} \) is an \( m \) by \( n \) matrix.
- \( \mathbf{x} \) is a vector of length \( m \).

**Error conditions**

**Resource Errors**

Unable to allocate internal work area.

**Computational Errors**

None

**Input-Argument Errors**

1. \texttt{cblas_layout} ≠ CblasRowMajor or CblasColMajor
2. \texttt{transa} ≠ 'N', 'T', or 'C'
3. \texttt{cblas_transa} ≠ CblasNoTrans, CblasTrans, or CblasConjTrans
4. \( m < 0 \)
5. \( m > \text{lda} \)
6. \( n < 0 \)
7. \( \text{lda} \leq 0 \)
8. \( \text{incy} = 0 \)

**Examples**

**Example 1**

This example shows the computation for \texttt{TRANSA} equal to 'N', where the real general matrix \( \mathbf{A} \) is used in the computation. Because \( \text{lda} = 10 \) and \( n = 3 \), array \( \mathbf{A} \) must be declared as \( \mathbf{A}(E1:E2,F1:F2) \), where \( E2-E1+1=10 \) and \( F2-F1+1 \geq 3 \). In this example, array \( \mathbf{A} \) is declared as \( \mathbf{A}(1:10,0:2) \).
Call Statement and Input:

TRANSA M N ALPHA A LDA X INCX BETA Y INCY
CALL SGEMV('N', 4, 3, 1.0, A(1,0), 10, X, 1, 1.0, Y, 2)

\[
A = \begin{bmatrix}
  1.0 & 2.0 & 3.0 \\
  2.0 & 2.0 & 4.0 \\
  3.0 & 2.0 & 2.0 \\
  4.0 & 2.0 & 1.0
\end{bmatrix}
\]

\[X = (3.0, 2.0, 1.0)\]
\[Y = (4.0, . , 5.0, . , 2.0, . , 3.0)\]

Output:
\[Y = (14.0, . , 19.0, . , 17.0, . , 20.0)\]

Example 2
This example shows the computation for \(\text{TRANSA}\) equal to 'T', where the transpose of the real general matrix \(A\) is used in the computation. Array \(A\) must follow the same rules as given in Example 1. In this example, array \(A\) is declared as \(A(-1:8,1:3)\).

Call Statement and Input:

TRANSA M N ALPHA A LDA X INCX BETA Y INCY
CALL SGEMV('T', 4, 3, 1.0, A(-1,1), 10, X, 1, 2.0, Y, 2)

\[A = \text{(same as input } A \text{ in Example 1)}\]
\[X = (3.0, 2.0, 1.0, 4.0)\]
\[Y = (1.0, . , 2.0, . , 3.0)\]

Output:
\[Y = (28.0, . , 24.0, . , 29.0)\]

Example 3
This example shows the computation for \(\text{TRANSA}\) 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)

\[\text{ALPHA} = (1.0, 0.0)\]
\[A = \begin{bmatrix}
  (1.0, 2.0) & (3.0, 5.0) & (2.0, 0.0) \\
  (2.0, 3.0) & (7.0, 9.0) & (4.0, 8.0) \\
  (7.0, 4.0) & (1.0, 4.0) & (6.0, 0.0) \\
  (8.0, 2.0) & (2.0, 5.0) & (8.0, 0.0) \\
  (9.0, 1.0) & (3.0, 6.0) & (1.0, 0.0)
\end{bmatrix}\]

\[\text{Output:}\]
\[Y = (1.0, . , 2.0, . , 3.0)\]
\[ X = ((1.0, 2.0), (4.0, 0.0), (1.0, 1.0)) \]
\[ \text{BETA} = (1.0, 0.0) \]
\[ Y = ((1.0, 2.0), (4.0, 0.0), (1.0, -1.0), (3.0, 4.0), (2.0, 0.0)) \]

Output:
\[ Y = ((12.0, 28.0), (24.0, 55.0), (10.0, 39.0), (23.0, 50.0), (22.0, 44.0)) \]

Example 4
This example shows the computation for \( \text{TRANS A} \) 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:
```
CALL CGEMV( 'T' , 5 , 3 , ALPHA , A , 10 , X , 1  , BETA , Y , 1  )
```

\[ \text{ALPHA} = (1.0, 0.0) \]
\[ A = \text{(same as input A in Example 3)} \]
\[ X = ((1.0, 2.0), (4.0, 0.0), (1.0, 1.0), (3.0, 4.0), (2.0, 0.0)) \]
\[ \text{BETA} = (0.0, 0.0) \]
\[ Y = \text{(not relevant)} \]

Output:
\[ Y = ((42.0, 67.0), (10.0, 87.0), (50.0, 74.0)) \]

Example 5
This example shows the computation for \( \text{TRANS A} \) equal to 'C', where the conjugate transpose of the complex general matrix \( A \) is used in the computation.

Call Statement and Input:
```
CALL CGEMV( 'C' , 5 , 3 , ALPHA , A , 10 , X , 1  , BETA , Y , 1  )
```

\[ \text{ALPHA} = (-1.0, 0.0) \]
\[ A = \text{(same as input A in Example 3)} \]
\[ X = ((1.0, 2.0), (4.0, 0.0), (1.0, 1.0), (3.0, 4.0), (2.0, 0.0)) \]
\[ \text{BETA} = (1.0, 0.0) \]
\[ Y = ((1.0, 2.0), (4.0, 0.0), (1.0, -1.0)) \]

Output:
\[ Y = ((-73.0, -13.0), (-74.0, 57.0), (-49.0, -11.0)) \]

Example 6
This example shows a matrix, \( A \), contained in a larger array, \( A \). The strides of vectors \( x \) and \( y \) are positive. Because \( lda \) is 10 and \( n \) is 3, array \( A \) must be declared as \( A(E1:E2, F1:F2) \), where \( E2-E1+1=10 \) and \( F2-F1+1 \geq 3 \). For this example, array \( A \) is declared as \( A(1:10, 0:2) \).
Call Statement and Input:

\[
\begin{array}{cccccccc}
M & N & \text{ALPHA} & A & \text{LDA} & X & \text{INCY} & Y \\
\hline
4 & 3 & 1.0 & A(1,0) & 10 & X & 1 & Y, 2 \\
\end{array}
\]

\[
\begin{array}{cccc}
1.0 & 2.0 & 3.0 \\
2.0 & 2.0 & 4.0 \\
3.0 & 2.0 & 2.0 \\
4.0 & 2.0 & 1.0 \\
\end{array}
\]

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

\[X = (3.0, 2.0, 1.0)\]
\[Y = (3.0, \text{.}, 2.0, \text{.}, 5.0, \text{.}, 4.0)\]

Output:
\[Y = (14.0, \text{.}, 19.0, \text{.}, 17.0, \text{.}, 20.0)\]

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:

\[
\begin{array}{cccccccc}
M & N & \text{ALPHA} & A & \text{LDA} & X & \text{INCY} & Y \\
\hline
4 & 3 & 1.0 & A(-1,1) & 10 & X & 1 & Y, -2 \\
\end{array}
\]

\[A = \begin{bmatrix}
1.0 & 2.0 & 3.0 \\
2.0 & 2.0 & 4.0 \\
3.0 & 2.0 & 2.0 \\
4.0 & 2.0 & 1.0 \\
\end{bmatrix}
\]

\[X = (3.0, 2.0, 1.0)\]
\[Y = (3.0, \text{.}, 2.0, \text{.}, 5.0, \text{.}, 4.0)\]

Output:
\[Y = (20.0, \text{.}, 17.0, \text{.}, 19.0, \text{.}, 14.0)\]

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:

\[
\begin{array}{cccccccc}
M & N & \text{ALPHA} & A & \text{LDA} & X & \text{INCY} & Y \\
\hline
4 & 3 & 1.0 & A(5,1) & 10 & X & 1 & Y, 1 \\
\end{array}
\]

\[
A = \begin{bmatrix}
1.0 & 2.0 & 3.0 \\
2.0 & 2.0 & 4.0 \\
3.0 & 2.0 & 2.0 \\
4.0 & 2.0 & 1.0 \\
\end{bmatrix}
\]
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:

```
M   N  ALPHA    A     LDA  X  INCX  Y  INCY
|   |    |      |      |   |   |    |   |
CALL SGEMX( 4 , 3 , 1.0 , A(1,0) , 4 , X , 1  , Y , 1  )
```

```
A = | 1.0  2.0  3.0 |
    | 2.0  2.0  4.0 |
    | 3.0  2.0  2.0 |
    | 4.0  2.0  1.0 |
```

$X = (3.0, 2.0, 1.0)$

$Y = (4.0, 5.0, 2.0, 3.0)$

Output:

$Y = (14.0, 19.0, 17.0, 20.0)$

Example 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:

```
M   N  ALPHA    A     LDA  X  INCX  Y  INCY
|   |    |      |      |   |   |    |   |
CALL SGEMTX( 4 , 3 , 1.0 , A(1,0) , 4 , X , 1  , Y , 1  )
```

```
A = | 1.0  2.0  3.0 |
    | 2.0  2.0  4.0 |
    | 3.0  2.0  2.0 |
    | 4.0  2.0  1.0 |
```

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

$Y = (1.0, 2.0, 3.0)$

Output:

$Y = (27.0, 22.0, 26.0)$

Example 11
This example shows a computation in which $alpha$ is greater than 1. Array $A$ must follow the same rules as given in Example 10. For this example, array $A$ is declared as $A(-1:2,1:3)$.

Call Statement and Input:

```
M   N  ALPHA     A     LDA  X  INCX  Y  INCY
|   |    |       |      |   |   |    |   |
CALL SGEMTX( 4 , 3 , 2.0 , A(-1,1) , 4 , X , 1  , Y , 1  )
```

Linear algebra subprograms 305
A = (same as input A in Example 10)
X = (3.0, 2.0, 1.0, 4.0)
Y = (1.0, 2.0, 3.0)

Output:

Y = (53.0, 42.0, 49.0)

**SGER, DGER, CGERU, ZGERU, CGERC, and ZGERC (Rank-One Update of a General Matrix)**

**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>
<thead>
<tr>
<th>Table 92: Data Types.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Types</strong></td>
</tr>
<tr>
<td>$\alpha, A, x, y$</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>
<tr>
<td><strong>Subprogram</strong></td>
</tr>
<tr>
<td>SGER</td>
</tr>
<tr>
<td>DGER</td>
</tr>
<tr>
<td>CGERU and CGERC</td>
</tr>
<tr>
<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 34.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGER</td>
</tr>
<tr>
<td>C and C++</td>
<td>sger</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_sger</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.

\( m \)

is the number of rows in matrix \( A \) and the number of elements in vector \( x \).
Specified as: an integer; \( 0 \leq m \leq lda \).

\( n \)

is the number of columns in matrix \( A \) and the number of elements in vector \( y \).
Specified as: an integer; \( n \geq 0 \).

\( \alpha \)

is the scaling constant \( \alpha \).
Specified as: a number of the data type indicated in Table 92 on page 306.

\( x \)

is the vector \( x \) of length \( m \).
Specified as: a one-dimensional array of (at least) length \( 1+(m-1)|incx| \), containing numbers of the data type indicated in Table 92 on page 306.

\( incx \)

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 \( y^\top \) 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 \( 1+(n-1)|incy| \), containing numbers of the data type indicated in Table 92 on page 306.

\( incy \)

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 \( lda \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 92 on page 306.

\( lda \)

is the size of the leading dimension of the array specified for \( a \).
Specified as: an integer; \( lda > 0 \) and \( lda \geq m \).

**On Return**

\( a \)

is the \( m \) by \( n \) matrix \( A \), containing the result of the computation.

Returned as: a two-dimensional array, containing numbers of the data type indicated in Table 92 on page 306.

**Notes**

1. In these subroutines, \( incx = 0 \) and \( incy = 0 \) are valid; however, the Level 2 BLAS standard considers \( incx = 0 \) and \( incy = 0 \) to be invalid. See references [“44” on page 1255] and [“45” on page 1255].
2. Matrix $A$ can have no common elements with vectors $x$ and $y$; otherwise, results are unpredictable. See “Vector concepts” on page 71.

**Function**

SGER, DGER, CGERU, and ZGERU compute the rank-one update of a general matrix:

$$A \leftarrow A + \alpha xy^T$$

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:

$$\begin{bmatrix}
  a_{11} & \ldots & a_{1n} \\
  \vdots & \ddots & \vdots \\
  a_{m1} & \ldots & a_{mn}
\end{bmatrix} \leftarrow \begin{bmatrix}
  a_{11} & \ldots & a_{1n} \\
  \vdots & \ddots & \vdots \\
  a_{m1} & \ldots & a_{mn}
\end{bmatrix} + \alpha \begin{bmatrix}
  x_1 \\
  \vdots \\
  x_m
\end{bmatrix} \begin{bmatrix}
  y_1 & \ldots & y_n
\end{bmatrix}$$

It can also be expressed as:

$$\begin{bmatrix}
  a_{11} & \ldots & a_{1n} \\
  \vdots & \ddots & \vdots \\
  a_{m1} & \ldots & a_{mn}
\end{bmatrix} \leftarrow \begin{bmatrix}
  a_{11} + \alpha x_1 y_1 & \ldots & a_{1n} + \alpha x_1 y_n \\
  \vdots & \ddots & \vdots \\
  a_{m1} + \alpha x_m y_1 & \ldots & a_{mn} + \alpha x_m y_n
\end{bmatrix}$$

CGERC and ZGERC compute a slightly different rank-one update of a general matrix:

$$A \leftarrow A + \alpha xy^H$$

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:
It can also be expressed as:

\[
\begin{bmatrix}
a_{11} & \ldots & a_{1n} \\
\vdots & \ddots & \vdots \\
a_{m1} & \ldots & a_{mn}
\end{bmatrix}
\begin{bmatrix}
\alpha \\
\vdots \\
\alpha
\end{bmatrix}
+ \alpha
\begin{bmatrix}
\bar{y}_1 & \ldots & \bar{y}_n
\end{bmatrix}
\]

See references [“44” on page 1255], [“45” on page 1255], and [“95” on page 1258]. No computation is performed if \( m, n \), or \( \alpha \) is zero. For CGERU and CGERC, intermediate results are accumulated in long precision when the VSX unit is not used. For SGER, intermediate results are accumulated in long precision on some platforms when the VSX unit is not used.

**Error conditions**

**Resource Errors:**
Unable to allocate internal work area.

**Computational Errors**
None

**Input-Argument Errors**

1. \( cblas_layout \neq \text{CblasRowMajor} \) or \( \text{CblasColMajor} \)
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:**

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|}
\hline
M & N & ALPHA & X & INCX & Y & INCY & A & LDA \\
\hline
4 & 3 & 1.0 & 1 & 1 & 2 & A(1,0) & 10 \\
\hline
\end{array}
\]

\[
X = (3.0, 2.0, 1.0, 4.0) \\
Y = (1.0, 2.0, 3.0)
\]

\[
\begin{bmatrix}
1.0 & 2.0 & 3.0 \\
2.0 & 2.0 & 4.0
\end{bmatrix}
\]
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:

```
CALL SGER( 4 , 3 , 1.0 , X , 1  , Y , -2  , A(-1,1) , 10 )
```

$x = (3.0, 2.0, 1.0, 4.0)$

$y = (3.0, . , 2.0, . , 1.0)$

Output:

$A = (same as input A in Example 1)$

Example 3
This example shows a matrix, $A$, contained in a larger array, $A$, and the first element of the matrix is not the first element of the array. Array $A$ must follow the same rules as given in Example 1. For this example, array $A$ is declared as $A(1:10,1:3)$.

Call Statement and Input:

```
CALL SGER( 4 , 3 , 1.0 , X , 3  , Y , 1  , A(4,1) , 10 )
```

$x = (3.0, . , . , 2.0, . , . , 1.0, . , . , 4.0)$

$y = (1.0, 2.0, 3.0)$

Output:

$A = (same as input A in Example 1)$
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:

\[
\begin{array}{cccccc}
M & N & \text{ALPHA} & X & \text{INCX} & Y & \text{INCY} & A & \text{LDA} \\
4 & 3 & 1.0 & X & 1 & Y & 1 & A(1,0) & 4
\end{array}
\]

\[
\begin{array}{c}
X = (3.0, 2.0, 1.0, 4.0) \\
Y = (1.0, 2.0, 3.0)
\end{array}
\]

\[
A = \begin{bmatrix}
1.0 & 2.0 & 3.0 \\
2.0 & 2.0 & 4.0 \\
3.0 & 2.0 & 2.0 \\
4.0 & 2.0 & 1.0
\end{bmatrix}
\]

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:

\[
\begin{array}{cccccc}
M & N & \text{ALPHA} & X & \text{INCX} & Y & \text{INCY} & A & \text{LDA} \\
4 & 3 & 2.0 & X & 1 & Y & 1 & A(-1,0) & 4
\end{array}
\]

\[
\begin{array}{c}
X = (3.0, 2.0, 1.0, 4.0) \\
Y = (1.0, 2.0, 3.0)
\end{array}
\]

\[
A = \begin{bmatrix}
1.0 & 2.0 & 3.0 \\
2.0 & 2.0 & 4.0 \\
3.0 & 2.0 & 2.0 \\
4.0 & 2.0 & 1.0
\end{bmatrix}
\]

Output:

\[
A = \begin{bmatrix}
4.0 & 8.0 & 12.0 \\
4.0 & 6.0 & 10.0 \\
4.0 & 4.0 & 5.0 \\
8.0 & 10.0 & 13.0
\end{bmatrix}
\]

\[
\begin{array}{cccccc}
M & N & \text{ALPHA} & X & \text{INCX} & Y & \text{INCY} & A & \text{LDA} \\
4 & 3 & 2.0 & X & 1 & Y & 1 & A(-1,1) & 4
\end{array}
\]

\[
\begin{array}{c}
X = (3.0, 2.0, 1.0, 4.0) \\
Y = (1.0, 2.0, 3.0)
\end{array}
\]

\[
A = \begin{bmatrix}
7.0 & 14.0 & 21.0 \\
6.0 & 10.0 & 16.0 \\
5.0 & 6.0 & 8.0 \\
12.0 & 18.0 & 25.0
\end{bmatrix}
\]
**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:**

```
M   N   ALPHA   X  INCX  Y  INCY    A     LDA
|   |     |     |   |    |   |      |      |
CALL CGERU( 5 , 3 , ALPHA , X , 1 , Y , 1 , A(1,0) , 10 )
```

**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:**

```
M   N   ALPHA   X  INCX  Y  INCY    A     LDA
|   |     |     |   |    |   |      |      |
CALL CGERC( 5 , 3 , ALPHA , X , 1 , Y , 1 , A(1,0) , 10 )
```

Output:

\[
\begin{bmatrix}
-2.0 & 6.0 & 7.0 & 13.0 & 5.0 & 1.0 \\
6.0 & 11.0 & 23.0 & 9.0 & 8.0 & 4.0 \\
6.0 & 7.0 & 5.0 & 8.0 & 8.0 & 0.0 \\
3.0 & 12.0 & 14.0 & 21.0 & 15.0 & 1.0 \\
11.0 & 5.0 & 11.0 & 6.0 & 3.0 & -2.0 \\
\end{bmatrix}
\]
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 $y$:

$$y \leftarrow y + \alpha Ax$$

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>
<thead>
<tr>
<th>Table 93: Data Types.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Types</td>
</tr>
<tr>
<td>$\alpha, \beta, A, x, y$</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
<tr>
<td>Short-precision complex</td>
</tr>
</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 34.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SSPMV</td>
</tr>
<tr>
<td></td>
<td>CALL SSYMV</td>
</tr>
<tr>
<td></td>
<td>CALL SSLMX</td>
</tr>
</tbody>
</table>
### Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>C and C++</th>
<th>CBLAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>sspmv</td>
<td>dspmv</td>
<td>chpmv (uplo, n, alpha, ap, x, incx, beta, y, incy);</td>
</tr>
<tr>
<td>ssymv</td>
<td>dsymv</td>
<td>chemv (uplo, n, alpha, a, lda, x, incx, beta, y, incy);</td>
</tr>
<tr>
<td>sslmx</td>
<td>dslmx (n, alpha, ap, x, incx, y, incy);</td>
<td></td>
</tr>
<tr>
<td>cblas_sspmv</td>
<td>cblas_dspmv</td>
<td>cblas_chpmv (cblas_layout, cblas_uplo, n, alpha, ap, x, incx, beta, y, incy);</td>
</tr>
<tr>
<td>cblas_ssymv</td>
<td>cblas_dsymv</td>
<td>cblas_chemv (cblas_layout, cblas_uplo, n, alpha, a, lda, x, incx, beta, y, incy);</td>
</tr>
<tr>
<td>cblas_slmx</td>
<td>cblas_dslmx (n, alpha, ap, x, incx, y, incy);</td>
<td></td>
</tr>
</tbody>
</table>

### 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 93 on page 313.

**ap**

has the following meaning:

- For SSWMV 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 93 on page 313.
\(a\) 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 93 on page 313.

\(lda\)

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) length \(1+(n-1)|incx|\), containing numbers of the data type indicated in Table 93 on page 313.

\(incx\)

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 \(β\).

Specified as: a number of the data type indicated in Table 93 on page 313.

\(y\)

is the vector \(y\) of length \(n\).

Specified as: a one-dimensional array of (at least) length \(1+(n-1)|incy|\), containing numbers of the data type indicated in Table 93 on page 313.

\(incy\)

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 93 on page 313.

**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 “Vector concepts” on page 71.
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 80. 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 84.
Function

These subroutines perform the calculations described. See references [“44” on page 1255], [“45” on page 1255], and [“95” on page 1258].

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 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:

\[
y := \beta y + \alpha A x
\]

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:

\[
\begin{bmatrix}
  y_1 \\
  \vdots \\
  y_n
\end{bmatrix}
\leftarrow \begin{bmatrix}
  \beta \\
  \vdots \\
  \beta
\end{bmatrix} + \alpha
\begin{bmatrix}
  a_{11} & \cdots & a_{1n} \\
  \vdots & \ddots & \vdots \\
  a_{n1} & \cdots & a_{nn}
\end{bmatrix}
\begin{bmatrix}
  x_1 \\
  \vdots \\
  x_n
\end{bmatrix}
\]

For SSLMX and DSLMX

These subroutines compute the matrix-vector product for a real symmetric matrix stored in lower-packed storage mode:

\[
y := y + \alpha A x
\]

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. \( \text{cblas\_layout} \neq \text{CblasRowMajor or CblasColMajor} \)
2. \( \text{uplo} \neq 'L' \text{ or 'U'} \)
3. \( \text{cblas\_uplo} \neq \text{CblasLower or CblasUpper} \)
4. \( n < 0 \)
5. \( \text{lda} < n \)
6. \( \text{lda} \leq 0 \)
7. \( \text{incx} = 0 \)
8. \( \text{incy} = 0 \)

Examples

Example 1
This example shows vectors \( \mathbf{x} \) and \( \mathbf{y} \) with positive strides and a real symmetric matrix \( \mathbf{A} \) of order 3, stored in lower-packed storage mode. Matrix \( \mathbf{A} \) 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:

\[
\text{CALL SSPMV( 'L' , 3 , 1.0 , AP , X , 1 , 1.0 , Y , 2 )}
\]

\[
\begin{align*}
\text{AP} & = \begin{pmatrix} 8.0, 4.0, 2.0, 6.0, 7.0, 3.0 \end{pmatrix} \\
\text{X} & = \begin{pmatrix} 3.0, 2.0, 1.0 \end{pmatrix} \\
\text{Y} & = \begin{pmatrix} 5.0, . , 3.0, . , 2.0 \end{pmatrix}
\end{align*}
\]

Output:

\[
\text{Y} = \begin{pmatrix} 39.0, . , 34.0, . , 25.0 \end{pmatrix}
\]

Example 2
This example shows vector \( \mathbf{x} \) and \( \mathbf{y} \) having strides of opposite signs. For \( \mathbf{x} \), which has negative stride, processing begins at element \( X(5) \), which is 1.0. The real symmetric matrix \( \mathbf{A} \) of order 3 is stored in upper-packed storage mode. It uses the same input matrix \( \mathbf{A} \) as in Example 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:

$$
\begin{bmatrix}
(1.0, 0.0) & (3.0, 5.0) & (2.0, -3.0) \\
(3.0, -5.0) & (7.0, 0.0) & (4.0, -8.0) \\
(2.0, 3.0) & (4.0, 8.0) & (6.0, 0.0)
\end{bmatrix}
$$

**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.

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.
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:

\[
\begin{align*}
\text{UPLO} & \quad N & \quad \alpha & \quad A & \quad \text{LDA} & \quad X & \quad \text{INCX} & \quad \beta & \quad Y & \quad \text{INCY} \\
\text{CALL SSYMV( 'L', 3 , 1.0 , A , 3 , X , 1 , 1.0 , Y , 2 )}
\end{align*}
\]

\[
A = \begin{bmatrix}
8.0 & . & . \\
4.0 & 6.0 & . \\
2.0 & 7.0 & 3.0
\end{bmatrix}
\]

\[
X = (3.0, 2.0, 1.0)
\]

\[
Y = (5.0, . , 3.0, . , 2.0)
\]

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:

\[
\begin{align*}
\text{UPLO} & \quad N & \quad \alpha & \quad A & \quad \text{LDA} & \quad X & \quad \text{INCX} & \quad \beta & \quad Y & \quad \text{INCY} \\
\text{CALL SSYMV( 'U', 3 , 1.0 , A , 4 , X , -2 , 2.0 , Y , 1 )}
\end{align*}
\]

\[
A = \begin{bmatrix}
8.0 & 4.0 & 2.0 \\
. & 6.0 & 7.0 \\
. & . & 3.0
\end{bmatrix}
\]

\[
X = (4.0, . , 2.0, . , 1.0)
\]

\[
Y = (6.0, 5.0, 4.0)
\]

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.
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.

Example 9
This example shows vectors \( x \) and \( y \) with positive strides and a real symmetric matrix \( A \) of order 3. Matrix \( A \) is:

\[
\begin{bmatrix}
8.0 & 4.0 & 2.0 \\
4.0 & 6.0 & 7.0 \\
2.0 & 7.0 & 3.0
\end{bmatrix}
\]
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 94: Data Types.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Types</strong></td>
</tr>
<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 34.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SSPR</td>
</tr>
<tr>
<td></td>
<td>CALL SSYR</td>
</tr>
<tr>
<td></td>
<td>CALL SSLR1</td>
</tr>
<tr>
<td>Language</td>
<td>Syntax</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>C and C++</td>
<td>sspr</td>
</tr>
<tr>
<td></td>
<td>ssysr</td>
</tr>
<tr>
<td></td>
<td>sslr1</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_ssp</td>
</tr>
<tr>
<td></td>
<td>cblas_ssy</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 vector x and the order of matrix A.
Specified as: an integer; n ≥ 0.

**alpha**
is the scaling constant α.
Specified as: a number of the data type indicated in Table 94 on page 321.

**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 94 on page 321.

**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.
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 94 on page 321.

\( a \) 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 94 on page 321.

\( lda \) 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 94 on page 321.

\( a \) 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 94 on page 321.

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 “Vector concepts” on page 71.
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 80. 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 84.

Function

These subroutines perform the computations described. See references [“44” on page 1255], [“45” on page 1255], and [“95” on page 1258].

Note: If \( n \) or \( \alpha \) is 0, no computation is performed.

For CHPR and CHER

Intermediate results are accumulated in long precision when the VSX unit is not used.
For SSPR, SSYR, and SSLR1
Intermediate results are accumulated in long precision on some platforms when the 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:

\[ A \leftarrow A + \alpha xx^T \]

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:

\[
\begin{bmatrix}
a_{11} & \cdots & a_{1n} \\
& \ddots & \vdots \\
& & a_{nn}
\end{bmatrix}
\leftarrow
\begin{bmatrix}
a_{11} & \cdots & a_{1n} \\
& \ddots & \vdots \\
& & a_{nn}
\end{bmatrix}
+ \alpha
\begin{bmatrix}
x_1 \\
\vdots \\
x_n
\end{bmatrix}
\begin{bmatrix}
x_1 & \cdots & x_n
\end{bmatrix}
\]

For CHPR, ZHPR, CHER, and ZHER
These subroutines compute the rank-one update of a complex Hermitian matrix:

\[ A \leftarrow A + \alpha xx^H \]

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:

\[
\begin{bmatrix}
a_{11} & \cdots & a_{1n} \\
& \ddots & \vdots \\
& & a_{nn}
\end{bmatrix}
\leftarrow
\begin{bmatrix}
a_{11} & \cdots & a_{1n} \\
& \ddots & \vdots \\
& & a_{nn}
\end{bmatrix}
+ \alpha
\begin{bmatrix}
\bar{x}_1 \\
\vdots \\
\bar{x}_n
\end{bmatrix}
\begin{bmatrix}
\bar{x}_1 & \cdots & \bar{x}_n
\end{bmatrix}
\]

Error conditions

Computational Errors
None

Input-Argument Errors
1. cblas_layout ≠ CblasRowMajor or CblasColMajor
2. uplo ≠ 'L' or 'U'
3. \texttt{cblas\_uplo} \neq \texttt{CblasLower} or \texttt{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:

\[
\begin{pmatrix}
8.0 & 4.0 & 2.0 \\
4.0 & 6.0 & 7.0 \\
2.0 & 7.0 & 3.0
\end{pmatrix}
\]

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:

\[
\begin{pmatrix}
(1.0, 0.0) & (3.0, 5.0) & (2.0, -3.0) \\
(3.0, -5.0) & (7.0, 0.0) & (4.0, -8.0) \\
(2.0, 3.0) & (4.0, 8.0) & (6.0, 0.0)
\end{pmatrix}
\]

**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 \(\alpha \neq 0.0\), 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 $\mathbf{x}$ with a negative stride, and a complex Hermitian matrix $\mathbf{A}$ of order 3, stored in upper-packed storage mode. It uses the same input matrix $\mathbf{A}$ as in Example 3.

Next: On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix $\mathbf{A}$ 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.

Example 5
This example shows a vector $\mathbf{x}$ with a positive stride, and a real symmetric matrix $\mathbf{A}$ of order 3, stored in lower storage mode. It uses the same input matrix $\mathbf{A}$ as in Example 1.

Example 6
This example shows a vector $\mathbf{x}$ with a negative stride, and a real symmetric matrix $\mathbf{A}$ of order 3, stored in upper storage mode. It uses the same input matrix $\mathbf{A}$ as in Example 1.
Example 7

This example shows a vector $x$ with a positive stride, and a complex Hermitian matrix $A$ of order 3, stored in lower storage mode. It uses the same input matrix $A$ as in Example 3.

**Note:** On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix $A$ are assumed to be zero, so you do not have to set these values. On output, if $\alpha \neq 0.0$, they are set to zero.

**Call Statement and Input:**

```
UPLO  N  ALPHA  X  INCX  A  LDA
|    |    |    |    |   |   |   |
CALL CHER( 'L', 3 , 1.0 , X , 1 , A , 3 )
```

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

```
A = | (1.0, .)       .            .   |
    | (3.0, -5.0)   (7.0, .)        .   |
    | (2.0, 3.0)  (4.0, 8.0)  (6.0, .) |
```

**Output:**

```
A = | (6.0, 0.0)        .            .      |
    | (7.0, -13.0)   (23.0, 0.0)       .      |
    | (13.0, 1.0)  (16.0, 24.0)  (31.0, 0.0) |
```

This example shows a vector $x$ with a negative stride, and a complex Hermitian matrix $A$ of order 3, stored in upper storage mode. It uses the same input matrix $A$ as in Example 3.

**Note:** On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix $A$ are assumed to be zero, so you do not have to set these values. On output, if $\alpha \neq 0.0$, they are set to zero.

**Call Statement and Input:**

```
UPLO  N  ALPHA  X  INCX  A  LDA
|    |    |    |    |   |   |   |
CALL CHER( 'U', 3 , 1.0 , X , -2 , A , 3 )
```

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

```
A = | (1.0, . )  (3.0, 5.0)  (2.0, -3.0) |
    |     .      (7.0, . )  (4.0, -8.0) |
    |     .          .         (6.0, . ) |
```

**Output:**

```
A = | (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 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:

<table>
<thead>
<tr>
<th>N</th>
<th>ALPHA</th>
<th>X</th>
<th>INCX</th>
<th>AP</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL SSLR1( 3 , 1.0 , X , 1 , AP )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>(3.0, 2.0, 1.0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP</td>
<td>(8.0, 4.0, 2.0, 6.0, 7.0, 3.0)</td>
</tr>
</tbody>
</table>

Output:

| AP | (17.0, 10.0, 5.0, 10.0, 9.0, 4.0) |

SSPR2, DSPR2, CHPR2, ZHPR2, SSYR2, DSYR2, CHER2, ZHER2, SSLR2, and DSLR2
(Rank-Two Update of a Real Symmetric or Complex Hermitian Matrix)

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 95: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$, $A$, $x$, $y$</td>
<td>SSPR2, SSYR2, and SSLR2</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>DSPR2, DSYR2, and DSLR2</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>CHPR2 and CHER2</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>ZHPR2 and ZHER2</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td></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” on page 34.
### Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
</table>
| **Fortran** | CALL SSPR2 | DSPR2 | CHPR2 | ZHPR2 (uplo, n, alpha, x, incx, y, incy, ap)  
CALL SSYR2 | DSYR2 | CHER2 | ZHER2 (uplo, n, alpha, x, incx, y, incy, a, lda)  
CALL SSLR2 | DSLR2 (n, alpha, x, incx, y, incy, ap) |
| **C and C++** | sspr2 | dspr2 | chpr2 | zhpr2 (uplo, n, alpha, x, incx, y, incy, ap);  
ssyr2 | dsyr2 | cher2 | zher2 (uplo, n, alpha, x, incx, y, incy, a, lda);  
sslr2 | dslr2 (n, alpha, x, incx, y, incy, ap); |
| **CBLAS** | cblas_sspr2 | cblas_dspr2 | cblas_chpr2 | cblas_zhpr2 (cblas_layout, cblas_uplo, n, alpha, x, incx, y, incy, ap);  
cblas_ssyr2 | cblas_dsyr2 | cblas_cher2 | cblas_zher2 (cblas_layout, cblas_uplo, n, alpha, x, incx, y, incy, a, lda); |

### 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 95 on page 328.

**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 95 on page 328.

**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) length $1+(n-1)|incy|$, containing numbers of the data type indicated in Table 95 on page 328.

$incy$ is the stride for vector $y$.

Specified as: an integer, where:

For SSPR2, DSPR2, CHPR2, ZHPR2, SSYR2, DSYR2, CHER2, and ZHER2, $incy < 0$ or $incy > 0$.

For SSLR2 and DSLR2, $incy$ can have any value.

$ap$ has the following meaning:

For SSPR2 and DSPR2, $ap$ is the real symmetric matrix $A$ of order $n$, stored in upper- or lower-packed storage mode.

For CHPR2 and ZHPR2, $ap$ is the complex Hermitian matrix $A$ of order $n$, stored in upper- or lower-packed storage mode.

For SSLR2 and DSLR2, $ap$ is the real symmetric matrix $A$ of order $n$, stored in lower-packed storage mode.

Specified as: a one-dimensional array of (at least) length $n(n+1)/2$, containing numbers of the data type indicated in Table 95 on page 328.

$a$ 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 95 on page 328.

$lda$ 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 95 on page 328.

$a$ 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 95 on page 328.

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 “Vector concepts” on page 71.
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 $\alpha \neq 0$, 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 80. 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 84.

Function

These subroutines perform the computation described. See references [“44” on page 1255], [“45” on page 1255], and [“95” on page 1258]. 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 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:

$$\begin{bmatrix}
a_{11} & \cdots & a_{1n} \\
\vdots & \ddots & \vdots \\
a_{n1} & \cdots & a_{nn}
\end{bmatrix} \leftarrow \begin{bmatrix}
a_{11} & \cdots & a_{1n} \\
\vdots & \ddots & \vdots \\
a_{n1} & \cdots & a_{nn}
\end{bmatrix} + \alpha \begin{bmatrix}
x_1 \\
\vdots \\
x_n
\end{bmatrix} \begin{bmatrix}
y_1 \\
\vdots \\
y_n
\end{bmatrix}$$

$$+ \alpha \begin{bmatrix}
y_1 \\
\vdots \\
y_n
\end{bmatrix} \begin{bmatrix}
x_1 & \cdots & x_n
\end{bmatrix}$$

CHPR2, ZHPR2, CHER2, and ZHER2

These subroutines compute the rank-two update of a complex Hermitian matrix:

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

where:
A is a complex Hermitian matrix of order n.
α 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:

\[
\begin{bmatrix}
a_{11} & \cdots & a_{1n} \\
\vdots & \ddots & \vdots \\
a_{n1} & \cdots & a_{nn}
\end{bmatrix}
\begin{bmatrix}
a_{11} & \cdots & a_{1n} \\
\vdots & \ddots & \vdots \\
a_{n1} & \cdots & a_{nn}
\end{bmatrix}
+ \alpha
\begin{bmatrix}
x_1 \\
\vdots \\
x_n
\end{bmatrix}
+ \overline{\alpha}
\begin{bmatrix}
y_1 \\
\vdots \\
y_n
\end{bmatrix}
\begin{bmatrix}
\overline{y}_1 \\
\vdots \\
\overline{y}_n
\end{bmatrix}
\]

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. incx = 0
6. incy = 0
7. lda ≤ 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:

\[
\begin{bmatrix}
8.0 & 4.0 & 2.0 \\
4.0 & 6.0 & 7.0
\end{bmatrix}
\]
Call Statement and Input:

\[
\begin{align*}
\text{UPLO} & \quad \text{N} & \quad \text{ALPHA} & \quad \text{X} & \quad \text{INCX} & \quad \text{Y} & \quad \text{INCY} & \quad \text{AP} \\
\text{CALL} & \quad \text{SSPR2}( 'L' , 3 , 1.0 , X , 1 , Y , 2 , AP ) \\
X & \quad = \quad (3.0, 2.0, 1.0) \\
Y & \quad = \quad (5.0, . , 3.0, . , 2.0) \\
AP & \quad = \quad (8.0, 4.0, 2.0, 6.0, 7.0, 3.0)
\end{align*}
\]

Output:

\[
\begin{align*}
\text{AP} & \quad = \quad (38.0, 23.0, 13.0, 18.0, 14.0, 7.0)
\end{align*}
\]

**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 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:

\[
\begin{align*}
\text{UPLO} & \quad \text{N} & \quad \text{ALPHA} & \quad \text{X} & \quad \text{INCX} & \quad \text{Y} & \quad \text{INCY} & \quad \text{AP} \\
\text{CALL} & \quad \text{SSPR2}( 'U' , 3 , 1.0 , X , -2 , Y , 2 , AP ) \\
X & \quad = \quad (1.0, . , 2.0, . , 3.0) \\
Y & \quad = \quad (5.0, . , 3.0, . , 2.0) \\
AP & \quad = \quad (8.0, 4.0, 6.0, 2.0, 7.0, 3.0)
\end{align*}
\]

Output:

\[
\begin{align*}
\text{AP} & \quad = \quad (38.0, 23.0, 18.0, 13.0, 14.0, 7.0)
\end{align*}
\]

**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:

\[
\begin{bmatrix}
(1.0, 0.0) & (3.0, 5.0) & (2.0, -3.0) \\
(3.0, -5.0) & (7.0, 0.0) & (4.0, -8.0) \\
(2.0, 3.0) & (4.0, 8.0) & (6.0, 0.0)
\end{bmatrix}
\]

**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 \(\alpha \neq \text{zero}\), the imaginary parts of the diagonal elements are set to zero.

Call Statement and Input:

\[
\begin{align*}
\text{UPLO} & \quad \text{N} & \quad \text{ALPHA} & \quad \text{X} & \quad \text{INCX} & \quad \text{Y} & \quad \text{INCY} & \quad \text{AP} \\
\text{CALL} & \quad \text{CHPR2}( 'L' , 3 , \text{ALPHA} , X , 1 , Y , 2 , AP ) \\
\text{ALPHA} & \quad = \quad (1.0, 0.0) \\
X & \quad = \quad ((1.0, 2.0), (4.0, 0.0), (3.0, 4.0)) \\
Y & \quad = \quad ((1.0, 0.0), . , (2.0, -1.0), . , (2.0, 1.0)) \\
AP & \quad = \quad ((1.0, . ), (3.0, -5.0), (2.0, 3.0), (7.0, . ), (4.0, 8.0), (6.0, . ))
\end{align*}
\]

Output:

\[
\begin{align*}
\text{AP} & \quad = \quad ((3.0, 0.0), (7.0, -10.0), (9.0, 4.0), (23.0, 0.0), (14.0, 23.0), (26.0, 0.0))
\end{align*}
\]
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 )
|    |     |     |    |   |   |   |   |
ALPHA  =  (1.0, 0.0)  
X     =  ((3.0, 4.0), . , (4.0, 0.0), . , (1.0, 2.0))  
Y     =  ((1.0, 0.0), . , (2.0, -1.0), . , (2.0, 1.0))  
AP    =  ((1.0, . ), (3.0, 5.0), (7.0, . ), (2.0, -3.0), 
         (4.0, -8.0), (6.0, . ))
```

Output:

```
AP    =  ((3.0, 0.0), (7.0, 10.0), (23.0, 0.0), (9.0, -4.0), 
         (14.0, -23.0), (26.0, 0.0))
```

Example 5
This example shows vectors \( x \) and \( y \) with positive strides, and a real symmetric matrix \( A \) of order 3, stored in lower storage mode. It uses the same input matrix \( A \) as in Example 1.

Call Statement and Input:

```
UPLO  N  ALPHA  X  INCX Y  INCY A  LDA
|    |    |    |   |   |   |   |   |
CALL SSYR2('L', 3 , 1.0 , X , 1 , Y , 2 , A , 3 )
|    |    |    |   |   |   |   |   |
X     =  (3.0, 2.0, 1.0)  
Y     =  (5.0, . , 3.0, . , 2.0)  
A     =  | 8.0  4.0  2.0 |
         |  4.0  6.0 .   |
         |  2.0  7.0  3.0 |
```

Output:

```
A     =  | 38.0    .    .   |
         | 23.0  18.0   .   |
         | 13.0  14.0  7.0 |
```

Example 6
This example shows vector \( x \) and \( y \) having strides of opposite signs. For \( x \), which has negative stride, processing begins at element \( X(5) \), which is 3.0. The real symmetric matrix \( A \) of order 3 is stored in upper storage mode. It uses the same input matrix \( A \) as in Example 1.

Call Statement and Input:

```
UPLO  N  ALPHA  X  INCX Y  INCY A  LDA
|    |    |    |    |   |   |   |   |
CALL SSYR2('U', 3 , 1.0 , X , -2 , Y , 2 , A , 4 )
|    |    |    |    |   |   |   |   |
X     =  (1.0, . , 2.0, . , 3.0)  
Y     =  (5.0, . , 3.0, . , 2.0)  
A     =  | 8.0  4.0  2.0 |
         | .   6.0  7.0 |
         | .   .  3.0 |
```
Example 7

This example shows vector $x$ and $y$ with positive stride, and a complex Hermitian matrix $A$ of order 3, stored in lower storage mode. It uses the same input matrix $A$ as in Example 3.

**Note:** On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix $A$ are assumed to be zero, so you do not have to set these values. On output, if $\alpha \neq 0$, the imaginary parts of the diagonal elements are set to zero.

Call Statement and Input:

```
UPLO  N   ALPHA   X  INCX  Y  INCY A  LDA
      |     |     |   |   |   |   |   |
CALL CHER2( 'L' , 3 , ALPHA , X , 1 , Y , 2 , A , 3 )
```

```
ALPHA    =  (1.0, 0.0)
X        =  ((1.0, 2.0), (4.0, 0.0), (3.0, 4.0))
Y        =  ((1.0, 0.0), (2.0, -1.0), (2.0, 1.0))
```

```
A  =  
[ (1.0, .)       .          .     ]
[ (3.0, -5.0)   (7.0, . )      .     ]
[ (2.0, 3.0)  (4.0, 8.0)  (6.0, . ) ]
```

Output:

```
A  =  
[ (1.0, . ) (3.0, 5.0) (2.0, -3.0) ]
[     .      (7.0, . )   (4.0, -8.0) ]
[     .          .         (6.0, . ) ]
```

Example 8

This example shows vector $x$ and $y$ having strides of opposite signs. For $x$, which has negative stride, processing begins at element $X(5)$, which is $(1.0, 2.0)$. The complex Hermitian matrix $A$ of order 3 is stored in upper storage mode. It uses the same input matrix $A$ as in Example 3.

**Note:** On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix $A$ are assumed to be zero, so you do not have to set these values. On output, if $\alpha \neq 0$, the imaginary parts of the diagonal elements are set to zero.

Call Statement and Input:

```
UPLO  N   ALPHA   X  INCX Y  INCY A  LDA
      |     |     |    |   |   |   |   |
CALL CHER2( 'U' , 3 , ALPHA , X , -2 , Y , 2 , A , 3 )
```

```
ALPHA    =  (1.0, 0.0)
X        =  ((3.0, 4.0), (4.0, 0.0), (1.0, 2.0))
Y        =  ((1.0, 0.0), (2.0, -1.0), (2.0, 1.0))
```

```
A  =  
[ (1.0, . )   (3.0, 5.0)   (2.0, -3.0) ]
[     .      (7.0, . )   (4.0, -8.0) ]
[     .          .         (6.0, . ) ]
```

Output:

```
A  =  
[ (3.0, 0.0) (7.0, -10.0)  (23.0, 0.0 ) ]
[ (9.0, 4.0) (14.0, 23.0) (26.0, 0.0 ) ]
```

Linear algebra subprograms 335
Example 9
This example shows vectors \( \mathbf{x} \) and \( \mathbf{y} \) with positive strides and a real symmetric matrix \( \mathbf{A} \) of order 3, stored in lower-packed storage mode. It uses the same input matrix \( \mathbf{A} \) as in Example 1.

Call Statement and Input:

\[
\begin{array}{ccccccc}
N & \text{ALPHA} & \mathbf{X} & \text{INCX} & \mathbf{Y} & \text{INCY} & \mathbf{AP} \\
3 & 1.0 & (3.0, 2.0, 1.0) & 1 & (5.0, . , 3.0, . , 2.0) & 2 & (8.0, 4.0, 2.0, 6.0, 7.0, 3.0) \\
\end{array}
\]

Output:

\[
\begin{array}{ccccccc}
\mathbf{AP} & = & (38.0, 23.0, 13.0, 18.0, 14.0, 7.0) \\
\end{array}
\]

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 \( \mathbf{x} \) and \( \mathbf{y} \), and general band matrix \( \mathbf{A} \) or its transpose:

\[
\mathbf{y} \leftarrow \beta \mathbf{y} + \alpha \mathbf{A} \mathbf{x}
\]

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 \( \mathbf{x} \) and \( \mathbf{y} \), and general band matrix \( \mathbf{A} \), its transpose, or its conjugate transpose:

\[
\mathbf{y} \leftarrow \beta \mathbf{y} + \alpha \mathbf{A}^T \mathbf{x}
\]

\[
\mathbf{y} \leftarrow \beta \mathbf{y} + \alpha \mathbf{A}^H \mathbf{x}
\]

Table 96: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha, \beta, \mathbf{x}, \mathbf{y}, \mathbf{A} )</td>
<td>SGBMV, DGBMV, CGBMV, ZGBMV</td>
</tr>
<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

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGBMV</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgbmv</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_sgbmv</td>
</tr>
</tbody>
</table>

### 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 `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:

- If `transa` = 'N', it is the length of vector \( y \).
- If `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 `transa` = 'N', it is the length of vector \( x \).
- If `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; \( mu \geq 0 \).

\textbf{alpha}

is the scaling constant \( \alpha \).
Specified as: a number of the data type indicated in Table 96 on page 336.

\textbf{a}

is the \( m \) by \( n \) general band matrix \( A \), stored in BLAS-general-band storage mode. It has an upper band width \( mu \) and a lower band width \( ml \). Also:

If \( transa = 'N' \), \( A \) is used in the computation.
If \( transa = 'T' \), \( A^T \) is used in the computation.
If \( transa = 'C' \), \( A^H \) 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 in BLAS-general-band storage mode.
Specified as: an \( lda \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 96 on page 336, where \( lda \geq ml+mu+1 \).

\textbf{lda}

is the leading dimension of the array specified for \( a \).
Specified as: an integer; \( lda > 0 \) and \( lda \geq ml+mu+1 \).

\textbf{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 96 on page 336, where:

If \( transa = 'N' \), it must have at least \( 1+(n-1)|incx| \) elements.
If \( transa = 'T' \) or \( 'C' \), it must have at least \( 1+(m-1)|incx| \) elements.

\textbf{incx}

is the stride for vector \( x \).
Specified as: an integer; \( incx > 0 \) or \( incx < 0 \).

\textbf{beta}

is the scaling constant \( \beta \).
Specified as: a number of the data type indicated in Table 96 on page 336.

\textbf{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 96 on page 336, where:

If \( transa = 'N' \), it must have at least \( 1+(m-1)|incy| \) elements.
If \( transa = 'T' \) or \( 'C' \), it must have at least \( 1+(n-1)|incy| \) elements.

\textbf{incy}

is the stride for vector \( y \).
Specified as: an integer; \( incy > 0 \) or \( incy < 0 \).

\textbf{On Return}
$y$ is the vector $y$, containing the result of the computation, where:

- If $\text{transa} = 'N'$, it has length $m$.
- If $\text{transa} = 'T'$ or 'C', it has length $n$.

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 96 on page 336.

**Notes**

1. For SGBMV and DGBMV, if you specify 'C' for the $\text{transa}$ argument, it is interpreted as though you specified 'T'.
2. All subroutines accept lowercase letters for the $\text{transa}$ argument.
3. Vector $y$ must have no common elements with matrix $A$ or vector $x$; otherwise, results are unpredictable. See “Vector concepts” on page 71.
4. To achieve optimal performance, use $\text{lda} = \text{mu} + \text{ml} + 1$.
5. For general band matrices, if you specify $\text{ml} \geq m$ or $\text{mu} \geq n$, 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 $\text{ml}$ and $\text{mu}$ 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 94. 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 94.

**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 [“44” on page 1255], [“45” on page 1255], [“48” on page 1255], [“56” on page 1256], and [“95” on page 1258]. 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:

$$y = \beta y + \alpha Ax$$

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 $\text{ml}$ and an upper band width of $\text{mu}$.

**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. `cblas_layout` ≠ CblasRowMajor or CblasColMajor
2. `transa` ≠ 'N', 'T', or 'C'
3. `cblas_transa` ≠ 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. \( incx = 0 \)
11. \( incy = 0 \)
Examples

Example 1
This example shows how to use SGBMV to perform the computation $y \leftarrow \beta y + \alpha A x$, where TRANSA is equal to 'N', and the following real general band matrix $A$ is used in the computation. Matrix $A$ is:

$$
\begin{pmatrix}
1.0 & 1.0 & 1.0 & 0.0 \\
2.0 & 2.0 & 2.0 & 2.0 \\
3.0 & 3.0 & 3.0 & 3.0 \\
4.0 & 4.0 & 4.0 & 4.0 \\
0.0 & 5.0 & 5.0 & 5.0
\end{pmatrix}
$$

Call Statement and Input:

```
CALL SGBMV( 'N' , 5 , 4 , 3 , 2 , 2.0 , A , 8 , X , 1 , 10.0 , Y , 2 )
```

Output:

$Y = (22.0, . , 60.0, . , 90.0, . , 120.0, . , 140.0, . )$

Example 2
This example shows how to use SGBMV to perform the computation $y \leftarrow \beta y + \alpha A^T x$, where TRANSA is equal to 'T', and the transpose of a real general band matrix $A$ is used in the computation. It uses the same input as Example 1.

Call Statement and Input:

```
CALL SGBMV( 'T' , 5 , 4 , 3 , 2 , 2.0 , A , 8 , X , 1 , 10.0 , Y , 2 )
```

Output:

$Y = (70.0, . , 130.0, . , 140.0, . , 148.0, . )$

Example 3
This example shows how to use CGBMV to perform the computation $y \leftarrow \beta y + \alpha A^H x$, where TRANSA is equal to 'C', and the complex conjugate of the following general band matrix $A$ is used in the computation. Matrix $A$ is:

$$
\begin{pmatrix}
(1.0, 1.0) & (1.0, 1.0) & (1.0, 1.0) & (0.0, 0.0) \\
(2.0, 2.0) & (2.0, 2.0) & (2.0, 2.0) & (2.0, 2.0) \\
(3.0, 3.0) & (3.0, 3.0) & (3.0, 3.0) & (3.0, 3.0) \\
(4.0, 4.0) & (4.0, 4.0) & (4.0, 4.0) & (4.0, 4.0) \\
(0.0, 0.0) & (5.0, 5.0) & (5.0, 5.0) & (0.0, 0.0)
\end{pmatrix}
$$
Example 4
This example shows how to use SGBMV to perform the computation \( y \leftarrow \beta y + \alpha A x \), 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:

\[
\begin{pmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
2.0 & 2.0 & 2.0 & 2.0 & 2.0 \\
3.0 & 3.0 & 3.0 & 3.0 & 3.0 \\
4.0 & 4.0 & 4.0 & 4.0 & 4.0 \\
\end{pmatrix}
\]

Call Statement and Input:

\[
\text{CALL SGBMV}( 'N' , 4 , 5 , 6 , 5 , 2.0 , A , 12 , X , 1 , 10.0 , Y , 2 )
\]

\[
A = \begin{pmatrix}
. & . & . & . & 1.0 \\
. & . & 1.0 & 2.0 & . \\
. & 1.0 & 2.0 & 3.0 & 4.0 \\
\end{pmatrix}
\]

\[
X = (1.0, 2.0, 3.0, 4.0, 5.0)
\]

\[
Y = (1.0, . , 2.0, . , 3.0, . , 4.0, . )
\]

Output:

\[
Y = (40.0, . , 80.0, . , 120.0, . , 160.0, . )
\]
SSBMV, DSBMV, CHBMV, and ZHBMV (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Band Matrix)

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 97: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$, $\beta$, $x$, $y$, $A$</td>
<td></td>
</tr>
<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

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SSBMV</td>
</tr>
<tr>
<td>C and C++</td>
<td>ssbmv</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_ssbmv</td>
</tr>
</tbody>
</table>

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 97 on page 343.

\( A \)

is the real symmetric or complex Hermitian band matrix \( A \) of order \( n \), having a half band width of \( k \), where:

If \( \text{uplo} = 'U' \), \( A \) is stored in upper-band-packed storage mode.

If \( \text{uplo} = 'L' \), \( A \) 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 97 on page 343, 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 97 on page 343.

\( \text{incx} \)

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 97 on page 343.

\( y \)

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 97 on page 343.

\( \text{incy} \)

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 97 on page 343.

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 “Vector concepts” on page 71.

3. To achieve optimal performance in these subroutines, use $lda = k+1$.

4. The imaginary parts of the diagonal elements of the complex Hermitian matrix $A$ are assumed to be zero, so you do not have to set these values.

5. For real symmetric and complex Hermitian band matrices, if you specify $k \geq n$, ESSL assumes, only for purposes of the computation, that the half band width of matrix $A$ is $n-1$; that is, it processes matrix $A$, of order $n$, as though it is a (nonbanded) real symmetric or complex Hermitian matrix. However, ESSL uses the original value for $k$ for the purposes of finding the locations of element $a_{11}$ and all other elements in the array specified for $A$, as described in the storage modes referenced in the next note. For an illustration of this technique, see Example 3.

6. For a description of how a real symmetric band matrix is stored, see “Upper-Band-Packed Storage Mode” on page 100 or “Lower-Band-Packed Storage Mode” on page 100. For a description of how a complex Hermitian band matrix is stored, see “Complex Hermitian Matrix” on page 84.

**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 VSX unit is not used. Occasionally, for performance reasons, these intermediate results are truncated to short precision and stored.

See references [“44” on page 1255], [“48” on page 1255], [“56” on page 1256], and [“95” on page 1258]. 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. $cblas_layout \neq$ CblasRowMajor or CblasColMajor
2. $uplo \neq 'U$' or 'L'
3. $cblas_uplo \neq$ CblasLower or CblasUpper
4. $n < 0$
5. $k < 0$
6. $lda \leq 0$
7. $lda < k+1$
8. $incx = 0$
9. $incy = 0$
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:

$$
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 2.0 & 2.0 & 2.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 2.0 & 3.0 & 3.0 & 3.0 & 0.0 & 0.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 4.0 & 4.0 & 4.0 \\
0.0 & 2.0 & 3.0 & 4.0 & 5.0 & 5.0 & 5.0 \\
0.0 & 0.0 & 3.0 & 4.0 & 5.0 & 6.0 & 6.0 \\
0.0 & 0.0 & 0.0 & 4.0 & 5.0 & 6.0 & 7.0 \\
\end{bmatrix}
$$

Call Statement and Input:

```fortran
CALL SSBMV( 'U' , 7 , 3 , 2.0 , A , 5 , X , 1 , 10.0 , Y , 2 )
```

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:

```fortran
CALL CHBMV( 'L' , 7 , 3 , ALPHA , A , 5 , X , 1 , BETA , Y , 2 )
```
**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:

\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 2.0 & 2.0 & 2.0 & 2.0 \\
1.0 & 2.0 & 3.0 & 3.0 & 3.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 4.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 \\
\end{bmatrix}
\]

Call Statement and Input:

\[
\text{CALL SSBMV( 'U', 5, 5, 2.0, A, 7, X, 1, 10.0, Y, 2 )}
\]

\[
\begin{bmatrix}
1.0 & . & . & . & 1.0 \\
. & 1.0 & 2.0 & . & . \\
. & 1.0 & 2.0 & 3.0 & . \\
. & . & 1.0 & 2.0 & 3.0 \\
. & . & . & 1.0 & 2.0 \\
\end{bmatrix}
\]

\[
X = (1.0, 2.0, 3.0, 4.0, 5.0)
\]

\[
Y = (1.0, ., 2.0, ., 3.0, ., 4.0, ., 5.0, .)
\]

Output:

\[
Y = (40.0, ., 78.0, ., 112.0, ., 140.0, ., 160.0, .)
\]

**STRMV, DTRMV, CTRMV, ZTRMV, STPMV, DTPMV, CTPMV, and ZTPMV**

**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:
$x \leftarrow Ax$

$\overline{x} \leftarrow A^\top \overline{x}$

$\overline{x} \leftarrow A^H \overline{x}$

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:

$\overline{x} \leftarrow Ax$

$\overline{x} \leftarrow A^\top \overline{x}$

$\overline{x} \leftarrow A^H \overline{x}$

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>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, x</td>
<td>STRMV and STPMV</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>DTRMV and DTPMV</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>CTRMV and CTPMV</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>ZTRMV and ZTPMV</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td></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 34.

### Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fortran</strong></td>
<td>CALL STRMV</td>
</tr>
<tr>
<td></td>
<td>CALL STPMV</td>
</tr>
<tr>
<td><strong>C and C++</strong></td>
<td>strmv</td>
</tr>
<tr>
<td></td>
<td>stpmv</td>
</tr>
<tr>
<td><strong>CBLAS</strong></td>
<td>cblas_strmv</td>
</tr>
<tr>
<td></td>
<td>cblas_stpmv</td>
</tr>
</tbody>
</table>

**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; $0 \leq n \leq lda$.

**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 98 on page 348.
lda
is the leading dimension of the array specified for a.
Specified as: an integer; lda > 0 and lda ≥ 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 98 on page 348.

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 98 on page 348.

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 98 on page 348.

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 88.

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←Ax
x←A^Tx
x←A^Hx (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 [“42” on page 1255] and [“48” on page 1255]. If n is 0, no computation is performed.
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 computation $x \leftarrow Ax$. 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:

```
CALL STRMV( 'L', 'N', 'U', 4, A, 4, X, 1 )
```

$$
A = \begin{bmatrix}
1.0 & . & . & . \\
2.0 & 3.0 & . & . \\
3.0 & 4.0 & 3.0 & . \\
\end{bmatrix}
$$

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

Output:

$X = (1.0, 3.0, 11.0, 24.0)$

Example 2
This example shows the computation $x \leftarrow A^T x$. 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:

$$
\begin{bmatrix}
1.0 & 2.0 & 3.0 & 2.0 \\
. & 1.0 & 2.0 & 5.0 \\
. & . & 1.0 & 3.0 \\
\end{bmatrix}
$$

Linear algebra subprograms 351
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:

```plaintext
UPLO TRANSA DIAG  N   A  LDA  X  INCX
CALL STRMV( 'U' , 'T' , 'U' , 4 , A , 4 , X , 1 )
```

$A = \begin{bmatrix}
1.0 & 2.0 & 3.0 & 2.0 \\
2.0 & 2.0 & 5.0 & 5.0 \\
3.0 & 2.0 & 3.0 & 3.0 \\
2.0 & 2.0 & 3.0 & 1.0 \\
\end{bmatrix}$

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

Output:

$X = (5.0, 14.0, 26.0, 41.0)$

Example 3

This example shows the computation $x \leftarrow A^T x$. 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:

$$
\begin{bmatrix}
(1.0, 0.0) & (2.0, 2.0) & (3.0, 3.0) & (2.0, 2.0) \\
(1.0, 0.0) & (2.0, 2.0) & (5.0, 5.0) & \\
(1.0, 0.0) & (3.0, 3.0) & \\
& & & (1.0, 0.0) \\
\end{bmatrix}
$$

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:

```plaintext
UPLO TRANSA DIAG  N   A  LDA  X  INCX
CALL CTRMV( 'U' , 'C' , 'U' , 4 , A , 4 , X , 1 )
```

$A = \begin{bmatrix}
(2.0, 2.0) & (3.0, 3.0) & (2.0, 2.0) \\
(2.0, 2.0) & (5.0, 5.0) & \\
(3.0, 3.0) & \\
& & \\
\end{bmatrix}$

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

Output:

$X = ((5.0, 5.0), (24.0, 4.0), (49.0, 3.0), (80.0, 2.0))$

Example 4

This example shows the computation $x \leftarrow Ax$. 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:

```
UPLO TRANSA DIAG N AP X INCX
CALL STPMV( 'L' , 'N' , 'U' , 4 , AP , X , 1 )
AP    = ( . , 1.0, 2.0, 3.0, . , 3.0, 4.0, . , 3.0, . )
X     = (1.0, 2.0, 3.0, 4.0)
```

Output:

```
X     = (1.0, 3.0, 11.0, 24.0)
```

**Example 5**

This example shows the computation $x \leftarrow A^T x$. 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:

```
UPLO TRANSA DIAG N AP X INCX
CALL STPMV( 'U' , 'T' , 'N' , 4 , AP , X , 1 )
AP    = (1.0, 2.0, 2.0, 3.0, 2.0, 3.0, 2.0, 5.0, 3.0, 1.0)
X     = (5.0, 4.0, 3.0, 2.0)
```

Output:

```
X     = (5.0, 18.0, 32.0, 41.0)
```

**Example 6**

This example shows the computation $x \leftarrow A^H x$. 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:

```
UPLO TRANSA DIAG N AP X INCX
CALL CTPMV( 'U' , 'C' , 'U' , 4 , AP , X , 1 )
AP    = ( . , (2.0, 2.0), . , (3.0, 3.0), (2.0, 2.0), . ,
       (2.0, 2.0), (5.0, 5.0), (3.0, 3.0), . )
X     = ((5.0, 5.0), (4.0, 4.0), (3.0, 3.0), (2.0, 2.0))
```

Linear algebra subprograms 353
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>1. ( x \leftarrow A^{-1}x )</td>
<td>( Ax = b )</td>
</tr>
<tr>
<td>2. ( x \leftarrow A^{T}x )</td>
<td>( A^{T}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>1. ( x \leftarrow A^{-1}x )</td>
<td>( Ax = b )</td>
</tr>
<tr>
<td>2. ( x \leftarrow A^{T}x )</td>
<td>( A^{T}x = b )</td>
</tr>
<tr>
<td>3. ( x \leftarrow A^{H}x )</td>
<td>( A^{H}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 99: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A, x )</td>
<td>STRSV and STPSV</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>DTRSV and DTPSV</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>CTRSV and CTPSV</td>
</tr>
<tr>
<td>Short-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 34.
## Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
</table>
| **Fortran** | CALL STRSV | DTRSV | CTRSV | ZTRSV (uplo, transa, diag, n, a, lda, x, incx)  
CALL STPSV | DTPSV | CTPSV | ZTPSV (uplo, transa, diag, n, ap, x, incx) |
| **C and C++** | strsv | dtrsv | ctrsv | ztrsv (uplo, transa, diag, n, a, lda, x, incx);  
stpsv | dtpsv | ctpsv | ztpsv (uplo, transa, diag, n, ap, x, incx); |
| **CBLAS** | cblas_strsv | cblas_dtrsv | cblas_ctrsv | cblas_ztrsv (cblas_layout, cblas_uplo,  
cblas_transa, cblas_diag, n, a, lda, x, incx);  
cblas_stpsv | cblas_dtpsv | cblas_ctpsv | cblas_ztpsv (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 \(A\) used in the system of equations, where:
  - If `transa` = 'N', \(A\) is used, resulting in solution 1.
  - If `transa` = 'T', \(A^T\) is used, resulting in solution 2.
  - If `transa` = '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` = CblasNoTrans, \(A\) is used, resulting in solution 1.
  - If `cblas_transa` = CblasTrans, \(A^T\) is used, resulting in solution 2.
  - If `cblas_transa` = 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 \( \text{diag} = 'U' \), \( A \) 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 99 on page 354.

\( \text{lda} \)

is the leading dimension of the array specified for \( a \).

Specified as: an integer; \( \text{lda} > 0 \) and \( \text{lda} \geq n \).

\( ap \)

is the upper or lower triangular matrix \( A \) of order \( n \), stored in upper- or lower-triangular-packed storage mode, respectively.

Specified as: a one-dimensional array of (at least) length \( n(n+1)/2 \), containing numbers of the data type indicated in Table 99 on page 354.

\( x \)

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 99 on page 354.

\( \text{incx} \)

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 99 on page 354.

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 88.

**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:

\[
\begin{align*}
    x & \leftarrow A^{-1}x \\
    x & \leftarrow A^T x \\
    x & \leftarrow A^H x \quad \text{(only for CTRSV, ZTRSV, CTPSV, and ZTPSV)}
\end{align*}
\]

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 [“42” on page 1255], [“46” on page 1255], and [“48” on page 1255].

**Error conditions**

**Computational Errors**

None

**Input-Argument Errors**

1. \texttt{cblas_layout} ≠ CblasRowMajor or CblasColMajor
2. \texttt{uplo} ≠ 'L' or 'U'
3. \texttt{cblas_uplo} ≠ CblasLower or CblasUpper
4. \texttt{transa} ≠ 'T', 'N', or 'C'
5. \texttt{cblas_transa} ≠ CblasNoTrans, CblasTrans, or CblasConjTrans
6. \texttt{diag} ≠ 'N' or 'U'
7. \texttt{cblas_diag} ≠ CblasNonUnit or CblasUnit
8. \( n < 0 \)
9. \( lda \leq 0 \)
10. \( lda < n \)
11. \( incx = 0 \)

**Examples**

**Example 1**

This example shows the solution \( x \leftarrow A^{-1}x \). 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:

\[
\begin{array}{cccccccc}
\text{UPLO} & \text{TRANSA} & \text{DIAG} & \text{N} & \text{A} & \text{LDA} & \text{X} & \text{INCX} \\
\text{'L'} & \text{'N'} & \text{'U'} & 4 & A & 4 & X & 1 \\
\end{array}
\]

Linear algebra subprograms 357
Example 2

This example shows the solution $x \leftarrow A^{\top}x$. 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:

```
CALL STRSV( 'U' , 'T' , 'N' , 4 , A , 4 , X , 1  )
```

Output:

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

Example 3

This example shows the solution $x \leftarrow A^{\dagger}x$. 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:

```
CALL CTRSV( 'U' , 'C' , 'U' , 4 , A , 4 , X , 1  )
```

Output:

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

Example 4

This example shows the solution $x \leftarrow A^{-1}x$. 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:

```
\[
A = \begin{bmatrix}
1.0 & . & . & . \\
2.0 & 3.0 & . & . \\
3.0 & 4.0 & 3.0 & . \\
\end{bmatrix}
\]
```

Output:

$X = (1.0, 2.0, 3.0, 4.0)$
Note: Because matrix $A$ is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of 1.0 for the diagonal elements.

Call Statement and Input:

```
UPLO TRANSA DIAG  N   AP   X  INCX
|     |     |    |   |    |   |
CALL 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 $x \leftarrow A^{-T}x$. 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:

```
| 1.0  2.0  3.0  2.0 |
| .    2.0  2.0  5.0 |
| .     .  3.0  3.0 |
| .     .     .  1.0 |
```

Call Statement and Input:

```
UPLO TRANSA DIAG  N   AP   X  INCX
|     |     |    |   |    |   |
CALL 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 $x \leftarrow A^{-H}x$. 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:

```
UPLO TRANSA DIAG  N   AP   X  INCX
|     |     |    |   |    |   |
CALL CTPSV( 'U', 'C', 'U', 4 , AP , X , 1 )
```

$AP = (. , (2.0, 2.0), . , (3.0, 3.0), (2.0, 2.0), . , (2.0, 2.0), (5.0, 5.0), (3.0, 3.0), . )$

$X = ((5.0, 5.0), (24.0, 4.0), (49.0, 3.0), (80.0, 2.0))$
Output:

\[
X = ((5.0, 5.0), (4.0, 4.0), (3.0, 3.0), (2.0, 2.0))
\]

**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:

\[
\begin{align*}
  x & \leftarrow Ax \\
  x & \leftarrow A^T x
\end{align*}
\]

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:

\[
\begin{align*}
  x & \leftarrow Ax \\
  x & \leftarrow A^T x \\
  x & \leftarrow A^H x
\end{align*}
\]

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>Table 100: Data Types.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Types</strong></td>
</tr>
<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>

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL STBMV</td>
</tr>
<tr>
<td>C and C++</td>
<td>stbmv</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_stbmv</td>
</tr>
</tbody>
</table>

**On Entry**

\( cblas_layout \)

indicates whether the input matrices are stored in row major order 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.

**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 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.

**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 100 on page 360.

**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.

Specified as: a one-dimensional array of (at least) length 1+(n-1)|incx|, containing numbers of the data type indicated in Table 100 on page 360.

**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 100 on page 360.

**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 ≥ 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 102. 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 102.
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 [“44” on page 1255], [“56” on page 1256], and [“48” on page 1255]. If $n$ is 0, no computation is performed.

**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. $k < 0$
10. `lda` ≤ 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:

```
  | 1.0  1.0  1.0  1.0  0.0  0.0  0.0 |
  | 0.0  2.0  2.0  2.0  2.0  0.0  0.0 |
  | 0.0  0.0  3.0  3.0  3.0  3.0  0.0 |
  | 0.0  0.0  0.0  4.0  4.0  4.0  4.0 |
  | 0.0  0.0  0.0  0.0  5.0  5.0  5.0 |
  | 0.0  0.0  0.0  0.0  0.0  6.0  6.0 |
  | 0.0  0.0  0.0  0.0  0.0  0.0  7.0 |
```

Call Statement and Input:

```
UPLO TRANSA DIAG N   K   A   LDA   X   INCX
    |    |    | | | | | | |
CALL STBMV( 'U' , 'N' , 'N' , 7 , 3 , A , 5 , X , 1  )

        | .    .    .   1.0  2.0  3.0  4.0 |
A = | .    .    .  1.0  2.0  3.0  4.0  5.0 |
    | 1.0  2.0  3.0  4.0  5.0  6.0  7.0 |
    | .    .    .    .    .    .    .  |

X = (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0)
```
Output:

\[ \mathbf{x} = (10.0, 28.0, 54.0, 88.0, 90.0, 78.0, 49.0) \]

**Example 2**

This example shows the computation \( \mathbf{x} \leftarrow \mathbf{A}^{\text{T}} \mathbf{x} \). Matrix \( \mathbf{A} \) 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 \( \mathbf{x} \) is a vector of length 7. Matrix \( \mathbf{A} \) is:

\[
\begin{bmatrix}
1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 2.0 & 3.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 2.0 & 3.0 & 4.0 & 5.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 3.0 & 4.0 & 5.0 & 6.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 4.0 & 5.0 & 6.0 & 7.0
\end{bmatrix}
\]

Call Statement and Input:

\[
\text{CALL STBMV( 'L', 'T', 'N', 7, 3, A, 5, X, 1 )}
\]

Output:

\[ \mathbf{x} = (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0) \]

**Example 3**

This example shows the computation \( \mathbf{x} \leftarrow \mathbf{A}^{\text{H}} \mathbf{x} \). Matrix \( \mathbf{A} \) 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 \( \mathbf{x} \) is a vector of length 7. Matrix \( \mathbf{A} \) is:

\[
\begin{bmatrix}
(1.0, 1.0) & (1.0, 1.0) & (1.0, 1.0) & (1.0, 1.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (2.0, 2.0) & (2.0, 2.0) & (2.0, 2.0) & (2.0, 2.0) & (0.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (0.0, 0.0) & (3.0, 3.0) & (3.0, 3.0) & (3.0, 3.0) & (3.0, 3.0) & (3.0, 3.0) \\
(0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (4.0, 4.0) & (4.0, 4.0) & (4.0, 4.0) & (4.0, 4.0) \\
(0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (5.0, 5.0) & (5.0, 5.0) & (5.0, 5.0) \\
(0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (6.0, 6.0) & (6.0, 6.0) \\
(0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (7.0, 7.0)
\end{bmatrix}
\]

Output:

\[ \mathbf{x} = (10.0, 28.0, 54.0, 88.0, 90.0, 78.0, 49.0) \]
Call Statement and Input:

```plaintext
CALL CTBMV( 'U', 'C', 'N', 7, 3, A, 5, X, 1 )
```

\[ A = \begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 \\
. & 2.0 & 2.0 & 2.0 \\
. & . & 3.0 & 3.0 \\
. & . & . & 4.0
\end{bmatrix} \]

\[ X = ((1.0, 1.0), (2.0, 2.0), (3.0, 3.0), (4.0, 4.0), (5.0, 5.0), (6.0, 6.0), (7.0, 7.0)) \]

Output:

\[ X = ((1.0, 2.0), (7.0, 9.0), (24.0, 23.0), (58.0, 46.0), (112.0, 79.0), (186.0, 122.0), (280.0, 175.0)) \]

Example 4

This example shows the computation \( x \leftarrow A^T x \), 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:

```plaintext
CALL STBMV( 'U', 'T', 'N', 4, 5, A, 6, X, 1 )
```

\[ A = \begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 \\
. & 2.0 & 2.0 & 2.0 \\
. & . & 3.0 & 3.0 \\
. & . & . & 4.0
\end{bmatrix} \]

\[ X = (1.0, 2.0, 3.0, 4.0) \]

Output:

\[ X = (1.0, 2.0, 3.0, 4.0) \]

**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

1. $x \leftarrow A^{-1}x$
2. $x \leftarrow A^T x$

Equation

$Ax = b$
$A^x = b$

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

1. $x \leftarrow A^{-1}x$
2. $x \leftarrow A^T x$
3. $x \leftarrow A^H x$

Equation

$Ax = b$
$A^x = b$
$A^H x = b$

Matrix $A$ can be either upper or lower triangular and is stored in upper- or lower-triangular-band-packed storage mode, respectively.

<table>
<thead>
<tr>
<th>Table 101: Data Types.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Types</strong></td>
</tr>
<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>

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL STBSV</td>
</tr>
<tr>
<td>C and C++</td>
<td>stbsv</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_stbsv</td>
</tr>
<tr>
<td></td>
<td>cblas_ztbsv (cblas_layout, cblas_uplo, cblas_trans, 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 order 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 $uplo = \text{'U'}$, $A$ is an upper triangular matrix.
    - If $uplo = \text{'L'}$, $A$ 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 CBLASTRANSPOSE. 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 101 on page 366.

**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 \(n\), 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 101 on page 366.
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 101 on page 366.

Notes
1. These subroutines accept lowercase letters for the uplo, trans, and diag arguments.
2. For STBSV and DTBSV, if you specify 'C' for the trans argument, it is interpreted as though you specified 'T'.
3. Matrix \( A \) and vector \( x \) must have no common elements; otherwise, results are unpredictable.
4. For unit triangular matrices, the elements of the diagonal are assumed to be 1.0 for real matrices and \((1.0, 0.0)\) for complex matrices, and you do not need to set these values in the array.
5. For both upper and lower triangular band matrices, if you specify \( k \geq n \), ESSL assumes, for purposes of the computation only, that the upper or lower band width of matrix \( A \) is \( n-1 \); that is, it processes matrix \( A \) of order \( n \), as though it is a (nonbanded) triangular matrix. However, ESSL uses the original value for \( k \) for the purposes of finding the locations of element \( a_{11} \) and all other elements in the array specified for \( A \), as described in “Triangular Band Matrix” on page 102. 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 102.
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. \( x \leftarrow A^{-1}x \)
2. \( x \leftarrow A^{T}x \)
3. \( x \leftarrow A^{H}x \) (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 [“44” on page 1255], [“56” on page 1256], and [“48” on page 1255]. If \( n \) is 0, no computation is performed.

Error conditions
Computational Errors
None
**Input-Argument Errors**

1. `cblas_layout` ≠ CblasRowMajor or CblasColMajor
2. `n` < 0
3. `k` < 0
4. `lda` ≤ 0
5. `lda` < `k`+1
6. `incx` = 0
7. `uplo` ≠ 'L' or 'U'
8. `cblas_uplo` ≠ CblasLower or CblasUpper
9. `trans` ≠ 'T', 'N', or 'C'
10. `cblas_transa` ≠ CblasNoTrans, CblasTrans, or CblasConjTrans
11. `diag` ≠ 'N' or 'U'
12. `cblas_diag` ≠ 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:

```
UPLO TRANS DIAG N K A LDA X INCX
|     |     |    |   |   |   |   |   |
CALL STBSV( 'U' , 'N' , 'N' , 9 , 2 , A , 3 , X , 1  )
```

\( A \) =

\[
\begin{bmatrix}
. & . & 1.0 & 3.0 & 1.0 & 2.0 & 1.0 & 2.0 & 0.0 \\
. & 1.0 & 2.0 & 1.0 & 2.0 & 1.0 & 2.0 & 1.0 & 2.0 \\
1.0 & 4.0 & 4.0 & 3.0 & 3.0 & 3.0 & 2.0 & 2.0 & 1.0 \\
\end{bmatrix}
\]

\( X \) = (2.0, 7.0, 1.0, 8.0, 2.0, 8.0, 1.0, 8.0, 3.0)

Output:

\( X \) = (1.0, 1.0, 0.0, 1.0, 0.0, 2.0, 0.0, 1.0, 3.0)

**Example 2**

This example shows the solution \( x ← A^T x \), solving the same system as in Example 1. Matrix \( A \) is a real 9 by 9 lower triangular band matrix with a lower band width of 2 that is not unit triangular, stored in lower-triangular-band-packed storage mode. Vector \( x \) is a vector of length 9 where matrix \( A \) is:

\[
\begin{bmatrix}
1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 4.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 2.0 & 4.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 3.0 & 1.0 & 4.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
\end{bmatrix}
\]
Example 3

This example shows the solution \( x \leftarrow A^T x \), where \( k > n \). Matrix \( A \) 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 \( x \) is a vector of length 4 where matrix \( A \) 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:

```assembly
UPLO TRANS DIAG N K A LDA X INCX
CALL STBSV( 'U' , 'T' , 'N' , 4 , 5 , A , 6 , X , 1 )
```

\[
A = \begin{bmatrix}
1.0 & 2.0 & 3.0 & 2.0 \\
0.0 & 2.0 & 2.0 & 5.0 \\
0.0 & 0.0 & 3.0 & 3.0 \\
0.0 & 0.0 & 0.0 & 1.0
\end{bmatrix}
\]

\( X = \) (same as input \( X \) in Example 1)

Output:

\( X = \) (same as output \( X \) in Example 1)
**Example 4**

This example shows the solution $x \leftarrow A^T x$. 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:**

<table>
<thead>
<tr>
<th>UPLD</th>
<th>TRANS</th>
<th>DIAG</th>
<th>N</th>
<th>K</th>
<th>A</th>
<th>LDA</th>
<th>X</th>
<th>INCX</th>
</tr>
</thead>
<tbody>
<tr>
<td>'L'</td>
<td>'T'</td>
<td>'N'</td>
<td>7</td>
<td>3</td>
<td>A</td>
<td>4</td>
<td>X</td>
<td>1</td>
</tr>
</tbody>
</table>

**A =**

$$
\begin{bmatrix}
(1.0, 0.0) & (2.0, 1.0) & (3.0, 1.0) & (4.0, 1.0) & (2.0, 1.0) & (3.0, 1.0) & (2.0, 1.0) \\
(1.0, 2.0) & (2.0, 2.0) & (3.0, 3.0) & (4.0, 2.0) & (2.0, 1.0) & (0.0, 0.0) & (0.0, 0.0) \\
(1.0, 3.0) & (2.0, 3.0) & (3.0, 3.0) & (4.0, 3.0) & (5.0, 2.0) & .    & .    \\
(1.0, 4.0) & (2.0, 4.0) & (3.0, 3.0) & (4.0, 4.0) & .    & .    & .    
\end{bmatrix}
$$

**X =**

$$
(2.0, 2.0), (7.0, 1.0), (1.0, 1.0), (8.0, 1.0),
(2.0, 0.0), (8.0, 1.0), (1.0, 2.0)
$$

**Output:**

$$
X = ((-12.048, -13.136), (6.304, -1.472), (-1.880, 1.040),
(2.600, -1.800), (-2.160, 1.880), (0.800, -1.400),
(0.800, 0.600))
$$

---

**Sparse 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 $y$:

$$y \leftarrow Ax$$

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 $y \leftarrow A^T x$.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</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.

\( y \) See On Return.

**On Return**

\( y \) is the vector \( y \) of length \( m \), containing the result of the computation. Returned as: a one-dimensional array of (at least) length \( m \), containing long-precision real numbers.

**Notes**

1. Matrix \( A \) must have no common elements with vectors \( x \) and \( y \); otherwise, results are unpredictable.

2. For the KA array, where there are no corresponding nonzero elements in AC, you must still fill in a number between 1 and \( n \). See the Example.

3. For a description of how sparse matrices are stored in compressed-matrix storage mode, see “Compressed-Matrix Storage Mode” on page 110.

4. If your sparse matrix is stored by rows, as defined in “Storage-by-Rows” on page 115, 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 1212, 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:
See reference [“89” on page 1258]. 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 1212.

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:

```
CALL DSMMX( 6 , 4 , AC , KA , 6 , X , Y )
```

\( AC \) =

\[
\begin{bmatrix}
  4.0 & 7.0 & 0.0 & 0.0 \\
  4.0 & 3.0 & 2.0 & 0.0 \\
  4.0 & 2.0 & 4.0 & 0.0 \\
  4.0 & 7.0 & 1.0 & 0.0 \\
  4.0 & 1.0 & 3.0 & 0.0 \\
  4.0 & 1.0 & 1.0 & 3.0 \\
\end{bmatrix}
\]

\( KA \) =

\[
\begin{bmatrix}
  1 & 3 & 1 & 1 \\
  2 & 1 & 4 & 1 \\
  3 & 2 & 5 & 1 \\
  4 & 3 & 6 & 1 \\
\end{bmatrix}
\]
\[
\begin{bmatrix}
5 & 1 & 4 & 1 \\
6 & 1 & 2 & 5 \\
\end{bmatrix}
\]

\[X = (1.0, 2.0, 3.0, 4.0, 5.0, 6.0)\]

\[
\begin{bmatrix}
25.0 & 19.0 & 36.0 & 43.0 & 33.0 & 42.0 \\
\end{bmatrix}
\]

Output:

\[
Y = (25.0, 19.0, 36.0, 43.0, 33.0, 42.0)
\]

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 \( y \):

\[
y \leftarrow Ax \\
y \leftarrow A^T x
\]

where \( A, x, \) and \( y \) contain long-precision real numbers.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL DSDMX (iopt, n, nd, ad, lda, trans, la, x, y)</td>
</tr>
<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 AD. The **iopt** argument indicates the storage variation used for storing matrix \( A \). The **trans** argument indicates the following:
  - If \( trans = 'N' \), \( A \) is used in the computation.
  - If \( trans = 'T' \), \( A^T \) 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 ≥ n.

**trans**

indicates the form of matrix A to use in the computation, where:

If trans = 'N', A is used in the computation.

If trans = 'T', AT is used in the computation.

Specified as: a single character; trans = 'N' or 'T'.

**la**

is the array, referred to as LA, containing the diagonal numbers k for the diagonals stored in each corresponding column in array AD. (For an explanation of how diagonal numbers are assigned, see “Compressed-Diagonal Storage Mode” on page 111.)

Specified as: a one-dimensional array of (at least) length nd, containing integers; 1-n ≤ LA(i) ≤ n-1.

**x**

is the vector x of length n. Specified as: a one-dimensional array, containing long-precision real numbers.

**y**

See On Return.

**On Return**

y

is the vector y of length n, containing the result of the computation. Returned as: a one-dimensional array, containing long-precision real numbers.

**Notes**

1. All subroutines accept lowercase letters for the trans argument.
2. Matrix A must have no common elements with vectors x and y; otherwise, results are unpredictable.
3. For a description of how sparse matrices are stored in compressed-diagonal storage mode, see “Compressed-Diagonal Storage Mode” on page 111.

**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 ← Ax

y ← ATx

where:

A is a sparse matrix of order n, stored in compressed-diagonal storage mode in AD and LA, using the storage variation for either general or symmetric sparse matrices, as indicated by the iopt argument.

x and y are vectors of length n.

It is expressed as follows for y ← Ax:
It is expressed as follows for $\mathbf{y} \leftarrow \mathbf{A}^\mathbf{t}\mathbf{x}$:

$$
\begin{bmatrix}
\mathbf{y}_1 \\
\vdots \\
\mathbf{y}_n \\
\end{bmatrix}
\leftarrow
\begin{bmatrix}
a_{11} & \cdots & a_{1n} \\
\vdots & \ddots & \vdots \\
a_{n1} & \cdots & a_{nn} \\
\end{bmatrix}
\begin{bmatrix}
\mathbf{x}_1 \\
\vdots \\
\mathbf{x}_n \\
\end{bmatrix}
$$

If $n$ is 0, no computation is performed; if $nd$ is 0, output vector $\mathbf{y}$ is set to zero, because matrix $\mathbf{A}$ 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. $\text{trans} \neq \text{‘N’}$ or $\text{‘T’}$
6. $nd < 0$
7. $\text{LA}(j) \leq -n$ or $\text{LA}(j) \geq n$, for any $j = 1, n$

**Examples**

**Example 1**

This example shows the matrix-vector product using $\text{trans} = \text{‘N’}$, which is computed for the following sparse matrix $\mathbf{A}$ of order 6. The matrix is stored in compressed-matrix storage mode in arrays $\mathbf{AD}$ and $\mathbf{LA}$ using the storage variation for general sparse matrices, storing all nonzero diagonals. Matrix $\mathbf{A}$ is:

```
| 4.0  0.0  7.0  0.0  0.0  0.0 |
| 3.0  4.0  0.0  2.0  0.0  0.0 |
| 0.0  2.0  4.0  0.0  4.0  0.0 |
| 0.0  0.0  7.0  4.0  0.0  1.0 |
| 1.0  0.0  0.0  3.0  4.0  0.0 |
| 1.0  1.0  0.0  0.0  3.0  4.0 |
```

376 IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and Reference
Example 2

This example shows the matrix-vector product using \texttt{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:

\[
\begin{matrix}
11.0 & 0.0 & 13.0 & 0.0 & 15.0 & 0.0 \\
0.0 & 22.0 & 0.0 & 24.0 & 0.0 & 26.0 \\
13.0 & 0.0 & 33.0 & 0.0 & 35.0 & 0.0 \\
0.0 & 24.0 & 0.0 & 44.0 & 0.0 & 46.0 \\
15.0 & 0.0 & 35.0 & 0.0 & 55.0 & 0.0 \\
0.0 & 26.0 & 0.0 & 46.0 & 0.0 & 66.0
\end{matrix}
\]

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 \texttt{trans} = 'T'. It is computed using the transpose of the following sparse matrix \( A \) of order 6, which is stored in compressed-matrix storage mode in arrays \( AD \) and \( LA \), using the storage variation for general sparse matrices, storing all nonzero diagonals. It uses the same matrix \( A \) as in Example 1.
Call Statement and Input:

```
IOPT N   ND  AD  LDA TRANS  LA   X   Y
CALL DSDMX( 0 , 6 , 5 , AD , 6 , 'T' , LA , X , Y )
```

\[ AD = \text{same as input AD in Example 1} \]
\[ LA = \text{same as input LA in Example 1} \]
\[ X = \text{same as input X in Example 1} \]

Output:

\[ Y = (21.0, 20.0, 47.0, 35.0, 50.0, 28.0) \]

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>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL DSMTM ( (m, nz, ac, ka, lda, n, nt, at, kt, ldt, aux, naux) )</td>
</tr>
<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 \( m \) 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 \( lda \geq m \).

\( n \)

is the number of columns in sparse matrix \( A \). Specified as: an integer; \( 0 \leq n \leq ldt \) and \( n \geq \) (maximum column index in KA).

\( nt \)

is the number of columns in output arrays AT and KT that are available for use. Specified as: 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; \( n = (\text{maximum column index in } KA) \).

\( nt \) 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 \( ldt \) by (at least) \( nt \) array, containing long-precision real numbers.

kt is the array, referred to as KT, containing the column numbers of the transposed matrix \( A^T \) elements, stored in the corresponding positions in array AT. Returned as: an \( ldt \) by (at least) \( nt \) array, containing 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 110.
5. If your sparse matrix is stored by rows, as defined in “Storage-by-Rows” on page 115, 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 1212, 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 49.
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 [*“89” on page 1258*]. 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: $y = A^T x$.

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 1212.

Error conditions

Resource Errors
Error 2015 is unrecoverable, $naux = 0$, and unable to allocate work area.

Computational Errors
None

Input-Argument Errors
1. $m, n < 0$
2. $lda, ldt < 1$
3. $lda < m$
4. $ldt < n$
5. $nz < 0$
6. $n$ is less than the maximum column index in KA.
7. $nt$ or $ldt$ are too small.
8. When the following two errors occur, arrays AT, KT, and AUX are overwritten:
   
   $naux < n$
   
   $nt \leq 0$

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:

| 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 |

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:

| 11.0 21.0 0.0 0.0 51.0 |
| 0.0 0.0 42.0 0.0 0.0 |
| 0.0 23.0 33.0 0.0 53.0 |
| 0.0 0.0 34.0 44.0 0.0 |
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 not accessed or modified by the subroutine. Column 4 and row 5 are accessed and modified. They are of no use in further computations and will not be used, because $NT = 3$ and $M = 4$.

Call Statement and Input:

```
M   NZ  AC   KA  LDA  N   NT  AT   KT   LDT  AUX  NAUX
|   |   |    |    |   |   |    |    |     |    |    |
CALL DSTM( 5 , 2 , AC , KA , 5 , 5 , 4 , AT , KT , 6 , AUX , 5 )
```

```
AC =
| 11.0  0.0 |
| 21.0  23.0 |
| 33.0  34.0 |
| 42.0  44.0 |
| 51.0  53.0 |
```

```
KA =
| 1  1 |
| 1  3 |
| 3  4 |
| 2  4 |
| 1  3 |
```

Output:

```
N        =  4
NT       =  3

AT =
| 11.0  21.0  51.0  0.0 |
| 42.0  0.0  0.0  0.0 |
| 33.0  23.0  53.0  0.0 |
| 34.0  44.0  0.0  0.0 |
| 0.0  0.0  0.0  0.0 |
| .  .  .  . |

KT =
| 1  2  5  1 |
| 4  1  1  1 |
| 3  2  5  1 |
| 3  4  1  1 |
| 1  1  1  1 |
| .  .  .  . |
```

Linear algebra subprograms 381
Chapter 9. Matrix operations subroutines

The matrix operations 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 [“42” on page 1255]. The matrix operation subroutines also include the commonly used matrix operations: addition, subtraction, multiplication, and transposition.

Table 102: 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>&quot;SGEADD, DGEADD, CGEADD, and ZGEADD (Matrix Addition for General Matrices or Their Transposes)&quot; on page 386</td>
</tr>
<tr>
<td>CGEADD</td>
<td>ZGEADD</td>
<td></td>
</tr>
<tr>
<td>SGESUB</td>
<td>DGESUB</td>
<td>&quot;SGESUB, DGESUB, CGESUB, and ZGESUB (Matrix Subtraction for General Matrices or Their Transposes)&quot; on page 392</td>
</tr>
<tr>
<td>CGESUB</td>
<td>ZGESUB</td>
<td></td>
</tr>
<tr>
<td>SGEMUL</td>
<td>DGEMUL</td>
<td>&quot;SGEMUL, DGEMUL, CGEMUL, and ZGEMUL (Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes)&quot; on page 398</td>
</tr>
<tr>
<td>CGEMUL</td>
<td>ZGEMUL</td>
<td></td>
</tr>
<tr>
<td>ZGEMUL§</td>
<td>DGEMLP§</td>
<td></td>
</tr>
<tr>
<td>SGEADD GPU</td>
<td>DGEADD GPU</td>
<td></td>
</tr>
<tr>
<td>SGEADD cblas_sgemm</td>
<td>DGEADD cblas_cgemm</td>
<td></td>
</tr>
<tr>
<td>SGEADD cblas_cgemm</td>
<td>DGEADD cblas_cgemm</td>
<td></td>
</tr>
<tr>
<td>SGEADD cblas_cgemm</td>
<td>DGEADD cblas_cgemm</td>
<td></td>
</tr>
<tr>
<td>Dgemms</td>
<td>ZGEMMS</td>
<td>&quot;SGEMMS, DGEMMS, CGEMMS, and ZGEMMS (Matrix Multiplication for General Matrices, Their Transposes, or Conjugate Transposes Using Winograd's Variation of Strassen's Algorithm)&quot; on page 408</td>
</tr>
<tr>
<td>CGEMMS</td>
<td>ZGEMMS</td>
<td></td>
</tr>
<tr>
<td>CGEMMS GPU</td>
<td>ZGEMMS GPU</td>
<td>&quot;SGEMM, DGEMM, CGEMM, and ZGEMM (Combined Matrix Multiplication and Addition for General Matrices, Their Transposes, or Conjugate Transposes)&quot; on page 413</td>
</tr>
<tr>
<td>cblas_sgemm</td>
<td>cblas_dgemm</td>
<td></td>
</tr>
<tr>
<td>cblas_zgemm</td>
<td>cblas_zgemm</td>
<td></td>
</tr>
<tr>
<td>SSYMM GPU</td>
<td>DSYMM GPU</td>
<td>&quot;SSYMM, DSYMM, CSYMM, ZSYMM, CHEMM, and ZHEMM (Matrix-Matrix Product Where One Matrix is Real or Complex Symmetric or Complex Hermitian)&quot; on page 422</td>
</tr>
<tr>
<td>CSYMM GPU</td>
<td>ZSYMM GPU</td>
<td></td>
</tr>
<tr>
<td>CHEMM GPU</td>
<td>ZHEMM GPU</td>
<td></td>
</tr>
<tr>
<td>cblas_ssymm</td>
<td>cblas_dsymm</td>
<td></td>
</tr>
<tr>
<td>cblas_zsymm</td>
<td>cblas_zsymm</td>
<td></td>
</tr>
<tr>
<td>cblas_chemm</td>
<td>cblas_zhemm</td>
<td></td>
</tr>
<tr>
<td>STRMM GPU</td>
<td>DTRMM GPU</td>
<td>&quot;STRMM, DTRMM, CTRMM, and ZTRMM (Triangular Matrix-Matrix Product)&quot; on page 430</td>
</tr>
<tr>
<td>CTRMM GPU</td>
<td>ZTRMM GPU</td>
<td></td>
</tr>
<tr>
<td>cblas_strmm</td>
<td>cblas_dtrmm</td>
<td></td>
</tr>
<tr>
<td>cblas_ctrmm</td>
<td>cblas_ztrmm</td>
<td></td>
</tr>
<tr>
<td>Short-Precision Subroutine</td>
<td>Long-Precision Subroutine</td>
<td>Descriptive Name and Location</td>
</tr>
<tr>
<td>----------------------------</td>
<td>----------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>STRSM*GPU</td>
<td>DTRSM*GPU</td>
<td>“STRSM, DTRSM, CTRSM, and ZTRSM (Solution of Triangular Systems of Equations with Multiple Right-Hand Sides)” on page 437</td>
</tr>
<tr>
<td>CTRSM*GPU</td>
<td>ZTRSM*GPU</td>
<td></td>
</tr>
<tr>
<td>cblas_strsm*</td>
<td>cblas_dtrsm*</td>
<td></td>
</tr>
<tr>
<td>cblas_ctrsm*</td>
<td>cblas_ztrsm*</td>
<td></td>
</tr>
<tr>
<td>SSYRK*GPU</td>
<td>DSYRK*GPU</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 445</td>
</tr>
<tr>
<td>CSYRK*GPU</td>
<td>ZSYRK*GPU</td>
<td></td>
</tr>
<tr>
<td>CHERK*GPU</td>
<td>ZHERK*GPU</td>
<td></td>
</tr>
<tr>
<td>cblas_ssyrk*</td>
<td>cblas_dsyrk*</td>
<td></td>
</tr>
<tr>
<td>cblas_csyrk*</td>
<td>cblas_zsyrk*</td>
<td></td>
</tr>
<tr>
<td>cblas_cherk*</td>
<td>cblas_zherk*</td>
<td></td>
</tr>
<tr>
<td>SSYRK2*GPU</td>
<td>DSYRK2*GPU</td>
<td>“SSYRK2K, DSYRK2K, CSYRK2K, ZSYRK2K, CHER2K, and ZHER2K (Rank-2K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix)” on page 451</td>
</tr>
<tr>
<td>CSYRK2*GPU</td>
<td>ZSYRK2*GPU</td>
<td></td>
</tr>
<tr>
<td>CHER2K*GPU</td>
<td>ZHER2K*GPU</td>
<td></td>
</tr>
<tr>
<td>cblas_ssyrk2k*</td>
<td>cblas_dsyrk2k*</td>
<td></td>
</tr>
<tr>
<td>cblas_csyrk2k*</td>
<td>cblas_zsyrk2k*</td>
<td></td>
</tr>
<tr>
<td>cblas_cher2k*</td>
<td>cblas_zher2k*</td>
<td></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 459</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>“SGETMO, DGETMO, CGETMO, ZGETMO, CGECMO, and ZGECMO (General Matrix Transpose or Conjugate Transpose [Out-of-Place])” on page 462</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.

> This subroutine can offload computations to the GPUs when the ESSL SMP CUDA Library is used.

---

**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^T B )</td>
<td>( A^T B )</td>
</tr>
<tr>
<td></td>
<td>( A^H B )</td>
</tr>
<tr>
<td>( AB^T )</td>
<td>( AB^T )</td>
</tr>
<tr>
<td>( A^T B^T )</td>
<td>( A^T B^T )</td>
</tr>
<tr>
<td></td>
<td>( A^H B^T )</td>
</tr>
<tr>
<td></td>
<td>( AB^H )</td>
</tr>
<tr>
<td></td>
<td>( A^T B^H )</td>
</tr>
<tr>
<td></td>
<td>( A^H 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 77.

**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 388 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 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 61.
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 [“42” on page 1255] and [“45” on page 1255].

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 $C$:

- $C \leftarrow A + B$
- $C \leftarrow A^T + B$
- $C \leftarrow A + B^T$
- $C \leftarrow A^T + B^T$

Table 103: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$, $B$, $C$</td>
<td></td>
</tr>
<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 34.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGEADD</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgeadd</td>
</tr>
</tbody>
</table>

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 103 on page 386, 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 103 on page 386, 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 103 on page 386.

**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 "Vector concepts" on page 71.

**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:

\[
\begin{align*}
c_{ij} &= a_{ij} + b_{ij} \quad \text{for } C \leftarrow A + B \\
c_{ij} &= a_{ij} + b_{ji} \quad \text{for } C \leftarrow A + B^T \\
c_{ij} &= a_{ji} + b_{ij} \quad \text{for } C \leftarrow A^T + B \\
c_{ij} &= a_{ji} + b_{ji} \quad \text{for } C \leftarrow A^T + B^T \\
\end{align*}
\]

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” on page 388 by using the following matrix identities:

\[
\begin{align*}
(A + B)^T &= A^T + B^T \\
(A + B^T)^T &= A^T + B \\
(A^T + B)^T &= A + B^T \\
(A^T + B^T)^T &= A + B \\
\end{align*}
\]

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. $\text{lda}, \text{ldb}, \text{ldc} \leq 0$
2. $m, n < 0$
3. $m > \text{ldc}$
4. $\text{transa}, \text{transb} \neq \text{'N'}$ or \text{'T'}$
5. \text{transa} = \text{'N'}$ and $m > \text{lda}$
6. $\text{transa} = \text{'T'}$ and $n > \text{ldc}$
7. $\text{transb} = \text{'N'}$ and $m > \text{ldb}$
8. $\text{transb} = \text{'T'}$ and $n > \text{ldb}$

Examples

Example 1

This example shows the computation $C \leftarrow A + B$, 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:

```
A  LDA TRANSA  B  LDB TRANSB  C  LDC  M   N
|   |    |     |   |    |     |   |   |   |
CALL SGEADD( A, 6, 'N', B, 4, 'N', C, 5, 4, 3 )
```

```
A    =  | 110000.0  120000.0  130000.0 |
| 210000.0  220000.0  230000.0 |
| 310000.0  320000.0  330000.0 |
| 410000.0  420000.0  430000.0 |
|       .         .         .   |
|       .         .         .   |
```

```
B    =  | 11.0  12.0  13.0 |
| 21.0  22.0  23.0 |
| 31.0  32.0  33.0 |
| 41.0  42.0  43.0 |
```

Output:

```
C    =  | 110011.0  120012.0  130013.0 |
| 210021.0  220022.0  230023.0 |
| 310031.0  320032.0  330033.0 |
| 410041.0  420042.0  430043.0 |
|       .         .         .   |
|       .         .         .   |
```

Example 2

This example shows the computation $C \leftarrow A^T + B$, where $A, B,$ and $C$ are the same size as arrays $A, B,$ and $C$, in which they are contained.

Call Statement and Input:

```
A  LDA TRANSA  B  LDB TRANSB  C  LDC  M   N
|   |    |     |   |    |     |   |   |   |
CALL SGEADD( A, 3, 'T', B, 4, 'N', C, 4, 4, 3 )
```

```
A    =  | 110000.0  120000.0  130000.0  140000.0 |
| 210000.0  220000.0  230000.0  240000.0 |
```
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:

```
$A =$ |
| 11.0  12.0  13.0 |
| 21.0  22.0  23.0 |
| 31.0  32.0  33.0 |
| 41.0  42.0  43.0 |
```

```
$B =$ |
| 110000.0  120000.0  130000.0 |
| 210000.0  220000.0  230000.0 |
| 310000.0  320000.0  330000.0 |
| 410000.0  420000.0  430000.0 |
```

```
$C =$ |
| 110011.0  120021.0  130031.0 |
| 210012.0  220022.0  230032.0 |
| 310013.0  320023.0  330033.0 |
| 410014.0  420024.0  430034.0 |
```

Example 4

This example shows how to produce the transpose of the result of the computation performed in Example 3, $C ← A + B^T$, which uses the calling sequence:

```
CALL SGEADD( A , 5 , 'N' , B , 3 , 'T' , C , 4 , 4 , 3 )
```

You instead code a calling sequence for $C^T ← A^T + B$, as shown below, where the resulting matrix $C^T$ in the output array $CT$ is the transpose of the matrix in the output array $C$ in Example 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 388.
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.

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:

```
CALL CGEADD( A, 6, 'N', B, 5, 'N', C, 5, 4, 3 )
```

**A** =

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0, 5.0</td>
<td>9.0, 2.0</td>
<td>1.0, 9.0</td>
</tr>
<tr>
<td>2.0, 4.0</td>
<td>8.0, 3.0</td>
<td>1.0, 8.0</td>
</tr>
<tr>
<td>3.0, 3.0</td>
<td>7.0, 5.0</td>
<td>1.0, 7.0</td>
</tr>
<tr>
<td>6.0, 6.0</td>
<td>3.0, 6.0</td>
<td>1.0, 4.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**B** =

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0, 8.0</td>
<td>2.0, 7.0</td>
<td>3.0, 2.0</td>
</tr>
<tr>
<td>4.0, 4.0</td>
<td>6.0, 8.0</td>
<td>6.0, 3.0</td>
</tr>
<tr>
<td>6.0, 2.0</td>
<td>4.0, 5.0</td>
<td>4.0, 5.0</td>
</tr>
<tr>
<td>7.0, 2.0</td>
<td>6.0, 4.0</td>
<td>1.0, 6.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**C** =

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0, 13.0</td>
<td>11.0, 9.0</td>
<td>4.0, 11.0</td>
</tr>
<tr>
<td>6.0, 8.0</td>
<td>14.0, 11.0</td>
<td>7.0, 11.0</td>
</tr>
<tr>
<td>9.0, 5.0</td>
<td>11.0, 10.0</td>
<td>5.0, 12.0</td>
</tr>
<tr>
<td>13.0, 8.0</td>
<td>9.0, 10.0</td>
<td>2.0, 10.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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 104: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A, B, C</strong></td>
<td></td>
</tr>
<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 34.
Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGESUB</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgesub</td>
</tr>
</tbody>
</table>

On Entry

**a**

is the matrix $A$, where:

If $\text{transa} = 'N'$, $A$ is used in the computation, and $A$ has $m$ rows and $n$ columns.

If $\text{transa} = 'T'$, $A^T$ 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 104 on page 392, where:

If $\text{transa} = 'N'$, its size must be $\text{lda}$ by (at least) $n$.

If $\text{transa} = 'T'$, its size must be $\text{lda}$ by (at least) $m$.

**lda**

is the leading dimension of the array specified for $a$.

Specified as: an integer; $\text{lda} > 0$ and:

If $\text{transa} = 'N'$, $\text{lda} \geq m$.

If $\text{transa} = 'T'$, $\text{lda} \geq n$.

**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.

Specified as: a single character; $\text{transa} = 'N'$ or 'T'.

**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.

**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 103 on page 386, where:

If $\text{transb} = 'N'$, its size must be $\text{ldb}$ by (at least) $n$.

If $\text{transb} = 'T'$, its size must be $\text{ldb}$ by (at least) $m$.

**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 $\text{transb} = 'T'$, $\text{ldb} \geq n$. 
**transb**
indicates the form of matrix $B$ to use in the computation, where:

If $\text{transb} = 'N'$, $B$ is used in the computation.
If $\text{transb} = 'T'$, $B^T$ is used in the computation.
Specified as: a single character; $\text{transb} = 'N' \text{ or } 'T'$.

**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 104 on page 392.

**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 $B$. 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 “Vector concepts” on page 71.

**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:

$$c_{ij} = a_{ij} - b_{ij} \quad \text{for } C \leftarrow A - B$$
$$c_{ij} = a_{ji} - b_{ji} \quad \text{for } C \leftarrow A - B^T$$
$$c_{ij} = a_{ij} - b_{ij} \quad \text{for } C \leftarrow A^T - B$$
$$c_{ij} = a_{ji} - b_{ji} \quad \text{for } C \leftarrow A^T - B^T$$

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” on page 394 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. \(\text{lda, ldb, ldc} \leq 0\)
2. \(m, n < 0\)
3. \(m > \text{ldc}\)
4. \(\text{transa, transb} \neq \text{'N'} \text{ or 'T'}\)
5. \(\text{transa} = \text{'N'} \text{ and } m > \text{lda}\)
6. \(\text{transa} = \text{'T'} \text{ and } n > \text{lda}\)
7. \(\text{transb} = \text{'N'} \text{ and } m > \text{ldb}\)
8. \(\text{transb} = \text{'T'} \text{ and } n > \text{ldb}\)

**Examples**

**Example 1**

This example shows the computation \(C\leftarrow A - B\), 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 SGESUB( A , 6 , 'N'  , B , 4 , 'N'  , C , 5 , 4 , 3 )
```

\[
A = \begin{bmatrix}
110000.0 & 120000.0 & 130000.0 \\
210000.0 & 220000.0 & 230000.0 \\
310000.0 & 320000.0 & 330000.0 \\
410000.0 & 420000.0 & 430000.0 \\
\vdots & \vdots & \vdots \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
-11.0 & -12.0 & -13.0 \\
-21.0 & -22.0 & -23.0 \\
-31.0 & -32.0 & -33.0 \\
-41.0 & -42.0 & -43.0 \\
\end{bmatrix}
\]

Output:

\[
C = \begin{bmatrix}
110011.0 & 120012.0 & 130013.0 \\
210021.0 & 220022.0 & 230023.0 \\
310031.0 & 320032.0 & 330033.0 \\
410041.0 & 420042.0 & 430043.0 \\
\vdots & \vdots & \vdots \\
\end{bmatrix}
\]
Example 2
This example shows the computation $C \leftarrow A^T - B$, where $A$, $B$, and $C$ are the same size as arrays $A$, $B$, and $C$, in which they are contained.

Call Statement and Input:

```
A LDA TRANSA B LDB TRANSB C LDC M N
CALL SGESUB( A , 3 , 'T' , B , 4 , 'N' , C , 4 , 4 , 3 )
```

<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>110000.0</td>
<td>120000.0</td>
<td>130000.0</td>
<td>140000.0</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>240000.0</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>340000.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output:

```
C = [ 110011.0  120012.0  130013.0 ]
   [ 210012.0  220012.0  230012.0 ]
   [ 310013.0  320013.0  330013.0 ]
   [ 410014.0  420014.0  430014.0 ]
```

Example 3
This example shows computation $C \leftarrow A - B^T$, where $A$ is contained in a larger array $A$, and $B$ and $C$ are the same size as arrays $B$ and $C$, in which they are contained.

Call Statement and Input:

```
A LDA TRANSA B LDB TRANSB C LDC M N
CALL SGESUB( A , 5 , 'N' , B , 3 , 'T' , C , 4 , 4 , 3 )
```

<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>110000.0</td>
<td>120000.0</td>
<td>130000.0</td>
<td></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>
<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>
<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>
<td></td>
</tr>
<tr>
<td>.         .         .</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output:

```
C = [ 110011.0  120012.0  130013.0 ]
   [ 210012.0  220012.0  230012.0 ]
   [ 310013.0  320013.0  330013.0 ]
   [ 410014.0  420014.0  430014.0 ]
```

Example 4
This example shows the computation $C \leftarrow A^T - B^T$, where $A$, $B$, and $C$ are the same size as the arrays $A$, $B$, and $C$, in which they are contained.
Example 5

This example shows how to produce the transpose of the result of the computation performed in Example 4, $\mathbf{C} \leftarrow \mathbf{A}^T \cdot \mathbf{B}^T$, which uses the calling sequence:

```fortran
```

You instead code a calling sequence for $\mathbf{C}^T \leftarrow \mathbf{A} - \mathbf{B}$, as shown below, where the resulting matrix $\mathbf{C}^T$ in the output array $\mathbf{CT}$ is the transpose of the matrix in the output array $\mathbf{C}$ in Example 4. Note that the array $\mathbf{CT}$ has dimensions large enough to receive the transposed matrix. For a description of all the matrix identities, see “Special Usage” on page 394.

Example 6

This example shows the computation $\mathbf{C} \leftarrow \mathbf{A} - \mathbf{B}$, where $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are contained in larger arrays $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$, respectively, and the arrays contain complex data.
Call Statement and Input:

```
CALL CGESUB( A, 6, 'N', B, 5, 'N', C, 5, 4, 3 )
```

\[
A = \begin{bmatrix}
(1.0, 5.0) & (9.0, 2.0) & (1.0, 9.0) \\
(2.0, 4.0) & (8.0, 3.0) & (1.0, 8.0) \\
(3.0, 3.0) & (7.0, 5.0) & (1.0, 7.0) \\
(6.0, 6.0) & (3.0, 6.0) & (1.0, 4.0) \\
\vdots & \vdots & \vdots \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
(1.0, 8.0) & (2.0, 7.0) & (3.0, 2.0) \\
(4.0, 4.0) & (6.0, 8.0) & (6.0, 3.0) \\
(6.0, 2.0) & (4.0, 5.0) & (4.0, 5.0) \\
(7.0, 2.0) & (6.0, 4.0) & (1.0, 6.0) \\
\vdots & \vdots & \vdots \\
\end{bmatrix}
\]

Output:

\[
C = \begin{bmatrix}
(0.0, -3.0) & (7.0, -5.0) & (-2.0, 7.0) \\
(-2.0, 0.0) & (2.0, -5.0) & (-5.0, 5.0) \\
(-3.0, 1.0) & (3.0, 0.0) & (-3.0, 2.0) \\
(-1.0, 4.0) & (-3.0, 2.0) & (0.0, -2.0) \\
\vdots & \vdots & \vdots \\
\end{bmatrix}
\]

**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 \( C \):

**SGEMUL and DGEMUL Combined Matrix Computations**

\[
\begin{align*}
C & \leftarrow AB \\
C & \leftarrow AB^T \\
C & \leftarrow A^T B \\
C & \leftarrow A^T B^T \\
\end{align*}
\]

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 \( C \):

**CGEMUL and ZGEMUL Combined Matrix Computations**

\[
\begin{align*}
C & \leftarrow AB \\
C & \leftarrow AB^H \\
C & \leftarrow A^T B \\
C & \leftarrow A^T B^H \\
C & \leftarrow A^H B \\
C & \leftarrow A^H B^H \\
\end{align*}
\]

**Table 105: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A, B, C )</td>
<td>SGEMUL</td>
</tr>
</tbody>
</table>

**398 IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and Reference**
Table 105: Data Types. (continued)

<table>
<thead>
<tr>
<th>Data Types</th>
<th></th>
</tr>
</thead>
<tbody>
<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 34.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGEMUL</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgemul</td>
</tr>
</tbody>
</table>

**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 105 on page 398, 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'`, $A^T$ 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 \( \text{transb} = 'T' \), \( B^T \) 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 105 on page 398, where:
- If \( \text{transb} = 'N' \), its size must be \( ldb \) by (at least) \( n \).
- If \( \text{transb} = 'T' \) 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 \( \text{transb} = 'N' \), \( ldb \geq m \).
- If \( \text{transb} = 'T' \) or \( 'C' \), \( ldb \geq n \).

**\( \text{transb} \)**

indicates the form of matrix \( B \) to use in the computation, where:
- If \( \text{transb} = 'N' \), \( B \) is used in the computation.
- If \( \text{transb} = 'T' \), \( B^T \) is used in the computation.
- If \( \text{transb} = 'C' \), \( B^H \) is used in the computation.
Specified as: a single character; \( \text{transb} = 'N' \) or \( 'T' \) for SGEMUL and DGEMUL; \( \text{transb} = 'N' \), \( 'T' \), 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; \( 0 \leq l \leq ldc \).

**\( m \)**

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 \( ldc \) by (at least) \( n \) numbers of the data type indicated in Table 105 on page 398.
Notes
1. All subroutines accept lowercase letters for the `transa` and `transb` arguments.
2. Matrix `C` must have no common elements with matrices `A` or `B`; otherwise, results are unpredictable. See “Vector concepts” on page 71.

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:
\[ c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj} \quad \text{for } C \leftarrow A \ B \]

\[ c_{ij} = \sum_{k=1}^{m} a_{ki} b_{kj} \quad \text{for } C \leftarrow A^T \ B \]

\[ c_{ij} = \sum_{k=1}^{m} \bar{a}_{ki} b_{kj} \quad \text{for } C \leftarrow A^H \ B \]

\[ c_{ij} = \sum_{k=1}^{m} a_{ik} b_{jk} \quad \text{for } C \leftarrow A \ B^T \]

\[ c_{ij} = \sum_{k=1}^{m} a_{ki} b_{jk} \quad \text{for } C \leftarrow A^T \ B^T \]

\[ c_{ij} = \sum_{k=1}^{m} \bar{a}_{ki} b_{jk} \quad \text{for } C \leftarrow A^H \ B^T \]

\[ c_{ij} = \sum_{k=1}^{m} a_{ik} \bar{b}_{jk} \quad \text{for } C \leftarrow A \ B^H \]

\[ c_{ij} = \sum_{k=1}^{m} a_{ki} \bar{b}_{jk} \quad \text{for } C \leftarrow A^T \ B^H \]

\[ c_{ij} = \sum_{k=1}^{m} \bar{a}_{ki} \bar{b}_{jk} \quad \text{for } C \leftarrow A^H \ B^H \]

for \( i = 1, l \) and \( j = 1, n \)

See reference [“48” on page 1255]. 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

\[(AB)^T = B^T A^T\]

Conjugate Transpose

\[(AB)^H = B^H A^H\]

\[(A^T B)^T = BA^T\]

\[(A^H B)^H = BA^H\]

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. \(lda, ldb, ldc \leq 0\)
2. \(l, m, n < 0\)
3. \(l > ldc\)
4. \(transa, transb \neq 'N'\) or \('T'\) for SGEMUL and DGEMUL
5. \(transa, transb \neq 'N', 'T',\) or \('C'\) for CGEMUL and ZGEMUL
6. \(transa = 'N'\) and \(l > lda\)
7. \(transa = 'T'\) or \('C'\) and \(m > lda\)
8. \(transb = 'N'\) and \(m > ldb\)
9. \(transb = 'T'\) or \('C'\) and \(n > ldb\)

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.

Call Statement and Input:

```
A   LDA  TRANSA   B  LDB  TRANSB   C  LDC  L  M  N
CALL  SGEMUL( A , 8 , 'N' , B , 6 , 'N' , C , 7 , 6 , 5 , 4 )
```

```
A =  | 1.0  2.0 -1.0 -1.0  4.0 |
   | 2.0  0.0  1.0  1.0  2.0 |
   | 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 -2.0  4.0 |
   | 1.0 -1.0  1.0 -3.0  1.0 |
   | .   .   .   .   .   .   |
```

```
B =  | 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 -3.0 -1.0 |
   | 4.0  2.0 -1.0  1.0 |
```
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:

```
CALL SGEMUL (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 \( CT \) is the transpose of the matrix in the output array \( C \) in Example 1. Note that the array \( CT \) has dimensions large enough to receive the transposed matrix. For a description of all the matrix identities, see “Special Usage” on page 402.

Call Statement and Input:

```
CALL SGEMUL( B, 6, 'T', A, 8, 'T', CT, 5, 4, CT, 5, 6 )
```

Output:

\[
\begin{bmatrix}
23.0 & 12.0 & -6.0 & 2.0 \\
-4.0 & -5.0 & 1.0 & 3.0 \\
3.0 & 0.0 & 1.0 & 4.0 \\
-3.0 & 5.0 & -2.0 & -10.0 \\
-5.0 & -7.0 & 4.0 & 4.0 \\
15.0 & 6.0 & -5.0 & 6.0
\end{bmatrix}
\]

\[
\begin{bmatrix}
-3.0 & 5.0 & -2.0 & -10.0 \\
-5.0 & -7.0 & 4.0 & 4.0 \\
15.0 & 6.0 & -5.0 & 6.0
\end{bmatrix}
\]

Example 3

This example shows the computation \( \text{C} \leftarrow A^T B \), 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:

\[
\begin{array}{cccccccc}
A & \text{LDA} & \text{TRANSA} & B & \text{LDB} & \text{TRANSB} & C & \text{LDC} & L & M & N \\
\end{array}
\]

\[
\text{CALL SGEMUL( } A, 4, 'T', B, 3, 'N', C, 5, 3, 3, 6 )
\]

\[
A = \begin{bmatrix}
1.0 & -3.0 & 2.0 \\
2.0 & 4.0 & 0.0 \\
1.0 & -1.0 & -1.0 \\
. & . & .
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
1.0 & -3.0 & 2.0 & 2.0 & -1.0 & 2.0 \\
2.0 & 4.0 & 0.0 & 0.0 & 1.0 & -2.0 \\
1.0 & -1.0 & -1.0 & -1.0 & -1.0 & 1.0 \\
. & . & . & . & . & .
\end{bmatrix}
\]

Output:

\[
C = \begin{bmatrix}
6.0 & 4.0 & 1.0 & 1.0 & 0.0 & -1.0 \\
4.0 & 26.0 & -5.0 & -5.0 & 8.0 & -15.0 \\
1.0 & -5.0 & 5.0 & 5.0 & -1.0 & 3.0 \\
. & . & . & . & . & . \\
. & . & . & . & . & . \\
. & . & . & . & . & .
\end{bmatrix}
\]

Example 4

This example shows how to produce the transpose of the result of the computation performed in Example 3, \( C^T \leftarrow A^T B \), which uses the calling sequence:

\[
\text{CALL \ SGEMUL( } A, 4, 'T', B, 3, 'N', C, 5, 3, 3, 6 )
\]

You instead code the calling sequence for \( C^T \leftarrow B^T A \), 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 402.

Call Statement and Input:

\[
\begin{array}{cccccccc}
A & \text{LDA} & \text{TRANSA} & B & \text{LDB} & \text{TRANSB} & C & \text{LDC} & L & M & N \\
\end{array}
\]

\[
\text{CALL SGEMUL( } B, 3, 'T', A, 4, 'N', CT, 8, 6, 3, 3 )
\]

\[
B = \begin{bmatrix}
1.0 & -3.0 & 2.0 & 2.0 & -1.0 & 2.0 \\
2.0 & 4.0 & 0.0 & 0.0 & 1.0 & -2.0 \\
1.0 & -1.0 & -1.0 & -1.0 & -1.0 & 1.0 \\
. & . & . & . & . & .
\end{bmatrix}
\]

\[
A = \begin{bmatrix}
1.0 & -3.0 & 2.0 \\
2.0 & 4.0 & 0.0 \\
1.0 & -1.0 & -1.0 \\
. & . & .
\end{bmatrix}
\]

Output:

\[
CT = \begin{bmatrix}
6.0 & 4.0 & 1.0 \\
4.0 & 26.0 & -5.0 \\
1.0 & -5.0 & 5.0 \\
0.0 & 8.0 & -1.0 \\
-1.0 & -15.0 & 3.0 \\
. & . & .
\end{bmatrix}
\]
Example 5

This example shows the computation $C \leftarrow A B^T$, 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:

```
A  LDA  TRANSA  B  LDB  TRANSB  C  LDC  L  M  N
CALL  SGEMUL( A , 4 , 'N' , B , 3 , 'T' , C , 5 , 3 , 2 , 3 )
```

```
A =
1.0  -3.0
2.0   4.0
1.0  -1.0
.
.
B =
1.0  -3.0
2.0   4.0
1.0  -1.0
```

Output:

```
C =
 10.0  -10.0   4.0
-10.0   20.0  -2.0
   4.0  -2.0   2.0
```

Example 6

This example shows the computation $C \leftarrow A^T B^T$, where $A$, $B$, and $C$ are the same size as the arrays $A$, $B$, and $C$ in which they are contained. (Based on the dimensions of the matrices, $A$ is actually a column vector, and $C$ is actually a row vector.)

Call Statement and Input:

```
A  LDA  TRANSA  B  LDB  TRANSB  C  LDC  L  M  N
CALL  SGEMUL( A , 3 , 'T' , B , 3 , 'T' , C , 1 , 1 , 3 , 3 )
```

```
A =
1.0
2.0
1.0
B =
1.0  -3.0   2.0
2.0   4.0   0.0
1.0  -1.0  -1.0
```

Output:

```
B =
-3.0  10.0  -2.0
```
Example 7

This example shows the computation $C \leftarrow A^T B$ using complex data, where $A$, $B$, and $C$ are contained in larger arrays $A$, $B$, and $C$, respectively.

Call Statement and Input:

```
A LDA TRANSA B LDB TRANSB C LDC L M N
CALL CGEMUL( A , 6 , 'T' , B , 7 , 'N' , C , 3 , 2 , 3 , 3 )
```

```
<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
CALL CGEMUL( A , 6 , 'T' , B , 7 , 'N' , C , 3 , 2 , 3 , 3 )
| (1.0, 2.0) | (3.0, 4.0) |
| (4.0, 6.0) | (7.0, 1.0) |
| (6.0, 3.0) | (2.0, 5.0) |
|   |   |   |   |   |   |   |   |   |   |
A = |
| (1.0, 9.0) | (2.0, 6.0) | (5.0, 6.0) |
| (2.0, 5.0) | (6.0, 2.0) | (6.0, 4.0) |
| (2.0, 6.0) | (5.0, 4.0) | (2.0, 6.0) |
|   |   |   |   |   |   |   |   |   |   |
B = |
| (1.0, 2.0) | (-3.0, 2.0) |
| (2.0, 6.0) | (4.0, 5.0) |
| (1.0, 2.0) | (-1.0, 8.0) |
|   |   |   |   |   |   |   |   |   |   |
| (1.0, 9.0) | (2.0, 6.0) | (5.0, 6.0) |
| (2.0, 5.0) | (6.0, 2.0) | (6.0, 4.0) |
| (2.0, 6.0) | (5.0, 4.0) | (2.0, 6.0) |
|   |   |   |   |   |   |   |   |   |   |
C = |
| (-45.0, 85.0) | (20.0, 93.0) | (-13.0, 110.0) |
| (-50.0, 90.0) | (12.0, 79.0) | (3.0, 94.0) |
|   |   |   |   |   |   |   |   |   |   |
Output:

```
| (-45.0, 85.0) | (20.0, 93.0) | (-13.0, 110.0) |
| (-50.0, 90.0) | (12.0, 79.0) | (3.0, 94.0) |
|   |   |   |   |   |   |   |   |   |   |
C = |
| (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) |
|   |   |   |   |   |   |   |   |   |   |
```

Example 8

This example shows the computation $C \leftarrow AB^H$ using complex data, where $A$ and $C$ are contained in larger arrays $A$ and $C$, respectively, and $B$ is the same size as the array $B$ in which it is contained.

Call Statement and Input:

```
A LDA TRANSA B LDB TRANSB C LDC L M N
CALL CGEMUL( A , 4 , 'N' , B , 3 , 'C' , C , 4 , 3 , 2 , 3 )
```

```
<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
CALL CGEMUL( A , 4 , 'N' , B , 3 , 'C' , C , 4 , 3 , 2 , 3 )
| (1.0, 2.0) | (-3.0, 2.0) |
| (2.0, 6.0) | (4.0, 5.0) |
| (1.0, 2.0) | (-1.0, 8.0) |
|   |   |   |   |   |   |   |   |   |   |
A = |
| (1.0, 3.0) | (-3.0, 2.0) |
| (2.0, 5.0) | (4.0, 6.0) |
| (1.0, 1.0) | (-1.0, 9.0) |
|   |   |   |   |   |   |   |   |   |   |
| (1.0, 3.0) | (-3.0, 2.0) |
| (2.0, 5.0) | (4.0, 6.0) |
| (1.0, 1.0) | (-1.0, 9.0) |
|   |   |   |   |   |   |   |   |   |   |
B = |
| (1.0, 2.0) | (-3.0, 2.0) |
| (2.0, 6.0) | (4.0, 5.0) |
| (1.0, 2.0) | (-1.0, 8.0) |
|   |   |   |   |   |   |   |   |   |   |
| (1.0, 3.0) | (-3.0, 2.0) |
| (2.0, 5.0) | (4.0, 6.0) |
| (1.0, 1.0) | (-1.0, 9.0) |
|   |   |   |   |   |   |   |   |   |   |
| (1.0, 3.0) | (-3.0, 2.0) |
| (2.0, 5.0) | (4.0, 6.0) |
| (1.0, 1.0) | (-1.0, 9.0) |
|   |   |   |   |   |   |   |   |   |   |
C = |
| (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) |
|   |   |   |   |   |   |   |   |   |   |
Output:

```
| (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) |
|   |   |   |   |   |   |   |   |   |   |
C = |
```

Matrix operations subroutines 407
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 $C$:

SGEMMS and DGEMMS Combined Matrix Computations

$C \leftarrow AB$

$C \leftarrow A^T B$

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 $C$:

CGEMMS and ZGEMMS Combined Matrix Computations

$C \leftarrow AB$

$C \leftarrow A^T B$

$C \leftarrow A^H B$

Table 106: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>aux</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A, B, C$</td>
<td>aux</td>
<td></td>
</tr>
<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

Language | Syntax
---|---
Fortran | CALL SGEMMS | DGEMMS | CGEMMS | ZGEMMS ($a$, lda, transa, b, ldb, transb, c, ldc, l, m, n, aux,iaux)

C and C++ | sgeams | dgemms | cgemms | zgemms ($a$, lda, transa, b, ldb, transb, c, ldc, l, m, n, aux,iaux);

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 106 on page 408, 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 $transa = 'T$', $A^T$ 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 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 $transb = 'T$', $B^T$ 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 106 on page 408, where:

- If $transb = 'N'$, its size must be $ldb$ by (at least) $n$.
- If $transb = 'T'$ or 'C', its size must be $ldb$ by (at least) $m$.

### $ldb$

is the leading dimension of the array specified for $b$.

Specified as: 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 $transb = 'T$', $B^T$ 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 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 ≤ l ≤ ldc.

m

has the following meaning, where:

If transa = 'N', it is the number of columns in matrix A.
If transa = 'T' or 'C', it is the number of rows in matrix A.

In addition:
If transb = 'N', it is the number of rows in matrix B.
If transb = 'T' or 'C', it is the number of columns in matrix B.

Specified as: an integer; m ≥ 0.

n

is the number of columns in matrix C.

Specified as: an integer; n ≥ 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 106 on page 408.

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 ≥ n, then nb1 ≥ (l)(n+20) and nb2 ≥ max[(n)(l), (m)(n+20)].
  If l < n, then nb1 ≥ (m)(n+20) and nb2 ≥ max[(n)(l), (l)(m+20)].

When this subroutine uses the direct method (_GEMUL), use naux ≥ 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 411.

On Return
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 106 on page 408.

Notes

1. There are two instances when these subroutines use the direct method (_GEMUL), rather than using
Strassen's algorithm:
   • When either or both of the input matrices are small
   • For CGEMMS and ZGEMMS, when input matrices $A$ and $B$ overlap
   In these instances when the direct method is used, the subroutine does not use auxiliary storage, and
   you can specify $naux = 0$.

2. For CGEMMS and ZGEMMS, one of the input matrices, $A$ or $B$, is rearranged during the computation and
   restored to its original form on return. Keep this in mind when diagnosing an abnormal termination.

3. All subroutines accept lowercase letters for the $transa$ and $transb$ arguments.

4. Matrix $C$ must have no common elements with matrices $A$ or $B$; otherwise, results are unpredictable.
   See “Vector concepts” on page 71.

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 49.

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 401.

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 ["18" on page 1254].) 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
obtained. For details about small matrices, see “Notes” on page 411.

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 402, 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. $l > ld$  
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:

```
<table>
<thead>
<tr>
<th>A</th>
<th>LDA</th>
<th>TRANSA</th>
<th>B</th>
<th>LDB</th>
<th>TRANSB</th>
<th>C</th>
<th>LDC</th>
<th>L</th>
<th>M</th>
<th>N</th>
<th>AUX</th>
<th>NAUX</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL SGEMMS( A , 8 , 'N' , B , 6 , 'N' , C , 7 , 6 , 5 , 4 , AUX , 0 )</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>
```

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:

{The call statement is shown here, followed by a table with the parameters and values.}

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 $C$:

**SGEMM and DGEMM Combined Matrix Computations**

$$
C \leftarrow \alpha AB + \beta C \\
C \leftarrow \alpha A^T B + \beta C \\
C \leftarrow \alpha A H B + \beta C
$$

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 $C$:

**CGEMM and ZGEMM Combined Matrix Computations**

$$
C \leftarrow \alpha AB + \beta C \\
C \leftarrow \alpha A^T B + \beta C \\
C \leftarrow \alpha A^H B + \beta C \\
C \leftarrow \alpha A^H B^H + \beta C
$$

Table 107: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$, $B$, $C$, $\alpha$, $\beta$</td>
<td>Subroutine</td>
</tr>
<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 34.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGEMM</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgemm</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_sgemm</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.

**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 ≤ l ≤ ldc.

**n**
is the number of columns in matrix \( C \).
Specified as: an integer; n ≥ 0.

**m**
has the following meaning, where:
- If transa = 'N', it is the number of columns in matrix \( A \).
If $\text{transa} = 'T' \text{ or } 'C'$, 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 $\text{transb} = 'T' \text{ or } 'C'$, 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 107 on page 413.

$a$ 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 107 on page 413, 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 107 on page 413, 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 $\beta$. 

Matrix operations subroutines 415
Specified as: a number of the data type indicated in Table 107 on page 413.

\( c \)
is the \( l \) by \( n \) matrix \( C \).

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 107 on page 413.

\( ldc \)
is the leading dimension of the array specified for \( c \).

Specified as: an integer; \( ldc > 0 \) and \( ldc \geq l \).

**On Return**

\( c \)
is the \( l \) by \( n \) matrix \( C \), containing the results of the computation. Returned as: an \( ldc \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 107 on page 413.

**Notes**

1. All subroutines accept lowercase letters for the \( transa \) and \( transb \) arguments.
2. For SGEMM and DGEMM, if you specify 'C' for the \( transa \) or \( transb \) argument, it is interpreted as though you specified 'T'.
3. Matrix \( C \) must have no common elements with matrices \( A \) or \( B \); otherwise, results are unpredictable. See “Vector concepts” on page 71.

**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:
\[
\begin{align*}
c_{ij} &= \left( \alpha \sum_{k=1}^{m} a_{ik} b_{kj} \right) c_{ij} + \beta c_{ij} \quad \text{for } C \leftarrow \alpha A B + \beta C \\
c_{ij} &= \left( \alpha \sum_{k=1}^{m} a_{ki} b_{kj} \right) c_{ij} + \beta c_{ij} \quad \text{for } C \leftarrow \alpha A^T B + \beta C \\
c_{ij} &= \left( \alpha \sum_{k=1}^{m} a_{ki} b_{kj} \right) c_{ij} + \beta c_{ij} \quad \text{for } C \leftarrow \alpha A^H B + \beta C \\
c_{ij} &= \left( \alpha \sum_{k=1}^{m} a_{ik} b_{jk} \right) c_{ij} + \beta c_{ij} \quad \text{for } C \leftarrow \alpha A B^T + \beta C \\
c_{ij} &= \left( \alpha \sum_{k=1}^{m} a_{ki} b_{jk} \right) c_{ij} + \beta c_{ij} \quad \text{for } C \leftarrow \alpha A^T B^T + \beta C \\
c_{ij} &= \left( \alpha \sum_{k=1}^{m} a_{ik} b_{jk} \right) c_{ij} + \beta c_{ij} \quad \text{for } C \leftarrow \alpha A^H B^T + \beta C \\
c_{ij} &= \left( \alpha \sum_{k=1}^{m} a_{ik} b_{jk} \right) c_{ij} + \beta c_{ij} \quad \text{for } C \leftarrow \alpha A^T B^H + \beta C \\
c_{ij} &= \left( \alpha \sum_{k=1}^{m} a_{ik} b_{jk} \right) c_{ij} + \beta c_{ij} \quad \text{for } C \leftarrow \alpha A^H B^H + \beta C \\
\end{align*}
\]

for \( i = 1, \ l \) and \( j = 1, n \)

See references [“42” on page 1255] and [“48” on page 1255]. In the following three cases, no computation is performed:

- \( l \) 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 402, 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. $\text{cblas\_layout} \neq \text{CblasRowMajor or CblasColMajor}$
2. $lda, ldb, ldc \leq 0$
3. $l, m, n < 0$
4. $l > ldc$
5. $\text{transa, transb} \neq 'N', 'T', \text{or 'C'}$
6. $\text{transa} = 'N' \text{ and } l > lda$
7. $\text{transa} = 'T' \text{ or 'C' and } m > lda$
8. $\text{cblas\_transa} \neq \text{CblasNoTrans, CblasTrans, or CblasConjTrans}$
9. $\text{cblas\_transa} = \text{CblasNoTrans and } l > lda$
10. $\text{cblas\_transa} = \text{CblasTrans, or CblasConjTrans and } m > lda$
11. $\text{transb} = 'N' \text{ and } m > ldb$
12. $\text{transb} = 'T' \text{ or 'C' and } n > ldb$
13. $\text{cblas\_transb} \neq \text{CblasNoTrans, CblasTrans, or CblasConjTrans}$
14. $\text{cblas\_transb} = \text{CblasNoTrans and } m > ldb$
15. $\text{cblas\_transb} = \text{CblasTrans, or CblasConjTrans and } n > ldb$

Examples

Example 1
This example shows the computation $C \leftarrow \alpha AB + \beta C$, where $A$, $B$, and $C$ are contained in larger arrays $A$, $B$, and $C$, respectively.

Call Statement and Input:

```plaintext
| TRANSA TRANSB L N M ALPHA A LDA B LDB BETA C LDC |
| CALL SGEMM( 'N' , 'N' , 6 , 4 , 5 , 1.0 , A , 8 , B , 6 , 2.0 , C , 7 ) |
| 1.0  2.0 -1.0 -1.0  4.0 |
```
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:

```
      |      |     |   |   |    |    |   |   |   |    |    |   |   |   |   |   |   |   |   |   |   |
CALL SGEMM( 'N'  , 'T'  , 3 , 3 , 2 , 1.0 , A , 4 , B , 3 , 2.0 , C , 5 )
```

```
A = | 1.0  -3.0 |
    | 2.0   4.0 |
    | 1.0  -1.0 |
    | .     .   |

B = | 1.0  -3.0 |
    | 2.0   4.0 |
    | 1.0  -1.0 |
    | .     .   |

C = | 0.5  0.5  0.5 |
    | 0.5  0.5  0.5 |
    | 0.5  0.5  0.5 |
    | .     .     . |
```

Output:

```
C = | 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 |
    | .     .     .     .   |
    | .     .     .     .   |
```
### Example 3

This example shows the computation $\mathbf{C} \leftarrow \alpha \mathbf{A}\mathbf{B} + \beta \mathbf{C}$ using complex data, where $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are contained in larger arrays, $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$, respectively.

#### Call Statement and Input:

```
CALL CGEMM( 'N' , 'N' , 6 , 2 , 3 , ALPHA , A , 8 , B , 4 , BETA , C , 8 )
```

| ALPHA     = (1.0, 0.0) |
| BETA      = (2.0, 0.0) |
| A         = | (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, 6.0) |
|           | (5.0, 1.0) (5.0, 1.0) (1.0, 5.0) |
|           | (6.0, 6.0) (3.0, 6.0) (1.0, 4.0) |
|           |   .           .           .      |
| B         = | (1.0, 8.0) (2.0, 7.0) |
|           | (4.0, 4.0) (6.0, 8.0) |
|           | (6.0, 2.0) (4.0, 5.0) |
|           |   .           .      |
| C         = | (0.5, 0.0) (0.5, 0.0) |
|           | (0.5, 0.0) (0.5, 0.0) |
|           | (0.5, 0.0) (0.5, 0.0) |
|           |   .           .      |

#### Output:

```
| (-27.0, 110.0) (-35.0, 141.0) |
| (-20.0, 119.0) (-43.0, 146.0) |
| (-27.0, 110.0) (-58.0, 131.0) |
```

### Example 4

This example shows how to obtain the conjugate transpose of $\mathbf{AB}^\mathbf{H}$. 
\[(AB^H)^H = A^H B^T = BA^H\]

This shows the conjugate transpose of the computation performed in Example 8 for CGEMUL, which uses the following calling sequence:

```plaintext
CALL CGEMUL( A, 4, 'N', B, 3, 'C', C, 4, 3, 2, 3 )
```

You instead code the calling sequence for \(C \leftarrow \beta C + \alpha BA^H\), 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 402.

Call Statement and Input:

```plaintext
CALL CGEMM( 'N', 'C', 3, 3, 2, ALPHA, B, 3, A, 3, BETA, C, 4 )
```

<table>
<thead>
<tr>
<th>ALPHA</th>
<th>BETA</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1.0, 0.0)</td>
<td>(0.0, 0.0)</td>
</tr>
</tbody>
</table>

\[
B = \begin{bmatrix}
1.0, 3.0 & -3.0, 2.0 \\
2.0, 5.0 & 4.0, 6.0 \\
1.0, 1.0 & -1.0, 9.0 \\
\end{bmatrix}
\]

\[
A = \begin{bmatrix}
1.0, 2.0 & -3.0, 2.0 \\
2.0, 6.0 & 4.0, 5.0 \\
1.0, 2.0 & -1.0, 8.0 \\
\end{bmatrix}
\]

\(C\) = (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 \(C \leftarrow \alpha A^T \beta B^H + \beta C\) 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:

```plaintext
CALL CGEMM( 'T', 'C', 1, 3, 3, ALPHA, A, 3, B, 3, BETA, C, 4 )
```

<table>
<thead>
<tr>
<th>ALPHA</th>
<th>BETA</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1.0, 1.0)</td>
<td>(0.0, 0.0)</td>
</tr>
</tbody>
</table>

\[
A = \begin{bmatrix}
1.0, 2.0 \\
2.0, 5.0 \\
\end{bmatrix}
\]
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 108: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha, A, B, \beta, C$</td>
<td>SSYMM</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>DSYMM</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>CSYMM and CHEMM</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>ZSYMM and ZHEMM</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td></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 34.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SSYMM</td>
</tr>
<tr>
<td>C and C++</td>
<td>ssymm</td>
</tr>
<tr>
<td>Language</td>
<td>Syntax</td>
</tr>
<tr>
<td>----------</td>
<td>--------</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_ssymm</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 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` = CblasLeft, `A` is to the left of `B`, resulting in equation 1.
  
  If `cblas_side` = 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` = CblasUpper, `A` is stored in upper storage mode.
  
  If `cblas_uplo` = 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 ≤ m ≤ ldb, m ≤ ldc`, and:
  
  If `side` = 'L', `m ≤ 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 \( \text{side} = 'R', n \leq \text{lda} \).

\textbf{alpha}

is the scalar \( \alpha \).

Specified as: a number of the data type indicated in Table 108 on page 422.

\textbf{a}

is the real symmetric, complex symmetric, or complex Hermitian matrix \( A \), where:

If \( \text{side} = 'L', A \) is order \( m \).

If \( \text{side} = 'R', A \) is order \( n \).

and where it is stored as follows:

If \( \text{uplo} = 'U', A \) is stored in upper storage mode.

If \( \text{uplo} = 'L', A \) is stored in lower storage mode.

Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 108 on page 422, 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) \( n \).

\textbf{lda}

is the leading dimension of the array specified for \( a \).

Specified as: an integer; \( lda > 0 \) and:

If \( \text{side} = 'L', lda \geq m \).

If \( \text{side} = 'R', lda \geq n \).

\textbf{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 108 on page 422.

\textbf{ldb}

is the leading dimension of the array specified for \( b \).

Specified as: an integer; \( ldb > 0 \) and \( ldb \geq m \).

\textbf{beta}

is the scalar \( \beta \).

Specified as: a number of the data type indicated in Table 108 on page 422.

\textbf{c}

is the \( m \) by \( n \) rectangular matrix \( C \).

Specified as: an \( ldc \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 108 on page 422.

\textbf{ldc}

is the leading dimension of the array specified for \( c \).

Specified as: an integer; \( ldc > 0 \) and \( ldc \geq m \).

\textbf{On Return}

\textbf{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 108 on page 422.
Notes

1. These subroutines accept lowercase letters for the side and uplo arguments.
2. Matrices A, B, and C must have no common elements; otherwise, results are unpredictable.
3. If matrix A is upper triangular (uplo = 'U'), these subroutines use only the data in the upper triangular portion of the array. If matrix A is lower triangular, (uplo = 'L'), these subroutines use only the data in the lower triangular portion of the array. In each case, the other portion of the array is altered during the computation, but restored before exit.
4. The imaginary parts of the diagonal elements of a complex Hermitian matrix A are assumed to be zero, so you do not have to set these values.
5. For a description of how symmetric matrices are stored in upper and lower storage mode, see “Symmetric Matrix” on page 80. For a description of how complex Hermitian matrices are stored in upper and lower storage mode, see “Complex Hermitian Matrix” on page 84.

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 [“42” on page 1255] and [“48” on page 1255]. 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. cblas_layout ≠ 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. \( \text{side} \neq 'L' \) or 'R'
8. \( \text{side} = 'L' \) and \( m > \text{lda} \)
9. \( \text{side} = 'R' \) and \( n > \text{lda} \)
10. \( \text{cblas\_side} \neq \text{CblasLeft or CblasRight} \)
11. \( \text{cblas\_side} = \text{CblasLeft and} \ m > \text{lda} \)
12. \( \text{cblas\_side} = \text{CblasRight and} \ n > \text{lda} \)
13. \( \text{uplo} \neq 'L' \text{ or } 'U' \)
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:

```
CALL SSYMM( 'L', 'U', 5, 4, 2.0, A, 8, B, 6, 1.0, C, 5 )
```

\[
\begin{bmatrix}
1.0 & 2.0 & -1.0 & -1.0 & 4.0 \\
.0 & 1.0 & 1.0 & -1.0 \\
. & -1.0 & 1.0 & 2.0 \\
. & . & . & 2.0 & 0.0 \\
. & . & . & . & -1.0 \\
. & . & . & . & . \\
. & . & . & . & . \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
1.0 & -1.0 & 0.0 & 2.0 \\
2.0 & 2.0 & -1.0 & -2.0 \\
1.0 & 0.0 & -1.0 & 1.0 \\
-3.0 & -1.0 & 1.0 & -1.0 \\
4.0 & 2.0 & -1.0 & 1.0 \\
. & . & . & . \\
. & . & . & . \\
\end{bmatrix}
\]

\[
C = \begin{bmatrix}
23.0 & 12.0 & -6.0 & 2.0 \\
-4.0 & -5.0 & 1.0 & 3.0 \\
5.0 & 6.0 & -1.0 & -4.0 \\
-4.0 & 1.0 & 0.0 & -5.0 \\
8.0 & -4.0 & -2.0 & 13.0 \\
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
69.0 & 36.0 & -18.0 & 6.0 \\
-12.0 & -15.0 & 3.0 & 9.0 \\
15.0 & 18.0 & -3.0 & -12.0 \\
-12.0 & 3.0 & 0.0 & -15.0 \\
8.0 & -20.0 & -2.0 & 35.0 \\
\end{bmatrix}
\]

**Example 2**

This example shows the computation \( C \leftarrow \alpha AB + \beta C \), where \( A \) is a real symmetric matrix of order 3, stored in lower storage mode, and \( B \) and \( C \) are 3 by 6 rectangular matrices.
### Example 3

This example shows the computation $C = \alpha BA + \beta C$, where $A$ is a real symmetric matrix of order 3, stored in upper storage mode, and $B$ and $C$ are 2 by 3 rectangular matrices.
Example 4

This example shows the computation $C \leftarrow \alpha BA + \beta C$, 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:

```plaintext
CALL SSYM( 'R' , 'L' , 3 , 3 , -1.0  , A , 3 , B , 3 , 1.0 , C , 3 )
```

Output:

```
 C = [  39.0  -54.0   30.0 ]
     [ -33.0   33.0  -12.0 ]
     [    .      .      .  ]
```

Example 5

This example shows the computation $C \leftarrow \alpha BA + \beta C$, where $A$ is a complex symmetric matrix of order 3, stored in upper storage mode, and $B$ and $C$ are 2 by 3 rectangular matrices.

Call Statement and Input:

```plaintext
CALL CSYM( 'R' , 'U' , 2 , 3 , ALPHA , A , 4 , B , 3 , BETA , C , 5 )
```

Output:

```
 A = [ (1.0, 5.0) (-3.0, 2.0) (1.0, 6.0) ]
      [ .            (4.0, 5.0) (-1.0, 4.0) ]
      [ .                (2.0, 5.0) ]
 B = [ (1.0, -3.0) 2.0 ]
      [ 2.0  4.0  0.0 ]
      [ 1.0  -1.0 -1.0 ]
 C = [ 1.0  -3.0   2.0 ]
      [ -3.0  10.0  -9.0 ]
      [ -2.0   8.0   0.0 ]
```
Example 6

This example shows the computation $C \leftarrow \alpha BA + \beta C$, where $A$ is a complex Hermitian matrix of order 3, stored in lower storage mode, and $B$ and $C$ are 3 by 3 square matrices.

Note: The imaginary parts of the diagonal elements of a complex Hermitian matrix are assumed to be zero, so you do not have to set these values.

Call Statement and Input:

```
SIDE  UPLO  M   N   ALPHA   A  LDA  B  LDB  BETA   C  LDC
|     |    |   |     |     |   |   |   |    |     |   |
CALL CHEMM( 'R' , 'L' , 2 , 3 , ALPHA , A , 4 , B , 3 , BETA , C , 5 )
```

```
ALPHA    =  (2.0, 3.0)
BETA     =  (1.0, 6.0)
```

```
A      =  
\begin{bmatrix}
  (1.0, 1.0) & (3.0, 2.0) & (4.0, 3.0) \\
  (3.0, 2.0) & (4.0, 3.0) & (2.0, 4.0) \\
  (1.0, 4.0) & (2.0, 4.0) & \cdot & \cdot & \cdot & \cdot \\
\end{bmatrix}
```

```
B      =  
\begin{bmatrix}
  (1.0, 1.0) & (3.0, 2.0) & (3.0, 3.0) \\
  (3.0, 2.0) & (4.0, 5.0) & (-1.0, 4.0) \\
\end{bmatrix}
```

```
C      =  
\begin{bmatrix}
  (13.0, 6.0) & (-18.0, 6.0) & (10.0, 7.0) \\
  (-11.0, 8.0) & (11.0, 1.0) & (-4.0, 2.0) \\
\end{bmatrix}
```

Output:

```
C      =  
\begin{bmatrix}
  (-96.0, 72.0) & (-141.0, -226.0) & (-112.0, 38.0) \\
  (-230.0, -269.0) & (-133.0, -23.0) & (-272.0, -198.0) \\
\end{bmatrix}
```

```
C      =  
\begin{bmatrix}
  (-137.0, 17.0) & (-158.0, -102.0) & (-39.0, 141.0) \\
  (-154.0, -77.0) & (-63.0, 186.0) & (159.0, 104.0) \\
\end{bmatrix}
```
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:

**Matrix-Matrix Products Computed by STRMM and DTRMM**

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:

**Matrix-Matrix Products Computed by CTRMM and ZTRMM**

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 109: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A, B, \alpha )</td>
<td></td>
</tr>
<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 34.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL STRMM</td>
</tr>
<tr>
<td>C and C++</td>
<td>strm</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_strmm</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.

\textbf{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 \textit{side} = 'L', \( A \) is to the left of \( B \) in the equation, resulting in either equation 1, 2, or 5.
- If \textit{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'.

\textbf{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 \textit{cblas_side} = CblasLeft, \( A \) is to the left of \( B \) in the equation, resulting in either equation 1, 2, or 5.
- If \textit{cblas_side} = 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 CBLAS_SIDE. It must be CblasLeft or CblasRight.

\textbf{uplo} indicates whether matrix \( A \) is an upper or lower triangular matrix, where:

- If \textit{uplo} = 'U', \( A \) is an upper triangular matrix.
- If \textit{uplo} = 'L', \( A \) is a lower triangular matrix.

Specified as: a single character. It must be 'U' or 'L'.

\textbf{cblas_uplo} indicates whether matrix \( A \) is an upper or lower triangular matrix, where:

- If \textit{cblas_uplo} = CblasUpper, \( A \) is an upper triangular matrix.
- If \textit{cblas_uplo} = CblasLower, \( A \) is a lower triangular matrix.

Specified as: an object of enumerated type CBLAS_UPLO. It must be CblasUpper or CblasLower.

\textbf{transa} indicates the form of matrix \( A \) to use in the computation, where:

- If \textit{transa} = 'N', \( A \) is used in the computation, resulting in either equation 1 or 3.
- If \textit{transa} = 'T', \( A^T \) is used in the computation, resulting in either equation 2 or 4.
- If \textit{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'.

\textbf{cblas_transa} indicates the form of matrix \( A \) to use in the computation, where:

- If \textit{cblas_transa} = CblasNoTrans, \( A \) is used in the computation, resulting in either equation 1 or 3.
- If \textit{cblas_transa} = CblasTrans, \( A^T \) is used in the computation, resulting in either equation 2 or 4.
- If \textit{cblas_transa} = CblasConjTrans, \( A^H \) is used in the computation, resulting in either equation 5 or 6.

Specified as: an object of enumerated type CBLAS_TRANSPOSE. It must be CblasNoTrans, CblasTrans, or CblasConjTrans.

\textbf{diag} indicates the characteristics of the diagonal of matrix \( A \), where:

- If \textit{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 \text{CblasNonUnit} or \text{CblasUnit}.

$m$
is the number of rows in rectangular matrix $B$, and:
If $\text{side} = 'L'$, $m$ is the order of triangular matrix $A$.
Specified as: an integer, where:
If $\text{side} = 'L'$, $0 \leq m \leq \text{lda}$ and $m \leq \text{ldb}$.
If $\text{side} = 'R'$, $0 \leq m \leq \text{ldb}$.

$n$
is the number of columns in rectangular matrix $B$, and:
If $\text{side} = 'R'$, $n$ is the order of triangular matrix $A$.
Specified as: an integer; $n \geq 0$ and:
If $\text{side} = 'R'$, $n \leq \text{lda}$.

$\alpha$
is the scalar $\alpha$.
Specified as: a number of the data type indicated in Table 109 on page 430.

$a$
is the triangular matrix $A$, of which only the upper or lower triangular portion is used, where:
If $\text{side} = 'L'$, $A$ is order $m$.
If $\text{side} = 'R'$, $A$ 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 109 on page 430, 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 109 on page 430.

$\text{ldb}$
is the leading dimension of the array specified for $b$. 
Specified as: an integer; $ldb > 0$ and $ldb \geq m$.

**On Return**

$b$

is the $m$ by $n$ matrix $B$, containing the results of the computation. Returned as: an $ldb$ by (at least) $n$ array, containing numbers of the data type indicated in Table 109 on page 430.

**Notes**

1. These subroutines accept lowercase letters for the $side$, $uplo$, $transa$, and $diag$ arguments.
2. For STRMM and DTRMM, if you specify 'C' for the $transa$ argument, it is interpreted as though you specified 'T'.
3. Matrices $A$ and $B$ must have no common elements; otherwise, results are unpredictable.
4. ESSL assumes certain values in your array for parts of a triangular matrix. As a result, you do not have to set these values. For unit triangular matrices, the elements of the diagonal are assumed to be 1.0 for real matrices and (1.0, 0.0) for complex matrices. When using upper- or lower-triangular storage, the unreferenced elements in the lower and upper triangular part, respectively, are assumed to be zero.
5. For a description of triangular matrices and how they are stored, see “Triangular Matrix” on page 88.

**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. $B -\!- \alpha AB$
2. $B -\!- \alpha \text{A}^T \text{B}$
3. $B -\!- \alpha \text{A}^H \text{B}$ (for CTRMM and ZTRMM only)
   
   where:
   
   - $\alpha$ is a scalar.
   - $A$ is a triangular matrix of order $m$.
   - $B$ is an $m$ by $n$ rectangular matrix.
4. $B -\!- \alpha \text{BA}$
5. $B -\!- \alpha \text{BA}^T$
6. $B -\!- \alpha \text{BA}^H$ (for CTRMM and ZTRMM only)

   where:

   - $\alpha$ is a scalar.
   - $A$ is a triangular matrix of order $n$.
   - $B$ is an $m$ by $n$ rectangular matrix.

See references [“42” on page 1255] and [“48” on page 1255]. 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. $cblas\_layout \neq \text{CblasRowMajor}$ or $\text{CblasColMajor}$

Matrix operations subroutines 433
2. $m < 0$
3. $m > ldb$
4. $n < 0$
5. $lda, ldb \leq 0$
6. $side \neq 'L' \text{ or } 'R'$
7. $side = 'L' \text{ and } m > lda$
8. $side = 'R' \text{ and } n > lda$
9. $cblas\_side \neq \text{CblasLeft or CblasRight}$
10. $cblas\_side = \text{CblasLeft and } m > lda$
11. $cblas\_side = \text{CblasRight and } n > lda$
12. $uplo \neq 'L' \text{ or } 'U'$
13. $cblas\_uplo \neq \text{CblasLower or CblasUpper}$
14. $transa \neq 'T', 'N', \text{ or } 'C'$
15. $cblas\_transa \neq \text{CblasNoTrans, CblasTrans, or CblasConjTrans}$
16. $diag \neq 'N' \text{ or } 'U'$
17. $cblas\_diag \neq \text{CblasNonUnit or CblasUnit}$

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 )
```

```
\[
A = \begin{bmatrix}
    3.0 & -1.0 & 2.0 & 2.0 & 1.0 \\
    -2.0 & 4.0 & -1.0 & 3.0 & \\
    . & -3.0 & 0.0 & 2.0 & \\
    . & . & 4.0 & -2.0 & \\
    . & . & . & 1.0 & \\
    . & . & . & . & \\
\end{bmatrix}
\]
```

```
\[
B = \begin{bmatrix}
    2.0 & 3.0 & 1.0 \\
    5.0 & 5.0 & 4.0 \\
    0.0 & 1.0 & 2.0 \\
    3.0 & 1.0 & -3.0 \\
    -1.0 & 2.0 & 1.0 \\
    . & . & . \\
\end{bmatrix}
\]
```

Output:

```
\[
B = \begin{bmatrix}
    6.0 & 10.0 & -2.0 \\
    -16.0 & -1.0 & 6.0 \\
    -2.0 & 1.0 & -4.0 \\
    14.0 & 0.0 & -14.0 \\
    -1.0 & 2.0 & 1.0 \\
    . & . & . \\
\end{bmatrix}
\]
```
Example 2

This example shows the computation $B \leftarrow \alpha A^T B$, where $A$ is a 5 by 5 upper triangular matrix that is not unit triangular, and $B$ is a 5 by 4 rectangular matrix.

Call Statement and Input:

```
SIDE  UPLO TRANSA DIAG M N  ALPHA A  LDA B  LDB
|     |     |      |    |   |     |     |   |   |   |
CALL STRMM( 'L' , 'U' , 'T'  , 'N' , 5 , 4 ,  1.0  , A , 7 , B , 6 )
```

\[
A = \begin{bmatrix}
-1.0 & -4.0 & -2.0 & 2.0 & 3.0 \\
 10.0 & 2.0 & 2.0 & 2.0 & 1.0 \\
 3.0 & -3.0 & -1.0 & 4.0 & 0.0 \\
 2.0 & 2.0 & 2.0 & 2.0 & -2.0 \\
 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
1.0 & 2.0 & 3.0 & 4.0 \\
3.0 & 3.0 & -1.0 & 2.0 \\
-2.0 & -1.0 & 0.0 & 1.0 \\
2.0 & 2.0 & 2.0 & 2.0 \\
\end{bmatrix}
\]

Output:

\[
B = \begin{bmatrix}
-1.0 & -2.0 & -3.0 & -4.0 \\
2.0 & -2.0 & -14.0 & -12.0 \\
10.0 & 5.0 & -8.0 & -7.0 \\
14.0 & 15.0 & 1.0 & 8.0 \\
-3.0 & 4.0 & 3.0 & 16.0 \\
\end{bmatrix}
\]

Example 3

This example shows the computation $B \leftarrow \alpha BA$, where $A$ is a 5 by 5 lower triangular matrix that is not unit triangular, and $B$ is a 3 by 5 rectangular matrix.

Call Statement and Input:

```
SIDE  UPLO TRANSA DIAG M N  ALPHA A  LDA B  LDB
|     |     |      |    |   |     |     |   |   |   |
CALL STRMM( 'R' , 'L' , 'N'  , 'N' , 3 , 5 ,  1.0  , A , 7 , B , 4 )
```

\[
A = \begin{bmatrix}
2.0 & . & . & . & . \\
2.0 & 3.0 & . & . & . \\
2.0 & 1.0 & 1.0 & . & . \\
0.0 & 3.0 & 0.0 & -2.0 & . \\
2.0 & 4.0 & -1.0 & 2.0 & -1.0 \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
3.0 & 4.0 & -1.0 & -1.0 & -1.0 \\
2.0 & 1.0 & -1.0 & 0.0 & 3.0 \\
2.0 & -1.0 & -3.0 & 0.0 & 2.0 \\
\end{bmatrix}
\]
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:

```
SIDE  UPLO TRANSA  DIAG  M   N    ALPHA  A  LDA  B  LDB
|     |     |      |    |   |     |     |   |   |   |
CALL STRMM( 'R' , 'U' , 'N' , 'U' , 1 , 6 ,  1.0  , A , 7 , B , 2 )
```

**Output:**

```
B = [ 10.0  4.0  0.0  0.0  1.0 1.0  
     -10.0 14.0 -4.0  6.0 -3.0 2.0  
     -8.0  2.0 -5.0  4.0 -2.0 -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:

```
SIDE  UPLO TRANSA  DIAG  M   N    ALPHA   A  LDA  B  LDB
|     |     |      |    |   |      |     |   |   |   |
CALL CTRMM( 'L' , 'U' , 'C' , 'N' , 5 , 1 ,  ALPHA , A , 6 , B , 6 )
```

**Output:**

```
B = [ 1.0  4.0 -2.0 10.0 2.0 6.0  ]
```

**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:

```
SIDE  UPLO TRANSA  DIAG  M   N    ALPHA  A  LDA  B  LDB
|     |     |      |    |   |     |     |   |   |   |
CALL STRMM( 'R' , 'U' , 'N' , 'U' , 1 , 6 ,  1.0  , A , 7 , B , 2 )
```

**Output:**

```
B = [ 10.0  4.0  0.0  0.0  1.0 1.0  
     -10.0 14.0 -4.0  6.0 -3.0 2.0  
     -8.0  2.0 -5.0  4.0 -2.0 -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:

```
SIDE  UPLO TRANSA  DIAG  M   N    ALPHA   A  LDA  B  LDB
|     |     |      |    |   |      |     |   |   |   |
CALL CTRMM( 'L' , 'U' , 'C' , 'N' , 5 , 1 ,  ALPHA , A , 6 , B , 6 )
```

**Output:**

```
B = [ 1.0  4.0 -2.0 10.0 2.0 6.0  ]
```
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:

<table>
<thead>
<tr>
<th>Solution</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $B\leftarrow \alpha(A^{-1})B$</td>
<td>$AX = \alpha B$</td>
</tr>
<tr>
<td>2. $B\leftarrow \alpha(A^{-\top})B$</td>
<td>$A^\top X = \alpha B$</td>
</tr>
<tr>
<td>3. $B\leftarrow \alpha B(A^{-1})$</td>
<td>$XA = \alpha B$</td>
</tr>
<tr>
<td>4. $B\leftarrow \alpha B(A^{-\top})$</td>
<td>$XA^\top = \alpha B$</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Solution</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $B\leftarrow \alpha(A^{-1})B$</td>
<td>$AX = \alpha B$</td>
</tr>
<tr>
<td>2. $B\leftarrow \alpha(A^{-\top})B$</td>
<td>$A^\top X = \alpha B$</td>
</tr>
<tr>
<td>3. $B\leftarrow \alpha B(A^{-1})$</td>
<td>$XA = \alpha B$</td>
</tr>
<tr>
<td>4. $B\leftarrow \alpha B(A^{-\top})$</td>
<td>$XA^\top = \alpha B$</td>
</tr>
<tr>
<td>5. $B\leftarrow \alpha(A^{-H})B$</td>
<td>$A^H X = \alpha B$</td>
</tr>
<tr>
<td>6. $B\leftarrow \alpha B(A^{-\top H})$</td>
<td>$XA^H = \alpha B$</td>
</tr>
</tbody>
</table>

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 110: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$, $B$, $\alpha$</td>
<td>STRSM</td>
</tr>
<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>
</tbody>
</table>
Table 110: Data Types. (continued)

<table>
<thead>
<tr>
<th>Data Types</th>
<th></th>
</tr>
</thead>
<tbody>
<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 34.

### Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL STRSM</td>
</tr>
<tr>
<td>C and C++</td>
<td>strsm</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_strsm</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 $cblas\_uplo = \text{CblasLower}$, $A$ 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 $cblas\_transa = \text{CblasNoTrans}$, $A$ is used, resulting in solution 1 or 3.
If $cblas\_transa = \text{CblasTrans}$, $A^T$ is used, resulting in solution 2 or 4.
If $cblas\_transa = \text{CblasConjTrans}$, $A^H$ 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 $A$, 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.

**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 $side = 'L'$, $0 \leq m \leq lda$ and $m \leq ldb$.
If $side = 'R'$, $0 \leq m \leq ldb$.

**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; $n \geq 0$, and:
If $side = 'R$', $n \leq lda$.

**alpha**
is the scalar $\alpha$. Specified as: a number of the data type indicated in Table 110 on page 437.

**a**
is the triangular matrix $A$, of which only the upper or lower triangular portion is used, where:
If $side = 'L$’, $A$ is order $m$.
If $side = 'R$’, $A$ is order $n$.
Specified as: a two-dimensional array, containing numbers of the data type indicated in Table 110 on page 437, 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, 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 110 on page 437.

**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 110 on page 437.

**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 88.

**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. \( B \leftarrow \alpha (A^{-1}) B \)
2. \( B \leftarrow \alpha (A^{-T}) B \)
3. \( B \leftarrow \alpha B(A^{-1}) \)
4. \( B \leftarrow \alpha B(A^{-T}) \)
5. \( B \leftarrow \alpha (A^{-H}) B \) (only for CTRSM and ZTRSM)
6. $B \leftarrow \alpha B(A^{-1})$ (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 $\text{side} = 'L'$, it has order $m$, and equation 1, 2, or 5 is performed.
If $\text{side} = 'R'$, it has order $n$, and equation 3, 4, or 6 is performed.

If $n$ or $m$ is 0, no computation is performed. See references [“42” on page 1255] and [“46” on page 1255].

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. $\text{cblas_layout} \neq \text{CblasRowMajor or CblasColMajor}$
2. $m < 0$
3. $m > \text{ldb}$
4. $n < 0$
5. $\text{lda, ldb} \leq 0$
6. $\text{side} \neq 'L'$ or 'R'
7. $\text{side} = 'L'$ and $m > \text{lda}$
8. $\text{side} = 'R'$ and $n > \text{lda}$
9. $\text{cblas_side} \neq \text{CblasLeft or CblasRight}$
10. $\text{cblas_side} = \text{CblasLeft and m > lda}$
11. $\text{cblas_side} = \text{CblasRight and n > lda}$
12. $\text{uplo} \neq 'L'$ or 'U'
13. $\text{cblas_uplo} \neq \text{CblasLower or CblasUpper}$
14. $\text{transa} \neq 'T', 'N'$, or 'C'
15. $\text{cblas_transa} \neq \text{CblasNoTrans, CblasTrans, or CblasConjTrans}$
16. $\text{diag} \neq 'N'$ or 'U'
17. $\text{cblas_diag} \neq \text{CblasNonUnit or CblasUnit}$

Examples

Example 1

This example shows the solution $B \leftarrow \alpha(A^{-1})B$, 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:

<table>
<thead>
<tr>
<th>SIDE</th>
<th>UPLO</th>
<th>TRANS</th>
<th>DIAG</th>
<th>M</th>
<th>N</th>
<th>ALPHA</th>
<th>A</th>
<th>LDA</th>
<th>B</th>
<th>LDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>'L'</td>
<td>'U'</td>
<td>'T'</td>
<td>'N'</td>
<td>5</td>
<td>4</td>
<td>1.0</td>
<td>A</td>
<td>7</td>
<td>B</td>
<td>6</td>
</tr>
</tbody>
</table>

CALL STRSM( 'L', 'U', 'T', 'N', 5, 4, 1.0, A, 7, B, 6 )

\[
A = \begin{bmatrix}
3.0 & -1.0 & 2.0 & 2.0 & 1.0 \\
2.0 & 4.0 & -1.0 & 3.0 & \\
-1.0 & 4.0 & -2.0 & & \\
& & & 1.0 & \\
& & & & \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
6.0 & 10.0 & -1.0 \\
-16.0 & -1.0 & 6.0 \\
-2.0 & 1.0 & -4.0 \\
14.0 & 0.0 & -14.0 \\
-1.0 & 2.0 & 1.0 \\
\end{bmatrix}
\]

Output:

\[
B = \begin{bmatrix}
2.0 & 3.0 & 1.0 \\
5.0 & 5.0 & 4.0 \\
0.0 & 1.0 & 2.0 \\
3.0 & 1.0 & -3.0 \\
-1.0 & 2.0 & 1.0 \\
\end{bmatrix}
\]

Example 2

This example shows the solution \( B \leftarrow \alpha (A^T)B \), 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:

<table>
<thead>
<tr>
<th>SIDE</th>
<th>UPLO</th>
<th>TRANS</th>
<th>DIAG</th>
<th>M</th>
<th>N</th>
<th>ALPHA</th>
<th>A</th>
<th>LDA</th>
<th>B</th>
<th>LDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>'L'</td>
<td>'U'</td>
<td>'T'</td>
<td>'N'</td>
<td>5</td>
<td>4</td>
<td>1.0</td>
<td>A</td>
<td>7</td>
<td>B</td>
<td>6</td>
</tr>
</tbody>
</table>

CALL STRSM( 'L', 'U', 'T', 'N', 5, 4, 1.0, A, 7, B, 6 )

\[
A = \begin{bmatrix}
-1.0 & -4.0 & -2.0 & 2.0 & 3.0 \\
-2.0 & 2.0 & 2.0 & 2.0 & \\
& & -3.0 & -1.0 & 4.0 \\
& & & 1.0 & 0.0 \\
& & & & -2.0 \\
& & & & & \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
-1.0 & -2.0 & -3.0 & -4.0 \ \\
2.0 & -2.0 & -4.0 & -12.0 \\
10.0 & 5.0 & -8.0 & -7.0 \\
14.0 & 15.0 & 1.0 & 8.0 \\
-3.0 & 4.0 & 3.0 & 16.0 \\
\end{bmatrix}
\]

Output:

\[
B = \begin{bmatrix}
1.0 & 2.0 & 3.0 & 4.0 \\
3.0 & 3.0 & -1.0 & 2.0 \\
-2.0 & -1.0 & 0.0 & 1.0 \\
4.0 & 4.0 & -3.0 & -3.0 \\
\end{bmatrix}
\]
Example 3

This example shows the solution $B \leftarrow \alpha B(A^{-1})$, where $A$ is a real 5 by 5 lower triangular matrix that is not unit triangular, and $B$ is a real 3 by 5 rectangular matrix.

Call Statement and Input:

```
SIDE  UPLO TRANSA  DIAG  M   N    ALPHA  A  LDA  B  LDB
|     |     |      |    |   |     |     |   |   |   |
CALL STRSM( 'R' , 'L' , 'N'  , 'N' , 3 , 5 ,  1.0  , A , 7 , B , 4 )
```

\[
A = \begin{bmatrix}
2.0 & . & . & . & . \\
2.0 & 3.0 & . & . & . \\
2.0 & 1.0 & 1.0 & . & . \\
0.0 & 3.0 & 0.0 & -2.0 & . \\
2.0 & 4.0 & -1.0 & 2.0 & -1.0 \\
. & . & . & . & . \\
. & . & . & . & . \\
. & . & . & . & . \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
10.0 & 4.0 & 0.0 & 0.0 & 1.0 \\
10.0 & 14.0 & -4.0 & 6.0 & -3.0 \\
-8.0 & 2.0 & -5.0 & 4.0 & -2.0 \\
. & . & . & . & . \\
. & . & . & . & . \\
\end{bmatrix}
\]

Output:

\[
B = \begin{bmatrix}
3.0 & 4.0 & -1.0 & -1.0 & -1.0 \\
2.0 & 1.0 & -1.0 & 0.0 & 3.0 \\
-2.0 & -1.0 & -3.0 & 0.0 & 2.0 \\
. & . & . & . & . \\
. & . & . & . & . \\
\end{bmatrix}
\]

Example 4

This example shows the solution $B \leftarrow \alpha B(A^{-1})$, where $A$ is a real 6 by 6 upper triangular matrix that is unit triangular, and $B$ is a real 1 by 6 rectangular matrix.

Note: Because matrix $A$ is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of 1.0 for the diagonal element.

Call Statement and Input:

```
SIDE  UPLO TRANSA  DIAG  M   N    ALPHA  A  LDA  B  LDB
|     |     |      |    |   |     |     |   |   |   |
CALL STRSM( 'R' , 'U' , 'N'  , 'U' , 1 , 6 ,  1.0  , A , 7 , B , 2 )
```

\[
A = \begin{bmatrix}
. & 2.0 & -3.0 & 1.0 & 2.0 & 4.0 \\
. & . & 0.0 & 1.0 & 1.0 & -2.0 \\
. & . & . & 4.0 & -1.0 & 1.0 \\
. & . & . & 0.0 & -1.0 & . \\
. & . & . & . & 2.0 & . \\
. & . & . & . & . & . \\
. & . & . & . & . & . \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
1.0 & 4.0 & -2.0 & 10.0 & 2.0 & -6.0 \\
. & . & . & . & . & . \\
. & . & . & . & . & . \\
\end{bmatrix}
\]
Example 5

This example shows the solution $B \leftarrow aB(A^{-1})$, where $A$ is a complex 5 by 5 lower triangular matrix that is not unit triangular, and $B$ is a complex 3 by 5 rectangular matrix.

Call Statement and Input:

```
SIDE UPLO TRANSA DIAG M N ALPHA A LDA B LDB
CALL CTRSM( 'R' , 'L' , 'N' , 'N' , 3 , 5 ,  ALPHA , A , 7 , B , 4 )
```

```
ALPHA    =  (1.0, 0.0)
```

```
A = 
\begin{bmatrix}
(2.0, -3.0) & (3.0, -1.0) & . & . & . \\
(2.0, 2.0) & (1.0, 1.0) & . & . & . \\
(0.0, 0.0) & (0.0, -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}
```

Output:

```
B = 
\begin{bmatrix}
(3.0, 0.0) & (4.0, 0.0) & (-1.0, -2.0) & (-1.0, -1.0) & (-1.0, -4.0) \\
(2.0, -1.0) & (1.0, 2.0) & (-1.0, -3.0) & (0.0, 2.0) & (3.0, -4.0) \\
(-2.0, 1.0) & (-1.0, -3.0) & (-3.0, 1.0) & (0.0, 0.0) & (2.0, -2.0) \\
. & . & . & . & .
\end{bmatrix}
```

Example 6

This example shows the solution $B \leftarrow a(A^{*}B)$, where $A$ is a complex 5 by 5 upper triangular matrix that is not unit triangular, and $B$ is a complex 5 by 1 rectangular matrix.

Call Statement and Input:

```
SIDE UPLO TRANSA DIAG M N ALPHA A LDA B LDB
CALL CTRSM( 'L' , 'U' , 'C' , 'N' , 5 , 1 ,  ALPHA , A , 6 , B , 6 )
```

```
ALPHA    =  (1.0, 0.0)
```

```
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) \\
. & . & . & . & (2.0, -1.0) \\
. & . & . & . & .
\end{bmatrix}
```

```
B = 
\begin{bmatrix}
(-8.0, -19.0) \\
(8.0, 21.0) \\
(44.0, -8.0)
\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>
<thead>
<tr>
<th>Table 111: Data Types.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Types</td>
</tr>
<tr>
<td>$A, C$</td>
</tr>
<tr>
<td>$\alpha, \beta$</td>
</tr>
<tr>
<td>Subprogram</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>SSYRK</td>
</tr>
<tr>
<td>Short-precision complex</td>
</tr>
<tr>
<td>Short-precision complex</td>
</tr>
<tr>
<td>CSYRK</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
<tr>
<td>DSYRK</td>
</tr>
<tr>
<td>Long-precision complex</td>
</tr>
<tr>
<td>Long-precision complex</td>
</tr>
<tr>
<td>ZSYRK</td>
</tr>
<tr>
<td>Short-precision complex</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>CHERK</td>
</tr>
<tr>
<td>Long-precision complex</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
<tr>
<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 34.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SSYRK</td>
</tr>
<tr>
<td>C and C++</td>
<td>ssyrk</td>
</tr>
<tr>
<td>Language</td>
<td>Syntax</td>
</tr>
<tr>
<td>----------</td>
<td>--------</td>
</tr>
</tbody>
</table>
| 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 `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` = CblasUpper, A is stored in upper storage mode.
  - If `cblas_uplo` = 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 matrix A to use in the computation, where:
  - If `trans` = 'N', A is used, resulting in equation 1 or 3.
  - If `trans` = 'T', $A^T$ is used, resulting in equation 2.
  - If `trans` = 'C', $A^H$ is used, resulting in equation 4.

  Specified as: a single character, where:
  - For SSYRK and DSYRK, it must be 'N', 'T', or 'C'.
  - For CSYRK and ZSYRK, it must be 'N' or 'T'.
  - For CHERK and ZHERK, it must be 'N' or 'C'.

- **cblas_trans** indicates the form of matrix A to use in the computation, where:
  - If `cblas_trans` = CblasNoTrans, A is used, resulting in equation 1 or 3.
  - If `cblas_trans` = CblasTrans, $A^T$ is used, resulting in equation 2.
  - If `cblas_trans` = 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 ≤ n ≤ ldc and:
If trans = 'N', then n ≤ lda.

k
has the following meaning, where:
If trans = 'N', it is the number of columns in matrix A.
If trans = 'T' or 'C', it is the number of rows in matrix A.
Specified as: an integer; k ≥ 0 and:
If trans = 'T' or 'C', then k ≤ lda.

alpha
is the scalar α.
Specified as: a number of the data type indicated in Table 111 on page 445.

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 Aᵀ or Aᴴ; 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 111 on page 445, 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 ≥ n.
If trans = 'T' or 'C', lda ≥ k.

beta
is the scalar β.
Specified as: a number of the data type indicated in Table 111 on page 445.

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 111 on page 445.

 ldc
is the leading dimension of the array specified for c.
Specified as: an integer; ldc > 0 and ldc ≥ n.

On Return
c
is matrix C of order n, which is real symmetric, complex symmetric, or complex Hermitian, containing the results of the computation, where:
If uplo = 'U', C is stored in upper storage mode.
If uplo = 'L', C is stored in lower storage mode.

Returned as: an ldc by (at least) n array, containing numbers of the data type indicated in Table 111 on page 445.

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 β is one and α or k is zero, in which case no computation is performed.
5. For a description of how symmetric matrices are stored in upper and lower storage mode, see "Symmetric Matrix" on page 80. For a description of how complex Hermitian matrices are stored in upper and lower storage mode, see "Complex Hermitian Matrix" on page 84.

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:
α and β are scalars.
A is a rectangular matrix, which is n by k for equations 1 and 3, and is k by n for equations 2 and 4.
C is a matrix of order n of the type indicated above, stored in upper or lower storage mode.

See references [“42” on page 1255] and [“48” on page 1255]. In the following two cases, no computation is performed:
• n is 0.
• β is one, and α is zero or k is zero.

Assuming the above conditions do not exist, if β is not one, and α is zero or k is zero, then βC is returned.

Error conditions
Resource Errors
Unable to allocate internal work area.

Computational Errors
None

Input-Argument Errors
1. cblas_layout ≠ CblasRowMajor or CblasColMajor
2. lda, ldc ≤ 0
3. ldc < n
4. k, n < 0
5. uplo ≠ 'U' or 'L'
6. `cblas_uplo` ≠ `CblasLower` or `CblasUpper`
7. `trans` ≠ 'N', 'T', or 'C' for `SSYRK` and `DSYRK`
8. `trans` ≠ 'N' or 'T' for `CSYRK` and `ZSYRK`
9. `trans` ≠ 'N' or 'C' for `CHERK` and `ZHERK`
10. `trans` = 'N' and `lda` < `n`
11. `trans` = 'T' or 'C' and `lda` < `k`
12. `cblas_trans` ≠ `CblasNoTrans`, `CblasTrans`, or `CblasConjTrans` for `SSYRK` and `DSYRK`
13. `cblas_trans` ≠ `CblasNoTrans` or `CblasTrans` for `CSYRK` and `ZSYRK`
14. `cblas_trans` ≠ `CblasNoTrans` or `CblasConjTrans` for `CHERK` and `ZHERK`
15. `cblas_trans` = `CblasNoTrans` and `lda` < `n`
16. `cblas_trans` = `CblasTrans`, or `CblasConjTrans` and `lda` < `k`

**Examples**

**Example 1**

This example shows the computation $C \leftarrow \alpha A A^T + \beta C$, 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:**

<table>
<thead>
<tr>
<th>UPL0</th>
<th>TRANS</th>
<th>N</th>
<th>K</th>
<th>ALPHA</th>
<th>A</th>
<th>LDA</th>
<th>BETA</th>
<th>C</th>
<th>LDC</th>
</tr>
</thead>
<tbody>
<tr>
<td>'U'</td>
<td>'N'</td>
<td>8</td>
<td>2</td>
<td>1.0</td>
<td>A</td>
<td>9</td>
<td>1.0</td>
<td>C</td>
<td>10</td>
</tr>
</tbody>
</table>

$$
A = \begin{bmatrix}
0.0 & 8.0 \\
1.0 & 9.0 \\
2.0 & 10.0 \\
3.0 & 11.0 \\
4.0 & 12.0 \\
5.0 & 13.0 \\
6.0 & 14.0 \\
7.0 & 15.0 \\
\end{bmatrix}
$$

$$
C = \begin{bmatrix}
0.0 & 1.0 & 3.0 & 6.0 & 10.0 & 15.0 & 21.0 & 28.0 \\
. & 2.0 & 4.0 & 7.0 & 11.0 & 16.0 & 22.0 & 29.0 \\
. & . & 5.0 & 8.0 & 12.0 & 17.0 & 23.0 & 30.0 \\
. & . & . & 9.0 & 13.0 & 18.0 & 24.0 & 31.0 \\
. & . & . & . & 14.0 & 19.0 & 25.0 & 32.0 \\
. & . & . & . & . & 20.0 & 26.0 & 33.0 \\
. & . & . & . & . & . & 27.0 & 34.0 \\
. & . & . & . & . & . & . & 35.0 \\
. & . & . & . & . & . & . & . \\
\end{bmatrix}
$$

**Output:**

$$
C = \begin{bmatrix}
64.0 & 73.0 & 83.0 & 94.0 & 106.0 & 119.0 & 133.0 & 148.0 \\
. & 84.0 & 96.0 & 109.0 & 123.0 & 138.0 & 154.0 & 171.0 \\
. & . & 189.0 & 124.0 & 148.0 & 157.0 & 175.0 & 194.0 \\
. & . & . & 190.0 & 157.0 & 176.0 & 196.0 & 217.0 \\
. & . & . & . & 174.0 & 195.0 & 217.0 & 240.0 \\
. & . & . & . & . & 214.0 & 238.0 & 263.0 \\
. & . & . & . & . & . & 259.0 & 286.0 \\
. & . & . & . & . & . & . & 399.0 \\
. & . & . & . & . & . & . & . \\
\end{bmatrix}
$$
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:

```
UPLO TRANS N K ALPHA A LDA BETA C LDC
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:

```
UPLO TRANS N K ALPHA A LDA BETA C LDC
CALL CSYRK( 'U', 'N', 3, 5, ALPHA, A, 3, BETA, C, 4 )
```

$\text{ALPHA} = (1.0, 1.0)$

$\text{BETA} = (1.0, 1.0)$

$A = \begin{bmatrix}
(2.0, 0.0) & (3.0, 2.0) & (4.0, 1.0) & (1.0, 7.0) & (0.0, 0.0) \\
(3.0, 3.0) & (8.0, 0.0) & (2.0, 5.0) & (2.0, 4.0) & (1.0, 2.0) \\
(1.0, 3.0) & (2.0, 1.0) & (6.0, 0.0) & (3.0, 2.0) & (2.0, 2.0) \\
\end{bmatrix}$

$C = \begin{bmatrix}
(2.0, 1.0) & (1.0, 9.0) & (4.0, 5.0) \\
. & (3.0, 1.0) & (6.0, 7.0) \\
. & . & (8.0, 1.0) \\
\end{bmatrix}$
Example 4

This example shows the computation $C \leftarrow \alpha A^H A + \beta C$, where $A$ is a 5 by 3 complex rectangular matrix, and $C$ is a complex Hermitian matrix of order 3, stored in lower storage mode.

**Note:** The imaginary parts of the diagonal elements of a complex Hermitian matrix are assumed to be zero, so you do not have to set these values. On output, they are set to zero.

**Call Statement and Input:**

```
CALL CHERK( 'L' , 'C' , 3 , 5 , 1.0  , A , 5 , 1.0  , C , 4 )
```

**Output:**

```
C = [ (-57.0, 13.0) (-63.0, 79.0) (-24.0, 78.0)]
    [ (-28.0, 90.0) (-55.0, 103.0)]
    [ (13.0, 75.0) . ]
```

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. $C \leftarrow \alpha A B^T + \alpha B A^T + \beta C$
2. $C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$

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:
Table 112: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A, B, C, \alpha, \beta$</td>
<td></td>
</tr>
<tr>
<td>Short-precision real</td>
<td>SSYR2K</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DSYR2K</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CSYR2K</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZSYR2K</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CHER2K</td>
</tr>
<tr>
<td>Long-precision complex</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 34.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SSYR2K</td>
</tr>
<tr>
<td>C and C++</td>
<td>ssyr2k</td>
</tr>
<tr>
<td>CBLAS</td>
<td>cblas_syr2k</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 $C$, 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:

3. $C \leftarrow \alpha AB^H + \overline{\alpha} BA^H + \beta C$

4. $C \leftarrow \alpha A^H B + \overline{\alpha} B^H A + \beta C$
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 \text{CblasUpper} or \text{CblasLower}.

\text{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'.

\text{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 \text{CblasNoTrans}, \text{CblasTrans}, or \text{CblasConjTrans}.
For CSYR2K and ZSYR2K, it must be \text{CblasNoTrans} or \text{CblasTrans}.
For CHER2K and ZHER2K, it must be \text{CblasNoTrans} or \text{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 \( trans = 'T' \) 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 112 on page 452.

\( a \)

is the rectangular matrix \( A \), where:

If \( trans = 'N' \), \( A \) is \( n \) by \( k \).
If \( trans = 'T' \) or \( 'C' \), \( A \) is \( k \) by \( n \).

\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: a two-dimensional array, containing numbers of the data type indicated in Table 112 on page 452, 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 112 on page 452, 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 \( 'C' \), \( \text{ldb} \geq k \).

\( \text{beta} \)

is the scalar \( \beta \).
Specified as: a number of the data type indicated in Table 112 on page 452.

\( c \)

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 \( \text{ldc} \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 112 on page 452.

\( \text{ldc} \)

is the leading dimension of the array specified for \( c \).
Specified as: an integer; \( \text{ldc} > 0 \) and \( \text{ldc} \geq n \).

**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 \( \text{uplo} = 'U' \), \( C \) is stored in upper storage mode.
If \( \text{uplo} = 'L' \), \( C \) is stored in lower storage mode.
Returned as: an $ldc$ by (at least) $n$ array, containing numbers of the data type indicated in Table 112 on page 452.

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 80. For a description of how complex Hermitian matrices are stored in upper and lower storage mode, see “Complex Hermitian Matrix” on page 84.

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. $C \leftarrow \alpha AB^H + \alpha BA^H + \beta C$
4. $C \leftarrow \alpha A^H B + \alpha B^H A + \beta C$

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 [“42” on page 1255], [“48” on page 1255], and [“86” on page 1257]. 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. $cblas_layout \neq CblasRowMajor$ or $CblasColMajor$
2. $lda, ldb, ldc \leq 0$
3. \( ldc < n \)
4. \( k, n < 0 \)
5. \( uplo \neq 'U' \) or \( 'L' \)
6. \( cblas_uplo \neq \text{CblasLower or CblasUpper} \)
7. \( trans \neq 'N', 'T', \) or \( 'C' \) for SSYR2K and DSYR2K
8. \( trans \neq 'N' \) or \( 'T' \) for CSYR2K and ZSYR2K
9. \( trans \neq 'N' \) or \( 'C' \) for CHER2K and ZHER2K
10. \( trans = 'N' \) and \( lda < n \)
11. \( trans = 'T' \) or \( 'C' \) and \( lda < k \)
12. \( trans = 'N' \) and \( ldb < n \)
13. \( trans = 'T' \) or \( 'C' \) and \( ldb < k \)
14. \( cblas_transa \neq \text{CblasNoTrans, CblasTrans, or CblasConjTrans} \) for SSYR2K and DSYR2K
15. \( cblas_transa \neq \text{CblasNoTrans or CblasTrans} \) for CSYR2K and ZSYR2K
16. \( cblas_transa \neq \text{CblasNoTrans or CblasConjTrans} \) for CHER2K and ZHER2K
17. \( cblas_transa = \text{CblasNoTrans and } lda < n \)
18. \( cblas_transa = \text{CblasTrans, or CblasConjTrans and } lda < k \)
19. \( cblas_trans = \text{CblasNoTrans and } ldb < n \)
20. \( cblas_trans = \text{CblasNoTrans or CblasConjTrans and } ldb < k \)

**Examples**

**Example 1**

This example shows the computation \( C \leftarrow \alpha A B^T + \alpha B A^T + \beta C \), 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)
```

\[
A = \begin{bmatrix}
0.0 & 8.0 \\
1.0 & 9.0 \\
2.0 & 10.0 \\
3.0 & 11.0 \\
4.0 & 12.0 \\
5.0 & 13.0 \\
6.0 & 14.0 \\
7.0 & 15.0 \\
. & . \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
15.0 & 7.0 \\
14.0 & 6.0 \\
13.0 & 5.0 \\
12.0 & 4.0 \\
11.0 & 3.0 \\
10.0 & 2.0 \\
9.0 & 1.0 \\
8.0 & 0.0 \\
\end{bmatrix}
\]

\[
C = \begin{bmatrix}
0.0 & 1.0 & 3.0 & 6.0 & 10.0 & 15.0 & 21.0 & 28.0 \\
. & 2.0 & 4.0 & 7.0 & 11.0 & 16.0 & 22.0 & 29.0 \\
. & . & 5.0 & 8.0 & 12.0 & 17.0 & 23.0 & 30.0 \\
. & . & . & 9.0 & 13.0 & 18.0 & 24.0 & 31.0 \\
. & . & . & . & 14.0 & 19.0 & 25.0 & 32.0 \\
\end{bmatrix}
\]
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 )
```

```
A    = [ 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 ]
```

```
B    = [ 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0 ]
      [ 2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 ]
      [ 3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0 ]
```

```
C    = [ 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 ]
```

Output:

```
C    = [ 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 ]
      [ . . 169.0 175.0 182.0 190.0 199.0 212.0 ]
      [ . . . 174.0 177.0 181.0 186.0 190.0 ]
      [ . . . . 172.0 172.0 173.0 174.0 ]
      [ . . . . . 163.0 168.0 174.0 ]
      [ . . . . . . 147.0 150.0 ]
      [ . . . . . . . 153.0 ]
```

```
C    = [ 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 ]
      [ 3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0 ]
      [ 4.0  5.0  6.0  7.0  8.0  9.0 10.0 11.0 ]
      [ 5.0  6.0  7.0  8.0  9.0 10.0 11.0 12.0 ]
      [ 6.0  7.0  8.0  9.0 10.0 11.0 12.0 13.0 ]
      [ 7.0  8.0  9.0 10.0 11.0 12.0 13.0 14.0 ]
```

```
C    = [ 16.0   .     .     .     .     .     .     .  ]
      [ 38.0  84.0   .     .     .     .     .     .  ]
      [ 60.0 124.0 187.0   .     .     .     .     .  ]
      [ 82.0 164.0 245.0 325.0   .     .     .     .  ]
      [ 104.0 204.0 303.0 401.0 498.0   .     .     .  ]
      [ 126.0 244.0 361.0 477.0 592.0 706.0   .     .  ]
      [ 148.0 284.0 419.0 553.0 686.0 818.0 949.0   .  ]
      [ 170.0 324.0 477.0 629.0 780.0 930.0 1079.0 1227.0 ]
```
**Example 3**

This example shows the computation $C \leftarrow \alpha AB^T + \beta BA^T + \beta C$, 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:**

```plaintext
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** =

\[
\begin{bmatrix}
(2.0, 5.0) & (3.0, 2.0) & (4.0, 1.0) & (1.0, 7.0) & (0.0, 0.0) \\
(1.0, 3.0) & (2.0, 1.0) & (0.0, 5.0) & (3.0, 2.0) & (2.0, 2.0)
\end{bmatrix}
\]

**B** =

\[
\begin{bmatrix}
(1.0, 5.0) & (6.0, 2.0) & (3.0, 1.0) & (2.0, 0.0) & (1.0, 0.0) \\
(2.0, 4.0) & (7.0, 5.0) & (2.0, 5.0) & (2.0, 4.0) & (0.0, 0.0) \\
(3.0, 5.0) & (8.0, 1.0) & (1.0, 5.0) & (1.0, 0.0) & (1.0, 1.0)
\end{bmatrix}
\]

**C** =

\[
\begin{bmatrix}
(2.0, 3.0) & . & . & . & . \\
(1.0, 9.0) & (3.0, 3.0) & . & . & . \\
(4.0, 5.0) & (6.0, 7.0) & (8.0, 3.0) & . & .
\end{bmatrix}
\]

**Output:**

\[
\begin{bmatrix}
(-101.0, 121.0) & . & . & . & . \\
(-182.0, 192.0) & (-274.0, 248.0) & . & . & . \\
(-98.0, 146.0) & (-163.0, 205.0) & (-151.0, 115.0) & . & .
\end{bmatrix}
\]

**Example 4**

This example shows the computation:

\[
C \leftarrow \alpha A^H B + \beta A^H A + \beta C
\]

where $A$ and $B$ are 5 by 3 complex rectangular matrices, and $C$ is a complex Hermitian matrix of order 3, stored in upper storage mode.

**Note:** The imaginary parts of the diagonal elements of a complex Hermitian matrix are assumed to be zero, so you do not have to set these values. On output, they are set to zero.

**Call Statement and Input:**

```plaintext
CALL CHER2K( 'U', 'C', 3, 5, ALPHA, A, 5, B, 5, 1.0, C, 4 )
```

**ALPHA** = (1.0, 1.0)

**A** =

\[
\begin{bmatrix}
(2.0, 0.0) & (3.0, 2.0) & (4.0, 1.0) \\
(3.0, 3.0) & (8.0, 0.0) & (2.0, 5.0) \\
(1.0, 3.0) & (2.0, 1.0) & (6.0, 0.0)
\end{bmatrix}
\]

**Output:**

\[
\begin{bmatrix}
(2.0, 3.0) & (4.0, 1.0) \\
(3.0, 3.0) & (8.0, 0.0) \\
(1.0, 3.0) & (2.0, 1.0)
\end{bmatrix}
\]
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>
<thead>
<tr>
<th>Table 113: Data Types.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Types</strong></td>
</tr>
<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></td>
</tr>
<tr>
<td>Long-precision complex</td>
</tr>
<tr>
<td></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 34.
Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGETMI</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgetmi</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 113 on page 459.

\[ lda \]

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 \( n \) by \( n \) 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 113 on page 459.

**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:

\[ a_{ji} = a_{ij} \text{ for } i, j = 1, n \]

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:

\[ a_{ji} = \bar{a}_{ij} \text{ for } i, j = 1, n \]

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:

```plaintext
A LDA N
CALL SGETMI( A(2,3) , 10 , 5 )
```

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

Output:

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

Example 2

This example shows an in-place matrix conjugate transpose of matrix A having 5 rows and 5 columns.

Call Statement and Input:

```plaintext
A LDA N
CALL ZGECMI( A(2,3) , 10 , 5 )
```

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

Output:

\[
\begin{bmatrix}
. & . & (1.0, -1.0) & (2.0, -2.0) & (3.0, -3.0) & (4.0, -4.0) & (5.0, -5.0) \\
. & . & (6.0, -6.0) & (7.0, -7.0) & (8.0, -8.0) & (9.0, -9.0) & (10.0, -10.0) \\
. & . & (11.0, -11.0) & (12.0, -12.0) & (13.0, -13.0) & (14.0, -14.0) & (15.0, -15.0) \\
. & . & (16.0, -16.0) & (17.0, -17.0) & (18.0, -18.0) & (19.0, -19.0) & (20.0, -20.0) \\
. & . & (21.0, -21.0) & (22.0, -22.0) & (23.0, -23.0) & (24.0, -24.0) & (25.0, -25.0) \\
. & . & . & . & . & . & . \\
. & . & . & . & . & . & . \\
. & . & . & . & . & . & . \\
. & . & . & . & . & . & . \\
\end{bmatrix}
\]
SGETMO, 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>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A, B$</td>
<td></td>
</tr>
<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>CGECMO</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGETMO</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 34.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGETMO</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgetmo</td>
</tr>
</tbody>
</table>

**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 114 on page 462.

**$lda$**

is the leading dimension of the array specified for $a$.

Specified as: an integer; $lda > 0$ and $lda \geq m$.

**$m$**

is the number of rows in matrix $A$ and the number of columns in matrix $B$.

Specified as: 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 114 on page 462.

Notes

1. The matrix \( B \) must have no common elements with matrix \( A \); otherwise, results are unpredictable. See “Vector concepts” on page 71.

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:

\[ b_{ji} = \overline{a_{ij}} \text{ for } i = 1, m \text{ and } j = 1, n \]

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 \( B \).
Example 2

This example uses the same input matrix A as in Example 1 to show that transposes can be achieved in the same array as long as the input and output data do not overlap. On output, the input data is not overwritten in the array.

Call Statement and Input:

```
A     LDA   M   N     B     LDB
      |    |   |     |      |
CALL SGETMO( A(2,3) , 10 , 5 , 4 , B(2,2) , 6 )
```

Output:

```
|   .     .     .     .     .     .   . |
|   .     .    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  . |
|   .     .    5.0  10.0  15.0  20.0  . |
|  1.0   2.0   3.0   4.0   5.0    .   . |
|  6.0   7.0   8.0   9.0  10.0    .   . |
| 11.0  12.0  13.0  14.0  15.0    .   . |
| 16.0  17.0  18.0  19.0  20.0    .   . |
```

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     LDA   M   N     B     LDB
      |    |   |     |      |
CALL ZGECMO( A(2,3) , 10 , 5 , 4 , B(2,2) , 6 )
```

Output:

```
|   .     .     .     .     .     .     . |
|   .     .   (1.0,1.0) (6.0, 6.0) (11.0,11.0) (16.0,16.0) |
|   .     .   (2.0,2.0) (7.0, 7.0) (12.0,12.0) (17.0,17.0) |
|   .     .   (3.0,3.0) (8.0, 8.0) (13.0,13.0) (18.0,18.0) |
|   .     .   (4.0,4.0) (9.0, 9.0) (14.0,14.0) (19.0,19.0) |
|   .     .   (5.0,5.0) (10.0,10.0) (15.0,15.0) (20.0,20.0) |
```
Matrix operations subroutines 465
Chapter 10. Linear algebraic equations subroutines

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” on page 467
- “Banded Linear Algebraic Equation Subroutines” on page 472
- “Sparse Linear Algebraic Equation Subroutines” on page 474
- “Linear Least Squares Subroutines” on page 475

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 [“9” on page 1253].

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 [“9” on page 1253].

Table 115: 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>SGESV^AGPU</td>
<td>DGESV^AGPU</td>
<td>“SGESV, DGESV, CGESV, ZGESV (General Matrix Factorization and Multiple Right-Hand Side Solve)” on page 480</td>
</tr>
<tr>
<td>CGESV^AGPU</td>
<td>ZGESV^AGPU</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgesv^A</td>
<td>LAPACKE_dgesv^A</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgesv^A</td>
<td>LAPACKE_zgesv^A</td>
<td></td>
</tr>
<tr>
<td>SGETRF^AGPU</td>
<td>DGETRF^AGPU</td>
<td>“SGETRF, DGETRF, CGETRF and ZGETRF (General Matrix Factorization)” on page 484</td>
</tr>
<tr>
<td>CGETRF^AGPU</td>
<td>ZGETRF^AGPU</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgetrf^A</td>
<td>LAPACKE_dgetrf^A</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgetrf^A</td>
<td>LAPACKE_zgetrf^A</td>
<td></td>
</tr>
<tr>
<td>SGETRS^AGPU</td>
<td>DGETRS^AGPU</td>
<td>“SGETRS, DGETRS, CGETRS, and ZGETRS (General Matrix Multiple Right-Hand Side Solve)” on page 488</td>
</tr>
<tr>
<td>CGETRS^AGPU</td>
<td>ZGETRS^AGPU</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgetrs^A</td>
<td>LAPACKE_dgetrs^A</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgetrs^A</td>
<td>LAPACKE_zgetrs^A</td>
<td></td>
</tr>
<tr>
<td>SGECON^A</td>
<td>DGECON^A</td>
<td>“SGECON, DGECON, CGECON, and ZGECON (Estimate the Reciprocal of the Condition Number of a General Matrix)” on page 504</td>
</tr>
<tr>
<td>CGECON^A</td>
<td>ZGECON^A</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgecon^A</td>
<td>LAPACKE_dgecon^A</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgecon^A</td>
<td>LAPACKE_zgecon^A</td>
<td></td>
</tr>
<tr>
<td>Short-Precision Subroutine</td>
<td>Long-Precision Subroutine</td>
<td>Descriptive Name and Location</td>
</tr>
<tr>
<td>----------------------------</td>
<td>---------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>SGETRIΔ</td>
<td>DGETRIΔ</td>
<td>“SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD (General Matrix Inverse, Condition Number Reciprocal, and Determinant)” on page 511</td>
</tr>
<tr>
<td>CGETRIΔ</td>
<td>ZGETRIΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgetriΔ</td>
<td>LAPACKE_dgetriΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgetriΔ</td>
<td>LAPACKE_zgetriΔ</td>
<td></td>
</tr>
<tr>
<td>SLANGEΔ</td>
<td>DLANGEΔ</td>
<td>“SLANGE, DLANGE, CLANGE, and ZLANGE (General Matrix Norm)” on page 518</td>
</tr>
<tr>
<td>CLANGEΔ</td>
<td>ZLANGEΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_slangeΔ</td>
<td>LAPACKE_dlange Δ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_clangeΔ</td>
<td>LAPACKE_zlange Δ</td>
<td></td>
</tr>
<tr>
<td>SPPSVΔ_GPU</td>
<td>DPPSVΔ_GPU</td>
<td>“SPPSV, DPPSV, CPPSV, and ZPPSV (Positive Definite Real Symmetric and Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)” on page 521</td>
</tr>
<tr>
<td>CPPSVΔ_GPU</td>
<td>ZPPSVΔ_GPU</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sppsvΔ</td>
<td>LAPACKE_dpssvΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cppsvΔ</td>
<td>LAPACKE_zppsvΔ</td>
<td></td>
</tr>
<tr>
<td>SPOSVΔ_GPU</td>
<td>DPOSVΔ_GPU</td>
<td>“SPOSV, DPOSV, CPOSV, and ZPOSV (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)” on page 527</td>
</tr>
<tr>
<td>CPOSVΔ_GPU</td>
<td>ZPOSVΔ_GPU</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sposvΔ</td>
<td>LAPACKE_dposvΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cposvΔ</td>
<td>LAPACKE_zposvΔ</td>
<td></td>
</tr>
<tr>
<td>SPOTRFΔ_GPU</td>
<td>DPOTRFΔ_GPU</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 533</td>
</tr>
<tr>
<td>CPOTRFΔ_GPU</td>
<td>ZPOTRFΔ_GPU</td>
<td></td>
</tr>
<tr>
<td>SPPTRFΔ_GPU</td>
<td>DPPTRFΔ_GPU</td>
<td></td>
</tr>
<tr>
<td>CPPTRFΔ_GPU</td>
<td>ZPPTRFΔ_GPU</td>
<td></td>
</tr>
<tr>
<td>LAPACKEスポットRFΔ</td>
<td>LAPACKE_dpotrfΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpotrfΔ</td>
<td>LAPACKE_zpotrfΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKEスポットTRFΔ</td>
<td>LAPACKE_dpptrfΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpptrfΔ</td>
<td>LAPACKE_zpptrfΔ</td>
<td></td>
</tr>
<tr>
<td>SPOTRSΔ_GPU</td>
<td>DPOTRSΔ_GPU</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 545</td>
</tr>
<tr>
<td>CPOTRSΔ_GPU</td>
<td>ZPOTRSΔ_GPU</td>
<td></td>
</tr>
<tr>
<td>SPPTRSΔ_GPU</td>
<td>DPPTRSΔ_GPU</td>
<td></td>
</tr>
<tr>
<td>CPPTRSΔ_GPU</td>
<td>ZPPTRSΔ_GPU</td>
<td></td>
</tr>
<tr>
<td>LAPACKEスポットRSΔ</td>
<td>LAPACKE_dpotrsΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpotrsΔ</td>
<td>LAPACKE_zpotrsΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKEスポットTRSΔ</td>
<td>LAPACKE_dpptrsΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpptrsΔ</td>
<td>LAPACKE_zpptrsΔ</td>
<td></td>
</tr>
<tr>
<td>Short-Precision Subroutine</td>
<td>Long-Precision Subroutine</td>
<td>Descriptive Name and Location</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>SPOCONΔ</td>
<td>DPOCONΔ</td>
<td>“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)” on page 555</td>
</tr>
<tr>
<td>CPOCONΔ</td>
<td>ZPOCONΔ</td>
<td></td>
</tr>
<tr>
<td>SPPCONΔ</td>
<td>DPPCONΔ</td>
<td></td>
</tr>
<tr>
<td>CPPCONΔ</td>
<td>ZPPCONΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spoconΔ</td>
<td>LAPACKE_dpoconΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpoconΔ</td>
<td>LAPACKE_zpoconΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sppconΔ</td>
<td>LAPACKE_dppconΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cppconΔ</td>
<td>LAPACKE_zppconΔ</td>
<td></td>
</tr>
<tr>
<td>SPOTRIΔ</td>
<td>DPOTRIΔ</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 569</td>
</tr>
<tr>
<td>CPOTRIΔ</td>
<td>ZPOTRIΔ</td>
<td></td>
</tr>
<tr>
<td>SPPTRIΔ</td>
<td>DPPTRIΔ</td>
<td></td>
</tr>
<tr>
<td>CPPTRIΔ</td>
<td>ZPPTRIΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spotriΔ</td>
<td>LAPACKE_dpotriΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpotriΔ</td>
<td>LAPACKE_zpotriΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spptriΔ</td>
<td>LAPACKE_dpptriΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpptriΔ</td>
<td>LAPACKE_zpptriΔ</td>
<td></td>
</tr>
<tr>
<td>SLANSYΔ&gt;</td>
<td>[CLANHEΔ]&lt;</td>
<td>DLANSYΔ&gt;</td>
</tr>
<tr>
<td>CLANHEΔ</td>
<td>SLANSPΔ</td>
<td></td>
</tr>
<tr>
<td>CLANHPΔ</td>
<td>DLANSPΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_slansyΔ&gt;</td>
<td>[&lt;CLANHEΔ]&lt;</td>
<td>LAPACKE_clansyΔ&gt;</td>
</tr>
<tr>
<td>LAPACKE_clanhedΔ</td>
<td>LAPACKE_slanspΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_clanhpΔ</td>
<td>LAPACKE_clanhpΔ</td>
<td></td>
</tr>
<tr>
<td>DLANSYΔ</td>
<td>ZLANHEΔ</td>
<td></td>
</tr>
<tr>
<td>ZLANHEΔ</td>
<td>DLANSPΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_dlansyΔ&gt;</td>
<td>[&lt;CLANHEΔ]&lt;</td>
<td>LAPACKE_zlansyΔ&gt;</td>
</tr>
<tr>
<td>LAPACKE_zlanheΔ&gt;</td>
<td>[&lt;CLANHEΔ]</td>
<td>LAPACKE_zlanspΔ&gt;</td>
</tr>
<tr>
<td>LAPACKE_zlanhpΔ</td>
<td>LAPACKE_zlanhpΔ</td>
<td></td>
</tr>
<tr>
<td>Short-Precision Subroutine</td>
<td>Long-Precision Subroutine</td>
<td>Descriptive Name and Location</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>SSYSVΔ</td>
<td>DSYSVΔ</td>
<td>“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 584</td>
</tr>
<tr>
<td>CSYSVΔ</td>
<td>ZSYSVΔ</td>
<td></td>
</tr>
<tr>
<td>CHESVΔ</td>
<td>ZHESVΔ</td>
<td></td>
</tr>
<tr>
<td>SSPSVΔ</td>
<td>DSPSVΔ</td>
<td></td>
</tr>
<tr>
<td>CSPSVΔ</td>
<td>ZSPSVΔ</td>
<td></td>
</tr>
<tr>
<td>CHPSVΔ</td>
<td>ZHPSVΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssysvΔ</td>
<td>LAPACKE_dsysvΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_csysvΔ</td>
<td>LAPACKE_zsysvΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chesvΔ</td>
<td>LAPACKE_zhesvΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sspsvΔ</td>
<td>LAPACKE_dspsvΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cspsvΔ</td>
<td>LAPACKE_zspsvΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chpsvΔ</td>
<td>LAPACKE_zhpsvΔ</td>
<td></td>
</tr>
<tr>
<td>SSYTRFΔ</td>
<td>DSYTRFΔ</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 593</td>
</tr>
<tr>
<td>CSYTRFΔ</td>
<td>ZSYTRFΔ</td>
<td></td>
</tr>
<tr>
<td>CHETRFΔ</td>
<td>ZHETRFΔ</td>
<td></td>
</tr>
<tr>
<td>SSPTRFΔ</td>
<td>DSPTRFΔ</td>
<td></td>
</tr>
<tr>
<td>CSPTRFΔ</td>
<td>ZSPTRFΔ</td>
<td></td>
</tr>
<tr>
<td>CHPTRFΔ</td>
<td>ZHPTRFΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssytrfΔ</td>
<td>LAPACKE_dsytrfΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_csytrfΔ</td>
<td>LAPACKE_zsytrfΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chetrfsΔ</td>
<td>LAPACKE_zhetrfsΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssptrfΔ</td>
<td>LAPACKE_dspsrfΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_csptrfΔ</td>
<td>LAPACKE_zsptrfΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chptrfΔ</td>
<td>LAPACKE_zhptrfΔ</td>
<td></td>
</tr>
<tr>
<td>SSYTRSΔ</td>
<td>DSYTRSΔ</td>
<td>“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 601</td>
</tr>
<tr>
<td>CSYTRSΔ</td>
<td>ZSYTRSΔ</td>
<td></td>
</tr>
<tr>
<td>CHETRSΔ</td>
<td>ZHETRSΔ</td>
<td></td>
</tr>
<tr>
<td>SSPTRSΔ</td>
<td>DSPTRSΔ</td>
<td></td>
</tr>
<tr>
<td>CSPTRSΔ</td>
<td>ZSPTRSΔ</td>
<td></td>
</tr>
<tr>
<td>CHPTRSΔ</td>
<td>ZHPTRSΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssytrsΔ</td>
<td>LAPACKE_dsytrsΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_csytrsΔ</td>
<td>LAPACKE_zsytrsΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chetrtsΔ</td>
<td>LAPACKE_zhetrtsΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssptrsΔ</td>
<td>LAPACKE_dspttrsΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_csptrsΔ</td>
<td>LAPACKE_zsptrsΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chptrsΔ</td>
<td>LAPACKE_zhptrsΔ</td>
<td></td>
</tr>
<tr>
<td>STRTRIΔ</td>
<td>DTRTRIΔ</td>
<td>“STRTRI, DTRTRI, CTRTRI, ZTRTRI, STPTRI, DTPTRI, CTPTRI, and ZTPTRI (Triangular Matrix Inverse)” on page 621</td>
</tr>
<tr>
<td>CTRTRIΔ</td>
<td>ZTRTRIΔ</td>
<td></td>
</tr>
<tr>
<td>STPTRIΔ</td>
<td>DTPTRIΔ</td>
<td></td>
</tr>
<tr>
<td>CTPTRIΔ</td>
<td>ZTPTRIΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_strtriΔ</td>
<td>LAPACKE_dtrtriΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_stptriΔ</td>
<td>LAPACKE_dtptriΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKECtrtriΔ</td>
<td>LAPACKE_ztrtriΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ctptriΔ</td>
<td>LAPACKE_ztptriΔ</td>
<td></td>
</tr>
</tbody>
</table>
Table 115: 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>SLANTRΔ</td>
<td>DLANTRΔ</td>
<td>“SLANTR, DLANTR, CLANTR, ZLANTR, SLANTP, DLANTP, CLANTP, and ZLANTP (Trapezoidal or Triangular Matrix Norm)” on page 628</td>
</tr>
<tr>
<td>CLANTRΔ</td>
<td>ZLANTRΔ</td>
<td></td>
</tr>
<tr>
<td>SLANTPΔ</td>
<td>DLANTPΔ</td>
<td></td>
</tr>
<tr>
<td>CLANTPΔ</td>
<td>ZLANTPΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_slantrΔ</td>
<td>LAPACKE_zlantrΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_clantrΔ</td>
<td>LAPACKE_clantpΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_slantpΔ</td>
<td>LAPACKE_clantpΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_clantrpΔ</td>
<td>LAPACKE_clantpΔ</td>
<td></td>
</tr>
</tbody>
</table>

Δ LAPACK

> | [GPU] This subroutine can offload computations to the GPUs when the ESSL SMP CUDA Library is used. |

Table 116: 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 493</td>
</tr>
<tr>
<td>CGEF</td>
<td>ZGEF</td>
<td></td>
</tr>
<tr>
<td>DGEFP$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SGESM</td>
<td>DGESM</td>
<td>“SGESM, DGESM, CGESM, and ZGESM (General Matrix, Its Transpose, or Its Conjugate Transpose Multiple Right-Hand Side Solve)” on page 499</td>
</tr>
<tr>
<td>CGESM</td>
<td>ZGESM</td>
<td></td>
</tr>
<tr>
<td>SGEICD</td>
<td>DGEICD</td>
<td>“SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD (General Matrix Inverse, Condition Number Reciprocal, and Determinant)” on page 511</td>
</tr>
<tr>
<td>SGEFCD</td>
<td>DGEFCD</td>
<td></td>
</tr>
<tr>
<td>SPOF</td>
<td>DPOF</td>
<td>“SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPFF, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)” on page 533</td>
</tr>
<tr>
<td>CPOF</td>
<td>ZPOF</td>
<td></td>
</tr>
<tr>
<td>SPPF</td>
<td>DPPF</td>
<td></td>
</tr>
<tr>
<td>SPPFCD</td>
<td>DPPFCD</td>
<td>“SPPFCD and DPPFCD (General Matrix Factorization, Condition Number Reciprocal, and Determinant)” on page 508</td>
</tr>
<tr>
<td>SPPF</td>
<td>DPPFP$</td>
<td></td>
</tr>
<tr>
<td>SPOSM</td>
<td>DPOSM</td>
<td>“SPOTRS, DPOTRS, CPOTRS, ZPOTRS, SPOSM, DPOSM, CPOSM, ZPOSM, SPPTRS, DPPTRS, CPPTRS, ZPPTRS, SPFF, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Multiple Right-Hand Side Solve)” on page 545</td>
</tr>
<tr>
<td>CPOSM</td>
<td>ZPOSM</td>
<td></td>
</tr>
<tr>
<td>SPPS</td>
<td>DPPS</td>
<td>“SPPS and DPPS (Positive Definite Real Symmetric Matrix Solve)” on page 552</td>
</tr>
<tr>
<td>SPPFCD</td>
<td>DPPFCD</td>
<td>“SPPFCD, DPPFCD, SPOFCD, and DPOFCD (Positive Definite Real Symmetric Matrix Factorization, Condition Number Reciprocal, and Determinant)” on page 563</td>
</tr>
<tr>
<td>SPOFCD</td>
<td>DPOFCD</td>
<td></td>
</tr>
</tbody>
</table>
Table 116: 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>SPPICD</td>
<td>DPPICD</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 569</td>
</tr>
<tr>
<td>DBSSV</td>
<td></td>
<td>“DBSSV (Symmetric Indefinite Matrix Factorization and Multiple Right-Hand Side Solve)” on page 607</td>
</tr>
<tr>
<td>DBSTRF</td>
<td></td>
<td>“DBSTRF (Symmetric Indefinite Matrix Factorization)” on page 613</td>
</tr>
<tr>
<td>DBSTRS</td>
<td></td>
<td>“DBSTRS (Symmetric Indefinite Matrix Multiple Right-Hand Side Solve)” on page 617</td>
</tr>
<tr>
<td>STRI§</td>
<td>DTRI§</td>
<td>“STRTRI, DTRTRI, CTRTRI, ZTRTRI, STPTRI, DTTPTRI, CTTPTRI, and ZTPTRI (Triangular Matrix Inverse)” on page 621</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 117: 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>SGBSVΔ</td>
<td>DGBSVΔ</td>
<td>“SGBSV, DGBSV, CGBSV, and ZGBSV (General Band Matrix Factorization and Multiple Right-Hand Side Solve)” on page 635</td>
</tr>
<tr>
<td>CGBSVΔ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgbsvΔ</td>
<td>LAPACKE_dgbsvΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgbsvΔ</td>
<td>LAPACKE_zgbsvΔ</td>
<td></td>
</tr>
<tr>
<td>SGBTRFΔ</td>
<td>DGBTRFΔ</td>
<td>“SGBTRF, DGBTRF, CGBTRF and ZGBTRF (General Band Matrix Factorization)” on page 639</td>
</tr>
<tr>
<td>CGBTRFΔ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgbtrfΔ</td>
<td>LAPACKE_dgbtrfΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgbtrfΔ</td>
<td>LAPACKE_zgbtrfΔ</td>
<td></td>
</tr>
<tr>
<td>SGBTRSΔ</td>
<td>DGBTRSΔ</td>
<td>“SGBTRS, DGBTRS, CGBTRS, and ZGBTRS (General Band Matrix Multiple Right-Hand Side Solve)” on page 643</td>
</tr>
<tr>
<td>CGBTRSΔ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgbtrsΔ</td>
<td>LAPACKE_dgbtrsΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgbtrsΔ</td>
<td>LAPACKE_zgbtrsΔ</td>
<td></td>
</tr>
<tr>
<td>Short-Precision Subroutine</td>
<td>Long-Precision Subroutine</td>
<td>Descriptive Name and Location</td>
</tr>
<tr>
<td>----------------------------</td>
<td>---------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>SPBSVΔ CPBSVΔ LAPACKE_spbsvΔ LAPACKE_cpbsvΔ</td>
<td>DPBSVΔ ZPBSVΔ LAPACKE_dpbsvΔ LAPACKE_zpbsvΔ</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 651</td>
</tr>
<tr>
<td>SPBTRFΔ CPBTRF Δ LAPACKE_spbtrfΔ LAPACKE_cpbtrfΔ</td>
<td>DPBTRFΔ ZPBTRFΔ LAPACKE_dp.ptrfΔ LAPACKE_zpbtrfΔ</td>
<td>“SPBTRF, DPBTRF, CPBTRF, and ZPBTRF (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization)” on page 656</td>
</tr>
<tr>
<td>SPBTRSΔ CPBTRS Δ LAPACKE_spbtrsΔ LAPACKE_cpbtrsΔ</td>
<td>DPBTRSΔ ZPBTRSΔ LAPACKE_dp.ptrsΔ LAPACKE_zpbtrsΔ</td>
<td>“SPBTRS, DPBTRS, CPBTRS, and ZPBTRS (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Multiple Right-Hand Side Solve)” on page 661</td>
</tr>
<tr>
<td>SGTSVΔ CGTSV Δ LAPACKE_sgtsvΔ LAPACKE_cgtsvΔ</td>
<td>DGTSAΔ ZGTSA Δ LAPACKE_dgtsvΔ LAPACKE_zgtsvΔ</td>
<td>“SGTSV, DGTSV, CGTSV, and ZGTSV (General Tridiagonal Matrix Factorization and Multiple Right-Hand Side Solve)” on page 666</td>
</tr>
<tr>
<td>SGTTRFΔ CGTTRF Δ LAPACKE_sg.ptrfΔ LAPACKE_cg.ptrfΔ</td>
<td>DGTTRFΔ ZGTTRFΔ LAPACKE_dgttrfΔ LAPACKE_zgttrfΔ</td>
<td>“SGTTRF, DGTTFR, CGTTRF, and ZGTTTRF (General Tridiagonal Matrix Factorization)” on page 670</td>
</tr>
<tr>
<td>SGTTRSΔ CGTTRS Δ LAPACKE_sg.ptrsΔ LAPACKE_cg.ptrsΔ</td>
<td>DGTTRSΔ ZGTTRSΔ LAPACKE_dgttrsΔ LAPACKE_zgttrsΔ</td>
<td>“SGTTRS, DGTTTR, CGTTRS, and ZGTTTRS (General Tridiagonal Matrix Multiple Right-Hand Side Solve)” on page 674</td>
</tr>
<tr>
<td>SPTSVΔ CPTSV Δ LAPACKE_sptsvΔ LAPACKE_cptsvΔ</td>
<td>DPTSVΔ ZPTSVΔ LAPACKE_dptsvΔ LAPACKE_zptsvΔ</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 679</td>
</tr>
<tr>
<td>SPTTRFΔ CPTTRF Δ LAPACKE_sptrfΔ LAPACKE_cpttrfΔ</td>
<td>DPTTRFΔ ZPTTRFΔ LAPACKE_dpttrfΔ LAPACKE_zpttrfΔ</td>
<td>“SPTTRF, DPTTRF, CPTTRF, and ZPTTRF (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Factorization)” on page 684</td>
</tr>
</tbody>
</table>
### Table 117: 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>SPTTRS(^\Delta)</td>
<td>DPTTRS(^\Delta)</td>
<td>“SPTTRS, DPTTRS, CPTTRS, and ZPTTRS (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Multiple Right-Hand Solve)” on page 687</td>
</tr>
<tr>
<td>CPTTRS(^\Delta)</td>
<td>ZPTTRS(^\Delta)</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spttrs(^\Delta)</td>
<td>LAPACKE_dpttrs(^\Delta)</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpttrs(^\Delta)</td>
<td>LAPACKE_zpttrs(^\Delta)</td>
<td></td>
</tr>
</tbody>
</table>

\(^\Delta\) LAPACK

### Table 118: 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(^\S)</td>
<td>DGBF(^\S)</td>
<td>“SGBF and DGBF (General Band Matrix Factorization)” on page 693</td>
</tr>
<tr>
<td>SGBS(^\S)</td>
<td>DGBS(^\S)</td>
<td>“SGBS and DGBS (General Band Matrix Solve)” on page 649</td>
</tr>
<tr>
<td>SPBF(^\S) SPBCHF(^\S)</td>
<td>DPBF(^\S) DPBCHF(^\S)</td>
<td>“SPBF, DPBF, SPBCHF, and DPBCHF (Positive Definite Symmetric Band Matrix Factorization)” on page 698</td>
</tr>
<tr>
<td>SPBS(^\S) SPBCHS(^\S)</td>
<td>DPBS(^\S) DPBCHS(^\S)</td>
<td>“SPBS, DPBS, SPBCHS, and DPBCHS (Positive Definite Symmetric Band Matrix Solve)” on page 701</td>
</tr>
<tr>
<td>SGTF(^\S)</td>
<td>DGTF(^\S)</td>
<td>“SGTF and DGTF (General Tridiagonal Matrix Factorization)” on page 704</td>
</tr>
<tr>
<td>SGTS(^\S)</td>
<td>DGTS(^\S)</td>
<td>“SGTS and DGTS (General Tridiagonal Matrix Solve)” on page 706</td>
</tr>
<tr>
<td>SGTNP CGTNP</td>
<td>DGTNP ZGTNP</td>
<td>“SGTNP, DGTNP, CGTNP, and ZGTNP (General Tridiagonal Matrix Combined Factorization and Solve with No Pivoting)” on page 708</td>
</tr>
<tr>
<td>SGTNPF CGTNPF</td>
<td>DGTNPF ZGTNPF</td>
<td>“SGTNPF, DGTNPF, CGTNPF, and ZGTNPF (General Tridiagonal Matrix Factorization with No Pivoting)” on page 711</td>
</tr>
<tr>
<td>SGTNPS CGTNPS</td>
<td>DGTNPS ZGTNPS</td>
<td>“SGTNPS, DGTNPS, CGTNPS, and ZGTNPS (General Tridiagonal Matrix Solve with No Pivoting)” on page 714</td>
</tr>
<tr>
<td>SPTF(^\S)</td>
<td>DPTF(^\S)</td>
<td>“SPTF and DPTF (Positive Definite Symmetric Tridiagonal Matrix Factorization)” on page 716</td>
</tr>
<tr>
<td>SPTS(^\S)</td>
<td>DPTS(^\S)</td>
<td>“SPTS and DPTS (Positive Definite Symmetric Tridiagonal Matrix Solve)” on page 718</td>
</tr>
</tbody>
</table>

\(^\S\) 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>
<thead>
<tr>
<th>Linear-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 720</td>
</tr>
<tr>
<td>DGSS</td>
<td>“DGSS (General Sparse Matrix or Its Transpose Solve Using Storage by Indices, Rows, or Columns)” on page 725</td>
</tr>
<tr>
<td>DGKFS</td>
<td>“DGKFS (General Sparse Matrix or Its Transpose Factorization, Determinant, and Solve Using Skyline Storage Mode)” on page 729</td>
</tr>
<tr>
<td>DGKFSP§</td>
<td></td>
</tr>
<tr>
<td>DSKFS</td>
<td>“DSKFS (Symmetric Sparse Matrix Factorization, Determinant, and Solve Using Skyline Storage Mode)” on page 744</td>
</tr>
<tr>
<td>DSKFSP§</td>
<td></td>
</tr>
<tr>
<td>DSRIS</td>
<td>“DSRIS (Iterative Linear System Solver for a General or Symmetric Sparse Matrix Stored by Rows)” on page 760</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 769</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 776</td>
</tr>
<tr>
<td>DSMGC‡</td>
<td>“DSMGC (General Sparse Matrix Iterative Solve Using Compressed-Matrix Storage Mode)” on page 782</td>
</tr>
<tr>
<td>DSDGCG</td>
<td>“DSDGCG (General Sparse Matrix Iterative Solve Using Compressed-Diagonal Storage Mode)” on page 788</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 [“9” on page 1253].
Table 120: 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>SGESVDΔ</td>
<td>DGESVDΔ</td>
<td>“SGESVD, DGESVD, CGESVD, ZGESVD, SGESDD, DGESDD, CGESDD, and ZGESDD (Singular Value Decomposition for a General Matrix)” on page 794</td>
</tr>
<tr>
<td>CGESVDΔ</td>
<td>ZGESVDΔ</td>
<td></td>
</tr>
<tr>
<td>SGESDDΔ</td>
<td>DGESDDΔ</td>
<td></td>
</tr>
<tr>
<td>CGESDDΔ</td>
<td>ZGESDDΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgesvdΔ</td>
<td>LAPACKE_dgesvdΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgesvdΔ</td>
<td>LAPACKE_zgesvdΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgesddΔ</td>
<td>LAPACKE_dgesddΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgesddΔ</td>
<td>LAPACKE_zgesddΔ</td>
<td></td>
</tr>
<tr>
<td>SGEQRFΔGPU</td>
<td>DGEQRFΔGPU</td>
<td>“SGEQRF, DGEQRF, CGEQRF, and ZGEQRF (General Matrix QR Factorization)” on page 807</td>
</tr>
<tr>
<td>CGEQRFPGPU</td>
<td>ZGEQRFΔGPU</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgeqrfΔ</td>
<td>LAPACKE_dgeqrfΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgeqrfΔ</td>
<td>LAPACKE_zgeqrfΔ</td>
<td></td>
</tr>
<tr>
<td>SGELSΔGPU</td>
<td>DGELSΔGPU</td>
<td>“SGELS, DGELS, CGELS, and ZGELS (Linear Least Squares Solution for a General Matrix)” on page 812</td>
</tr>
<tr>
<td>CGELSΔGPU</td>
<td>ZGELSΔGPU</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgelsΔ</td>
<td>LAPACKE_dgelsΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgelsΔ</td>
<td>LAPACKE_zgelsΔ</td>
<td></td>
</tr>
<tr>
<td>SGELSDΔ</td>
<td>DGELSDΔ</td>
<td>“SGELSD, DGELSD, CGELSD, and ZGELSD (Linear Least Squares Solution for a General Matrix Using the Singular Value Decomposition)” on page 822</td>
</tr>
<tr>
<td>CGELSDΔ</td>
<td>ZGELSDΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgelsdΔ</td>
<td>LAPACKE_dgelsdΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgelsdΔ</td>
<td>LAPACKE_zgelsdΔ</td>
<td></td>
</tr>
</tbody>
</table>

Δ LAPACK  

>GPU This subroutine can offload computations to the GPUs when the ESSL SMP CUDA Library is used.  

Table 121: 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>SGESVF§</td>
<td>DGESVF§</td>
<td>“SGESVF and DGESVF (Singular Value Decomposition for a General Matrix)” on page 830</td>
</tr>
<tr>
<td>SGESVS§</td>
<td>DGESVS§</td>
<td>“SGESVS and DGESVS (Linear Least Squares Solution for a General Matrix Using the Singular Value Decomposition)” on page 837</td>
</tr>
<tr>
<td>SGELLS§</td>
<td>DGELLS§</td>
<td>“SGELLS and DGELLS (Linear Least Squares Solution for a General Matrix with Column Pivoting)” on page 841</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 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 61.

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 $lna > 2nz$ on entry to DGSF does not guarantee a successful run of the program. Some programs may be terminated because of the large number of fill-ins generated upon factorization. Fill-ins generated in a program depend on the structure of each matrix. If a large number of fill-ins is anticipated when factoring a matrix, the value of $lna$ should be large enough to accommodate your problem.

Performance and Accuracy Considerations

1. To make the subroutine more efficient, an input matrix comprised of all nonzero elements is preferable. See the syntax description of each subroutine for details.
2. DGSF optionally checks the validity of the indices and pointers of the input matrix. Use of this option is suggested; however, it may affect performance. For details, see the syntax description for DGSF.
3. In DGSS, if there are multiple sparse right-hand sides to be solved, you should take advantage of the sparsity by selecting a proper value for $j_{opt}$ (such as $j_{opt} = 10$ 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 ($RPARM(1)$) affects the accuracy of the result. Specifying a larger number allows you to gain some performance; however, you may lose some accuracy in the solution.

5. In DGSF, the threshold pivot tolerance ($RPARM(2)$) is used to select pivots. A value that is close to 0.0 approaches no pivoting. A value close to 1.0 approaches partial pivoting. A value of 0.1 is considered to be a good compromise between numerical stability and sparsity.

6. If the ESSL subroutine performs storage compressions, you receive an attention message. When this occurs, the performance of this subroutine is affected. You can improve the performance by increasing the value specified for $lna$.

7. There are ESSL-specific rules that apply to the results of computations on the workstation processors using the ANSI/IEEE standards. For details, see “What Data Type Standards Are Used by ESSL, and What Exceptions Should You Know About?” on page 61.

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. If you want, you can factor a very large matrix progressively by using this option.

2. Forward elimination can be done with or without scaling the right-hand side by the diagonal matrix elements. To perform the computation without scaling, call DGKFS with the normal solve-only option, and define the upper triangular skyline matrix ($AU$) as a diagonal. To perform the computation with scaling, call DGKFS with the transpose solve-only option and define the lower triangular skyline matrix ($AL$) as a diagonal.

3. Back substitution can be done with or without scaling the right-hand side by the diagonal matrix elements. To perform the computation without scaling, call DGKFS with the transpose solve-only option, and define the upper triangular skyline matrix ($AU$) as a diagonal. To perform the computation with scaling, call DGKFS with the normal solve-only option, and define the lower triangular skyline matrix ($AL$) as a diagonal.

Performance and Accuracy Considerations

1. For optimal performance, use diagonal-out skyline storage mode for both your input and output matrices. If you specify profile-in skyline storage mode for your input matrix, and either you do not plan to use the factored output or you plan to do a solve only, it is more efficient to specify diagonal-out skyline storage mode for your output matrix. These rules apply to all the computations.

2. In some cases, elapsed time may be reduced significantly by using the combined factor-and-solve option to solve for all right-hand sides at once, in conjunction with the factorization, rather than doing the factorization and solve separately.
3. If you do a solve only, and you solve for more than one right-hand side, it is most efficient to call the skyline subroutine once with all right-hand sides, rather than once for each right-hand side.

4. The skyline subroutines allow some control over processing of the pivot (diagonal) elements of the matrix during the factorization phase. Pivot processing is controlled by IPARM(10) through IPARM(15) and RPARM(10) through RPARM(15). If a pivot occurs within a range that is designated to be fixed (IPARM(0) = 1, IPARM(10) = 1, and the appropriate element IPARM(11) through IPARM(15) = 1), it is replaced with the corresponding element of RPARM(11) through RPARM(15). Should this pivot fix-up occur, you receive an attention message. This message indicates that the matrix being factored may be unstable (singular or not definite). The results produced in this situation may be inaccurate, and you should review them carefully.

**Sparse Matrix Iterative Solver Considerations**

This 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 110.

   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 \( \varepsilon \).

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 \( \varepsilon \) 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, \((A + A^T)/2\), is not positive definite.
   - The matrix is ill-conditioned, which may cause overflows during the computation.

6. These algorithms have a tendency to generate underflows that may hurt overall performance. The system default is to mask underflow, which improves the performance of these subroutines.
Linear Least Squares Considerations

This 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 \( AX = B \), 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 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 61.

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>Table 122: Data Types.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A, B )</td>
</tr>
<tr>
<td>Subroutine</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>SGESV*</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
<tr>
<td>DGESV*</td>
</tr>
<tr>
<td>Short-precision complex</td>
</tr>
<tr>
<td>CGESV*</td>
</tr>
<tr>
<td>Long-precision complex</td>
</tr>
<tr>
<td>ZGESV*</td>
</tr>
</tbody>
</table>

* LAPACK
## Syntax

Table 123: Syntax for supported computing languages.

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGESV</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgesv</td>
</tr>
<tr>
<td>LAPACK</td>
<td>(\text{info} = \text{LAPACKE_sgesv}</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. The value of the \(\text{matrix\_layout}\) argument must be \text{LAPACK\_ROW\_MAJOR} or \text{LAPACK\_COL\_MAJOR}.

**n**
- is the order \(n\) of matrix \(\text{A}\) and the number of rows of matrix \(\text{B}\).
- Specified as: an integer; \(n \geq 0\), \(n \leq \text{lda}\), and \(n \leq \text{ldb}\).

**nrhs**
- is the number of right-hand sides; that is, the number of columns of matrix \(\text{B}\).
- Specified as: an integer; \(nrhs \geq 0\).

**a**
- is the general matrix \(\text{A}\) to be factored.
- Specified as: an \(\text{lda}\) by (at least) \(n\) array, containing numbers of the data type indicated in Table 122 on page 480.

**lda**
- is the leading dimension of the array specified for \(\text{A}\).
- Specified as: an integer; \(lda > 0\) and \(lda \geq n\).

**ipvt**
- See On Return.

**bx**
- is the general matrix \(\text{B}\), containing the \(nrhs\) right-hand sides of the system. The right-hand sides, each of length \(n\), reside in the columns of matrix \(\text{B}\).
- Specified as: an \(\text{ldb}\) by (at least) \(nrhs\) array, containing numbers of the data type indicated in Table 122 on page 480.

**ldb**
- is the leading dimension of the array specified for \(\text{B}\).
- Specified as: an integer; \(ldb > 0\) and \(ldb \geq n\).

**info**
- See On Return.

### On Return

**a**
- is the transformed matrix \(\text{A}\) 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 122 on page 480. See “Function” on page 482.

**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 ≤ ipvt(i) ≤ 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 122 on page 480.

**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 Ui is singular and its inverse could not be computed.

Returned as:

- For SGESV, DGESV, CGESV, and ZGESV returned as: an integer; info ≥ 0.
- For LAPACKE_sgesv, LAPACKE_dgesv, LAPACKE_cgesv, and LAPACKE_zgesv returned as an integer function value; info ≥ 0.

**Note:**

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 [“9” on page 1253], [“46” on page 1255], and [“75” on page 1257].
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:

```
CALL DGESV( 9 , 5 , A , 9 , IPVT, BX , 9 , INFO)
```

$A =$ (same as input $A$ in Example 1)
$BX =$ (same as input $BX$ in Example 1)

Output:

```
IPIV = (9, 9, 9, 9, 9, 9, 9, 9, 9)
```

```
A =
| 2.6  2.4  2.2  2.0  1.8  1.6  1.4  1.2  1.0 |
|  0.4  0.3  0.6  0.8  1.1  1.4  1.7  1.9  2.2 |
|  0.5 -0.4  0.4  0.8  1.2  1.6  2.0  2.4  2.8 |
|  0.5 -0.3  0.0  0.4  0.8  1.2  1.6  2.0  2.4 |
|  0.6 -0.3  0.0  0.0  0.4  0.8  1.2  1.6  2.0 |
|  0.7 -0.2  0.0  0.0  0.0  0.4  0.8  1.2  1.6 |
|  0.8 -0.1  0.0  0.0  0.0  0.0  0.4  0.8  0.8 |
|  0.9 -0.1  0.0  0.0  0.0  0.0  0.0  0.4  0.4 |

$BX =$
| 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 |
```
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:

```
CALL ZGESV( 9, 5, A, 9, IPVT, BX, 9, INFO)
```

A = (same as input A in Example 2)
IPVT = (same as input IPVT in Example 2)
BX = (same as input BX in Example 2)

Output:

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

INFO = 0

**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 124: Data Types.**

<table>
<thead>
<tr>
<th>$A$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGETRF$^*$</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGETRF$^*$</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGETRF$^*$</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGETRF$^*$</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>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>
<tr>
<td>Inverse</td>
<td>Reciprocal of the condition number</td>
</tr>
<tr>
<td>SGETRF</td>
<td>SGETRI</td>
</tr>
<tr>
<td>DGETRF</td>
<td>DGETRI</td>
</tr>
<tr>
<td>CGETRF</td>
<td>CGETRI</td>
</tr>
<tr>
<td>ZGETRF</td>
<td>ZGETRI</td>
</tr>
<tr>
<td>Reciprocal of the condition number</td>
<td>SGECON</td>
</tr>
<tr>
<td>SGETRF</td>
<td>SGECON</td>
</tr>
<tr>
<td>DGETRF</td>
<td>DGECON</td>
</tr>
<tr>
<td>CGETRF</td>
<td>CGECON</td>
</tr>
<tr>
<td>ZGETRF</td>
<td>ZGECON</td>
</tr>
</tbody>
</table>

Syntax

Table 125: Syntax for supported computing languages.

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGETRF</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgetrf</td>
</tr>
<tr>
<td>LAPACK</td>
<td>info = LAPACKE_sgetrf</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. The value of the **matrix_layout** argument 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 ≤ m ≤ lda.

**n**
the number of columns in general matrix **A** used in the computation.
Specified as: an integer; n ≥ 0.

**a**
is the m by n general matrix **A** to be factored.
Specified as: an lda by (at least) n array, containing numbers of the data type indicated in Table 124 on page 484.

**lda**
is the leading dimension of matrix **A**.
Specified as: an integer; \( \text{lda} > 0 \) and \( \text{lda} \geq m \).

**ipvt**
See On Return.

**info**
See On Return.

**On Return**

\( a \)

is the \( m \) by \( n \) transformed matrix \( A \), containing the results of the factorization. See “Function” on page 486. Returned as: an \( \text{lda} \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 124 on page 484.

**ipvt**
is the integer vector \( ipvt \) 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 \( 1 \leq \text{ipvt}(i) \leq m \).

**info**
has the following meaning:

If \( info = 0 \), the factorization of general matrix \( A \) completed successfully.

If \( info > 0 \), \( info \) is set equal to the first \( i \) where \( U_{ii} \) is singular and its inverse could not be computed.

Returned as:

- For \( \text{SGGETRF}, \text{DGETRF}, \text{CGETRF} \) and \( \text{ZGETRF} \) returned as: an integer; \( info \geq 0 \).
- For \( \text{LAPACKE_sgetrf}, \text{LAPACKE_dgetrf}, \text{LAPACKE_cgetrf} \) and \( \text{LAPACKE_zgetrf} \), returned as an integer function value; \( info \geq 0 \).

Returned as: an integer; \( info \geq 0 \).

**Note:**

1. In your C program, argument \( info \) must be passed by reference.
2. The matrix \( A \) and vector \( ipvt \) must have no common elements; otherwise results are unpredictable.
3. The way these subroutines handle singularity differs from LAPACK. 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, matrix \( A \) conforms to LAPACK format.

**Function**
The matrix \( A \) is factored using Gaussian elimination with partial pivoting to compute the \textit{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 \[^{9}\] on page 1253,\[^{46}\] on page 1255, and \[^{75}\] on page 1257.\[^{486}\]
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:

\[
\begin{align*}
\text{M} & \quad \text{N} & \quad \text{A} & \quad \text{LDA} & \quad \text{IPVT} & \quad \text{INFO} \\
\hline
5 & 5 & \text{A} & 5 & \text{IPVT} & \text{INFO} \\
\hline
\text{CALL DGETRF( 9 , 9 , A,  9 , IPVT, INFO )}
\end{align*}
\]

\[
\begin{bmatrix}
1.0 & 1.2 & 1.4 & 1.6 & 1.8 & 2.0 & 2.2 & 2.4 & 2.6 \\
1.2 & 1.0 & 1.2 & 1.4 & 1.6 & 1.8 & 2.0 & 2.2 & 2.4 \\
1.4 & 1.2 & 1.0 & 1.2 & 1.4 & 1.6 & 1.8 & 2.0 & 2.2 \\
1.6 & 1.4 & 1.2 & 1.0 & 1.2 & 1.4 & 1.6 & 1.8 & 2.0 \\
1.8 & 1.6 & 1.4 & 1.2 & 1.0 & 1.2 & 1.4 & 1.6 & 1.8 \\
2.0 & 1.8 & 1.6 & 1.4 & 1.2 & 1.0 & 1.2 & 1.4 & 1.6 \\
2.2 & 2.0 & 1.8 & 1.6 & 1.4 & 1.2 & 1.0 & 1.2 & 1.4 \\
2.4 & 2.2 & 2.0 & 1.8 & 1.6 & 1.4 & 1.2 & 1.0 & 1.2 \\
2.6 & 2.4 & 2.2 & 2.0 & 1.8 & 1.6 & 1.4 & 1.2 & 1.0
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
2.6 & 2.4 & 2.2 & 2.0 & 1.8 & 1.6 & 1.4 & 1.2 & 1.0 \\
0.4 & 0.3 & 0.6 & 0.8 & 1.1 & 1.4 & 1.7 & 1.9 & 2.2 \\
0.5 & -0.4 & 0.4 & 0.8 & 1.2 & 1.6 & 2.0 & 2.4 & 2.8 \\
0.5 & -0.3 & 0.0 & 0.4 & 0.8 & 1.2 & 1.6 & 2.0 & 2.4 \\
0.6 & -0.3 & 0.0 & 0.4 & 0.8 & 1.2 & 1.6 & 2.0 & 2.4 \\
0.7 & -0.2 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.4 \\
0.8 & -0.2 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.4 \\
0.8 & -0.1 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.4 \\
0.9 & -0.1 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.4 \\
\end{bmatrix}
\]

\[
\begin{align*}
\text{IPVT} & \quad = \quad (9, 9, 9, 9, 9, 9, 9, 9, 9) \\
\text{INFO} & \quad = \quad 0
\end{align*}
\]

Example 2
   This example shows a factorization of a complex general matrix $A$ of order 9.
Call Statement and Input:

```
M   N   A  LDA  IPVT  INFO
|   |   |   |     |     |
CALL ZGETRF( 9 , 9 , A,  9 , IPVT, INFO )
```

| (2.0, 1.0) (2.4,-1.0) (2.8,-1.0) (3.2,-1.0) (3.6,-1.0) (4.0,-1.0) (4.4,-1.0) (4.8,-1.0) (5.2,-1.0) |
| (2.4, 1.0) (2.8, 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) |
| (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) |

```
A    =  | (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) |
```

```
CALL SGETRF( 9 , 9 , A,  9 , IPVT, INFO )
```

Output:

```
A    =  | 4.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000 |
| 0.0000  5.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000 |
| 0.0000  0.0000  6.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000 |
| 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  1.0000  1.0000 |
```

Example 3

This example shows a factorization of a real general matrix $A$ of order 9.

Call Statement and Input:

```
M   N   A  LDA  IPVT  INFO
|   |   |   |     |     |
CALL SGETRF( 9 , 9 , A,  9 , IPVT, INFO )
```

Output:

```
A    =  | 4.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000 |
| 0.0000  5.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000 |
| 0.0000  0.0000  6.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000 |
| 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  1.0000  1.0000 |
```

IPVT     =  (9, 9, 9, 9, 9, 9, 9, 9, 9)
INFO     =  0

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:
CGETRS and ZGETRS solve one of the following systems of equations for multiple right-hand sides:

1. $AX = B$
2. $A^TX = B$
3. $A^H X = B$

In the formulas above:

- $A$ represents the general matrix $A$ containing the $LU$ factorization.
- $B$ represents the general matrix $B$ containing the right-hand sides in its columns.
- $X$ represents the general matrix $B$ containing the solution vectors in its columns.

These subroutines use the results of the factorization of matrix $A$, produced by a preceding call to SGETRF, DGETRF, CGETRF, or ZGETRF, respectively.

**Table 126: Data Types.**

<table>
<thead>
<tr>
<th>$A, B$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGETRS*</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGETRS*</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGETRS*</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGETRS*</td>
</tr>
</tbody>
</table>

\* LAPACK

**Note:** The input to these solve subroutines must be the output from the factorization subroutines SGETRF, DGETRF, CGETRF and ZGETRF, respectively.

**Syntax**

**Table 127: Syntax for supported computing languages.**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGETRS</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgetrs</td>
</tr>
<tr>
<td>LAPACKE</td>
<td>info = LAPACKE_sgetrs</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. The value of the `matrix_layout` argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

- **transa** indicates the form of matrix $A$ to use in the computation, where:
  - If `transa = ‘N’`, $A$ is used in the computation, resulting in solution 1.
If \( transa = 'T', A^T \) is used in the computation, resulting in solution 2.
If \( transa = 'C', A^H \) 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; \( n \geq 0 \).

\( nrhs \)

the number of right-hand sides—that is, the number of columns in 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 SGETRF, DGETRF, CGETRF, or ZGETRF, respectively.
Specified as: an \( lda \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 126 on page 489.

\( lda \)

is the leading dimension of the array specified for \( a \).
Specified as: an integer; \( lda > 0 \) and \( lda \geq n \).

\( ipvt \)

is the integer vector \( ipvt \) of length \( n \), containing the pivot indices produced by a preceding call to SGETRF, DGETRF, CGETRF, or ZGETRF, respectively.
Specified as: a one-dimensional array of (at least) length \( n \), containing integers, where \( 1 \leq ipvt(i) \leq n \).

\( bx \)

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 matrix \( B \).
Specified as: an \( ldb \) by (at least) \( nrhs \) array, containing numbers of the data type indicated in Table 126 on page 489.

\( ldb \)

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

\( bx \)

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 126 on page 489.

\( info \)

\( info \) 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; \( info \geq 0 \).
• For LAPACKE_sgetrs, LAPACKE_dgetrs, LAPACKE_cgetrs, and LAPACKE_zgetrs, returned as an integer function value; \( info \geq 0 \).
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 “Vector concepts” on page 71.
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. \( \mathbf{AX} = \mathbf{B} \)
2. \( \mathbf{A}^T \mathbf{X} = \mathbf{B} \) (only for CGETRS and ZGETRS)

where \( \mathbf{A} \), \( \mathbf{B} \), and \( \mathbf{X} \) are general matrices. These subroutines use the results of the factorization of matrix \( \mathbf{A} \), 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 484.

If \( n = 0 \) or \( \text{nrhs} = 0 \), no computation is performed and the subroutine returns after doing some parameter checking. See references [“9” on page 1253, [“46” on page 1255], and [“75” on page 1257].

Error conditions

Computational Errors

None

Note: If the factorization performed by SGETRF, DGETRF, CGETRF or ZGETRF failed because a pivot element is zero, the results returned by this subroutine are unpredictable, and there may be a divide-by-zero program exception message.

Input-Argument Errors

1. transa ≠ 'N', 'T', or 'C'
2. \( n < 0 \)
3. \( \text{nrhs} < 0 \)
4. \( n > \text{lda} \)
5. \( \text{lda} \leq 0 \)
6. \( n > \text{ldb} \)
7. \( \text{ldb} \leq 0 \)

Examples

Example 1

This example shows how to solve the system \( \mathbf{AX} = \mathbf{B} \), where matrix \( \mathbf{A} \) is the same matrix factored in the Example 1 for DGETRF.
Call Statement and Input:

```
TRANSA N NRHS A LDA IPIV BX LDB INFO
|     |    |   |   |     |   |    |    |
CALL DGETRS('N', 9, 5, A, 9, IPIV, BX, 9, INFO)
```

IPVT = (9, 9, 9, 9, 9, 9, 9, 9, 9)
A = (same as output A in Example 1)

<table>
<thead>
<tr>
<th></th>
<th>93.0</th>
<th>186.0</th>
<th>279.0</th>
<th>372.0</th>
<th>465.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>84.4</td>
<td>168.8</td>
<td>253.2</td>
<td>337.6</td>
<td>422.0</td>
</tr>
<tr>
<td></td>
<td>76.6</td>
<td>153.2</td>
<td>229.8</td>
<td>306.4</td>
<td>383.0</td>
</tr>
<tr>
<td></td>
<td>78.0</td>
<td>146.0</td>
<td>210.0</td>
<td>280.0</td>
<td>350.0</td>
</tr>
<tr>
<td></td>
<td>65.0</td>
<td>130.0</td>
<td>195.0</td>
<td>260.0</td>
<td>325.0</td>
</tr>
<tr>
<td></td>
<td>62.0</td>
<td>124.0</td>
<td>186.0</td>
<td>248.0</td>
<td>310.0</td>
</tr>
<tr>
<td></td>
<td>61.4</td>
<td>122.8</td>
<td>184.2</td>
<td>245.6</td>
<td>307.0</td>
</tr>
<tr>
<td></td>
<td>63.6</td>
<td>127.2</td>
<td>190.8</td>
<td>254.4</td>
<td>318.0</td>
</tr>
<tr>
<td></td>
<td>69.0</td>
<td>138.0</td>
<td>207.0</td>
<td>276.0</td>
<td>345.0</td>
</tr>
</tbody>
</table>

Output:

<table>
<thead>
<tr>
<th></th>
<th>1.0</th>
<th>2.0</th>
<th>3.0</th>
<th>4.0</th>
<th>5.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.0</td>
<td>4.0</td>
<td>6.0</td>
<td>8.0</td>
<td>10.0</td>
</tr>
<tr>
<td></td>
<td>3.0</td>
<td>6.0</td>
<td>9.0</td>
<td>12.0</td>
<td>15.0</td>
</tr>
<tr>
<td></td>
<td>4.0</td>
<td>8.0</td>
<td>12.0</td>
<td>16.0</td>
<td>20.0</td>
</tr>
<tr>
<td></td>
<td>5.0</td>
<td>10.0</td>
<td>15.0</td>
<td>20.0</td>
<td>25.0</td>
</tr>
<tr>
<td></td>
<td>6.0</td>
<td>12.0</td>
<td>18.0</td>
<td>24.0</td>
<td>30.0</td>
</tr>
<tr>
<td></td>
<td>7.0</td>
<td>14.0</td>
<td>21.0</td>
<td>28.0</td>
<td>35.0</td>
</tr>
<tr>
<td></td>
<td>8.0</td>
<td>16.0</td>
<td>24.0</td>
<td>32.0</td>
<td>40.0</td>
</tr>
<tr>
<td></td>
<td>9.0</td>
<td>18.0</td>
<td>27.0</td>
<td>36.0</td>
<td>45.0</td>
</tr>
</tbody>
</table>

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:

```
CALL ZGETRS('N', 9, 5, A, 9, IPIV, BX, 9, INFO)
```

IPVT = (9, 9, 9, 9, 9, 9, 9, 9, 9)
A = (same as output A in Example 2)

<table>
<thead>
<tr>
<th></th>
<th>(193.0, -10.6)</th>
<th>(200.0, 21.8)</th>
<th>(207.0, 54.2)</th>
<th>(214.0, 86.6)</th>
<th>(221.0, 119.0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(173.0, -9.4)</td>
<td>(178.0, 26.2)</td>
<td>(183.0, 49.8)</td>
<td>(188.0, 79.4)</td>
<td>(193.0, 109.0)</td>
</tr>
<tr>
<td></td>
<td>(156.0, -5.4)</td>
<td>(159.0, 22.2)</td>
<td>(162.0, 49.8)</td>
<td>(165.0, 77.4)</td>
<td>(168.0, 105.0)</td>
</tr>
<tr>
<td></td>
<td>(141.0, 1.4)</td>
<td>(142.0, 27.8)</td>
<td>(143.0, 54.2)</td>
<td>(144.0, 80.6)</td>
<td>(145.0, 107.0)</td>
</tr>
<tr>
<td></td>
<td>(129.0, 11.0)</td>
<td>(128.0, 37.0)</td>
<td>(127.0, 63.0)</td>
<td>(126.0, 89.0)</td>
<td>(125.0, 115.0)</td>
</tr>
<tr>
<td></td>
<td>(121.0, 23.4)</td>
<td>(118.0, 49.8)</td>
<td>(115.0, 76.2)</td>
<td>(112.0, 102.6)</td>
<td>(109.0, 129.0)</td>
</tr>
<tr>
<td></td>
<td>(117.0, 38.6)</td>
<td>(112.0, 66.2)</td>
<td>(107.0, 93.8)</td>
<td>(102.0, 121.4)</td>
<td>(97.0, 149.0)</td>
</tr>
<tr>
<td></td>
<td>(120.0, 56.6)</td>
<td>(113.0, 86.2)</td>
<td>(106.2, 115.8)</td>
<td>(99.2, 145.4)</td>
<td>(92.2, 175.0)</td>
</tr>
<tr>
<td></td>
<td>(129.0, 77.4)</td>
<td>(120.0, 109.8)</td>
<td>(111.0, 142.2)</td>
<td>(102.0, 174.6)</td>
<td>(93.0, 207.0)</td>
</tr>
</tbody>
</table>
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 128: Data Types.

<table>
<thead>
<tr>
<th>Data Types</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

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGEF</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgef</td>
</tr>
</tbody>
</table>

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 128 on page 493.

- **lda**
  - is the leading dimension of the array specified for \( a \).
  - Specified as: an integer; \( lda > 0 \) and \( lda \geq n \).
is the order of matrix \( A \).

Specified as: an integer; \( 0 \leq n \leq \text{lda} \).

\textit{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 494. Returned as: an \( \text{lda} \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 128 on page 493.

\textit{ipvt}

is the integer vector \( \text{ipvt} \) 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 \( iopt = 0 \) 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 (\textit{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 \textit{ipvt} contains the pivots representing permutation \( P \), such that \( A = PLU \).

If \( n \) is 0, no computation is performed. See references [“46” on page 1255] and [“48” on page 1255].

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 65.

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:

```
A  LDA  N   IPVT
   |   |    |
CALL SGEF( A , 9 , 9 , IPVT )
```

```
A = | 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  1.0 |
    | 0.0  0.0  0.0  0.0  8.0  1.0  1.0  1.0  1.0 |
    | 0.0  0.0  0.0  0.0  0.0  9.0  1.0  1.0  1.0 |
    | 0.0  0.0  0.0  0.0  0.0  0.0  10.0 11.0 12.0 |
```

Output:

```
A = | 4.0000  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  0.0000  0.0000 |
    | 0.0000  0.0000  6.0000  1.0000  1.0000  1.0000  1.0000  1.0000 |
    | 0.0000  0.0000  0.0000  7.0000  1.0000  1.0000  1.0000  1.0000 |
    | 0.0000  0.0000  0.0000  0.0000  8.0000  1.0000  1.0000  1.0000 |
    | 0.0000  0.0000  0.0000  0.0000  0.0000  9.0000  1.0000  1.0000 |
    | 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000 10.0000 11.0000 |
    | 0.2500  0.1500  0.1000  0.0714  0.0536  -0.0694  -0.0306  0.1806 |
    | 0.2500  0.1500  0.1000  0.0714  -0.0714  -0.0556  -0.0194  0.9385 |
```

```
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 )
```

```
A = | (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) |
```

Output:

```
A = | (8.0000, 5.0000) (1.0000, 9.0000) (6.0000, 6.0000) (8.0000, 1.0000) |
    | (0.0000, 0.0000) (1.0000, 1.0000) (2.0000, 2.0000) (3.0000, 3.0000) |
    | (0.0000, 0.0000) (1.0000, 1.0000) (2.0000, 2.0000) (3.0000, 3.0000) |
    | (0.2500, 0.1500) (0.1000, 0.0714) (0.0536, -0.0694) (0.0306, 0.1806) |
    | (0.2500, 0.1500) (0.1000, 0.0714) (0.0536, -0.0694) (0.0306, 0.1806) |
    | (0.2500, 0.1500) (0.1000, 0.0714) (0.0536, -0.0694) (0.0306, 0.1806) |
    | (0.2500, 0.1500) (0.1000, 0.0714) (0.0536, -0.0694) (0.0306, 0.1806) |
    | (0.2500, 0.1500) (0.1000, 0.0714) (0.0536, -0.0694) (0.0306, 0.1806) |
    | (0.2500, 0.1500) (0.1000, 0.0714) (0.0536, -0.0694) (0.0306, 0.1806) |
```

```
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/DGEP/DGEFCD, CGEF, or ZGEF, respectively.

Table 129: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A, b, x$</td>
<td></td>
</tr>
<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/DGEP/DGEFCD, CGEF, and ZGEF, respectively.

Syntax

Language | 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/DGEP/DGEFCD, CGEF, or ZGEF, respectively. Specified as: an lda by (at least) $n$ array, containing numbers of the data type indicated in Table 129 on page 496.

**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**
- is the integer vector $ipvt$ of length $n$, containing the pivot indices produced by a preceding call to SGEF/SGEFCD, DGEF/DGEP/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 129 on page 496.
**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 129 on page 496.

**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 “Vector concepts” on page 71.

**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 493.

If $n$ is 0, no computation is performed. See references [“46” on page 1255] and [“48” on page 1255].

**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:

A LDA N IPVT BX IOPT
CALL SGES( A , 9 , 9 , IPVT , BX , 0 )

IPVT = (3, 4, 5, 6, 7, 8, 9, 8, 9)
BX = (4.0, 5.0, 9.0, 10.0, 11.0, 12.0, 12.0, 12.0, 33.0)
A = (same as output A in Example 1)

Output:
BX = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)

Part 2
This part of the example shows how to solve the system $A^T x = b$, where matrix $A$ is the input matrix factored in Example 1 for SGES and DGEF. Most of the input is the same in Part 2 as in Part 1.

Call Statement and Input:

A LDA N IPVT BX IOPT
CALL SGES( A , 9 , 9 , IPVT , BX , 1 )

IPVT = (3, 4, 5, 6, 7, 8, 9, 8, 9)
BX = (6.0, 8.0, 10.0, 12.0, 13.0, 14.0, 15.0, 15.0, 15.0)
A = (same as output A in Example 1)

Output:
BX = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)

Example 2

Part 1
This part of the example shows how to solve the system $A x = b$, where matrix $A$ is the same matrix factored in the Example 2 for CGES and ZGEF.

Call Statement and Input:

A LDA N IPVT BX IOPT
CALL CGES( A , 4 , 4 , IPVT , BX , 0 )

IPVT = (4, 4, 3, 4)
BX = ((-10.0, 85.0), (-6.0, 61.0), (10.0, 38.0),
(58.0, 168.0))
A = (same as output A in Example 1)

Output:
BX = ((9.0, 0.0), (5.0, 1.0), (1.0, 6.0), (3.0, 4.0))

Part 2
This part of the example shows how to solve the system $A^H x = b$, where matrix $A$ is the input matrix factored in Example 2 for CGES and ZGEF. Most of the input is the same in Part 2 as in Part 1.

Call Statement and Input:

A LDA N IPVT BX IOPT
CALL CGES( A , 4 , 4 , IPVT , BX , 1 )
IPVT = (4, 4, 3, 4)
BX = ((71.0, 12.0), (123.0, -70.0), (68.0, 7.0))
A = (same as output A in Example 1)

Output:

\[
\text{BX} = ((9.0, 0.0), (5.0, 1.0), (1.0, 6.0), (3.0, 4.0))
\]

**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. \( A^TX = B \)

CGESM and ZGESM solve one of the following:

1. \( AX = B \)
2. \( A^TX = B \)
3. \( A^HX = B \)

These subroutines use the results of the factorization of matrix \( A \), produced by a preceding call to SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, or ZGEF, respectively.

**Table 130: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A, B, X )</td>
<td></td>
</tr>
<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 SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, and ZGEF, respectively.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGESM</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgesm</td>
</tr>
</tbody>
</table>

**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 $\text{transa} = 'C'$, $A^H$ is used in the computation, resulting in equation 3.

Specified as: a single character. It must be 'N', 'T', or 'C'.

$a$

is the factorization of matrix $A$, produced by a preceding call to SGEF/SGEFCD, DGEF/DGEFP/DGEFCD, CGEF, or ZGEF, respectively. Specified as: an $lda$ by (at least) $n$ array, containing numbers of the data type indicated in Table 130 on page 499.

$lda$

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; $0 \leq n \leq lda$.

$ipvt$

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 130 on page 499.

$ldb$

is the leading dimension of the array specified for $b$.

Specified as: an integer; $ldb > 0$ and $ldb \geq n$.

$nrhs$

is the number of right-hand sides in the system to be solved.

Specified as: an integer; $nrhs \geq 0$.

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 130 on page 499.

Notes

1. For SGESM and DGESM, if you specify 'C' for the $\text{trans}$ 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 “Vector concepts” on page 71.
**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 factored, see “SGEF, DGEF, CGEF, and ZGEF (General Matrix Factorization)” on page 493.

If \( n \) or \( nrhs \) is 0, no computation is performed. See references [“46” on page 1255] and [“48” on page 1255].

**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 factored in the Example 1 for SGEF and DGEF.

**Call Statement and Input:**

```plaintext
CALL SGESM('N', A, 9, 9, IPVT, BX, 9, 2)
```

**IPVT = (3, 4, 5, 6, 7, 8, 9, 8, 9)**

**A = (same as output A in Example 1)**

```
  [  4.0  10.0 ]
  [   5.0  15.0 ]
  [   9.0  24.0 ]
  [  10.0  35.0 ]
  [  11.0  48.0 ]
  [  12.0  63.0 ]
  [  12.0  78.0 ]
  [  12.0  78.0 ]
  [  33.0 266.0 ]
```

**BX =**

```
  [  11.0  48.0 ]
  [  12.0  63.0 ]
  [  12.0  78.0 ]
  [  33.0 266.0 ]
```
Example 1

This part of the example shows how to solve the system $\mathbf{A}^T \mathbf{x} = \mathbf{b}$ for two right-hand sides, where matrix $\mathbf{A}$ is the input matrix factored in Example 1 for SGEF and DGEF.

Call Statement and Input:

```c
CALL SGESM('T', A, 9, 9, IPVT, BX, 9, 2)
```

IPVT = (3, 4, 5, 6, 7, 8, 9, 8, 9)

A = (same as output A in Example 1)

Output:

$$
\begin{bmatrix}
1.0 & 1.0 \\
1.0 & 2.0 \\
1.0 & 3.0 \\
1.0 & 4.0 \\
1.0 & 5.0 \\
1.0 & 6.0 \\
1.0 & 7.0 \\
1.0 & 8.0 \\
1.0 & 9.0 \\
\end{bmatrix}
$$

Example 2

Part 1

This part of the example shows how to solve the system $\mathbf{A} \mathbf{x} = \mathbf{b}$ for two right-hand sides, where matrix $\mathbf{A}$ is the same matrix factored in the Example 2 for CGEF and ZGEF.

Call Statement and Input:

```c
CALL CGESM('N', A, 4, 4, IPVT, BX, 4, 2)
```

IPVT = (4, 4, 3, 4)

A = (same as output A in Example 2)

Output:

$$
\begin{bmatrix}
6.0 & 15.0 \\
8.0 & 26.0 \\
10.0 & 40.0 \\
12.0 & 57.0 \\
13.0 & 76.0 \\
14.0 & 97.0 \\
15.0 & 120.0 \\
15.0 & 125.0 \\
15.0 & 129.0 \\
\end{bmatrix}
$$
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 2 for CGEF and ZGEF.

Call Statement and Input:

```
TRANS  A  LDA  N   IPVT   BX  LDB  NRHS
CALL CGESM( 'T' , A , 4 , 4 , IPVT , BX , 4 ,   2  )
```  

$IPVT = (4, 4, 3, 4)$  

$A = $ (same as output $A$ in Example 2)

Output:

```
BX = | (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) |
```  

Part 3:

This part of the example shows how to solve the system $A^H X = B$ for two right-hand sides, where matrix $A$ is the input matrix factored in Example 2 for CGEF and ZGEF.

Call Statement and Input:

```
TRANS  A  LDA  N   IPVT   BX  LDB  NRHS
CALL CGESM( 'C' , A , 4 , 4 , IPVT , BX , 4 ,   2  )
```  

$IPVT = (4, 4, 3, 4)$  

$A = $ (same as output $A$ in Example 2)

Output:

```
BX = | (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) |
```
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 484.

Table 131: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$, work</td>
<td>SGECON*</td>
</tr>
<tr>
<td>anorm, rcond, rwork</td>
<td>DGECON*</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>CGECON*</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>ZGECON*</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td></td>
</tr>
<tr>
<td>Long-precision complex</td>
<td></td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGECON</td>
</tr>
<tr>
<td></td>
<td>CALL CGECON</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgecon</td>
</tr>
<tr>
<td></td>
<td>cgecon</td>
</tr>
<tr>
<td>LAPACK</td>
<td>info = LAPACKE_sgecon</td>
</tr>
<tr>
<td></td>
<td>info = LAPACKE_cgecon</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. The value of the `matrix_layout` argument must be LAPACK_ROW_MAJOR or 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 = 'O'` or `'1'`, the one norm is used in the computation.
- If `norm = 'I'`, the infinity norm is used in the computation.

Specified as: a single character; `norm = 'O', '1', or 'I'.`
The order of the factored general matrix \( \mathbf{A} \) used in the computation.

Specified as: an integer; \( n \geq 0 \).

\( \mathbf{A} \) is the general matrix \( \mathbf{A} \), containing the factorization of matrix \( \mathbf{A} \) produced by a preceding call to SGETRF, DGETRF, CGETRF, or ZGETRF, respectively.

Specified as: an \( \mathbf{a} \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 131 on page 504.

\( \text{lda} \) is the leading dimension of matrix \( \mathbf{A} \).

Specified as: an integer; \( \text{lda} > 0 \) and \( \text{lda} \geq n \).

\( \text{anorm} \) has the following meaning:

If \( \text{norm} = 'O' \) or '1', then \( \text{anorm} \) is the one norm of the original matrix.

If \( \text{norm} = 'I' \), then \( \text{anorm} \) is the infinity norm of the original matrix.

**Note:** You may obtain the value of \( \text{anorm} \) by a preceding call to SLANGE, DLANGE, CLANGE, or ZLANGE, respectively. Refer to “SLANGE, DLANGE, CLANGE, and ZLANGE (General Matrix Norm)” on page 518.

Specified as: a number \( \geq 0.0 \), of the data type indicated in Table 131 on page 504.

\( \text{rcond} \) See On Return.

\( \text{work} \) is the work area used by this subroutine, where:

For SGECON and DGECON

The size of \( \text{work} \) is (at least) of length \( 4n \).

For CGECON and ZGECON

The size of \( \text{work} \) is (at least) of length \( 2n \).

Specified as: an area of storage containing numbers of data type indicated in Table 131 on page 504.

\( \text{iwork} \) is a work area used by this subroutine whose size is (at least) of length \( n \).

Specified as: an area of storage containing integers.

\( \text{rwork} \) is a work area used by this subroutine whose size is (at least) of length \( 2n \).

Specified as: an area of storage containing numbers of the data type indicated in Table 131 on page 504.

\( \text{info} \) See On Return.

**On Return**

\( \text{rcond} \) has the following meaning:

If \( \text{info} = 0 \), an estimate of the reciprocal of the condition number of general matrix \( \mathbf{A} \) is returned; i.e., \( \text{rcond} = 1.0/(\text{NORM}(\mathbf{A}) \times \text{NORM}(\mathbf{A}^{-1})) \).

If \( n = 0 \), the subroutines return with \( \text{rcond} = 1.0 \).

If \( n \neq 0 \) and \( \text{anorm} = 0.0 \), the subroutines return with \( \text{rcond} = 0.0 \).

Returned as: a number \( \geq 0.0 \), of the data type indicated in Table 131 on page 504.
**info**

has the following meaning:

If \( \text{info} = 0 \), the computation completed normally.

Returned as:

- For SGECON, DGECON, CGECON, and ZGECON returned as: an integer; \( \text{info} \geq 0 \).
- For LAPACK\_sgecon, LAPACK\_dgecon, LAPACK\_cgecon, and LAPACK\_zgecon returned as an integer function value; \( \text{info} \geq 0 \).

**Notes**

1. In your C program, arguments \( rcond \) and \( \text{info} \) must be passed by reference.
2. This subroutine accepts lowercase letters for the \( \text{norm} \) argument.
3. The scalar data specified for input argument \( n \) must be the same for SLANGE/DLANGE/CLANCE/ZLANGE, SGETRF/DGETRF/CGETRF/ZGETRF, and SGECON/DGECON/CGECON/ZGECON. In addition, the scalar data specified for input argument \( m \) in SLANGE/DLANGE/CLANCE/ZLANGE and SGETRF/DGETRF/CGETRF/ZGETRF must be the same as input argument \( n \) in SLANGE/DLANGE/CLANCE/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 \( \text{rcond} = 1.0 \).

If \( n \neq 0 \) and \( \text{anorm} = 0.0 \), the subroutines return with \( \text{rcond} = 0.0 \).

See reference [“84” on page 1257].

**Error conditions**

**Resource Errors**

None.

**Computational Errors**

None.

**Input-Argument Errors**

1. \( \text{norm} \neq 'O', '1', \text{or} 'I' \)
2. \( n < 0 \)
3. \( n > \text{lda} \)
4. \( \text{lda} \leq 0 \)
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**

\[ \text{big} \text{ and tiny} \text{ have the following values:} \]
\[ \text{big} = 2^{1023} \times (1 \text{ - ULP}) \]
\[ \text{tiny} = 2^{-1022} \times (2^{49}) \]

For **ZGECON**

\[ \text{big} \text{ and tiny} \text{ have the following values:} \]
\[ \text{big} = 2^{1023} \times (1 \text{ - ULP}) \]
\[ \text{tiny} = 2^{-1022} \times (2^{50}) \]

Where 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:

```
NORM  M   N   A  LDA  WORK
|   |   |   |     |     |
ANORM = DLANGE( '1', 9 , 9 , A , 9 , WORK )

M   N   A  LDA  IPVT  INFO
|   |   |     |     |     |
CALL DGETRF(  9 , 9 , A , 9 , IPVT, INFO )

NORM  N   A  LDA  ANORM  RCOND  WORK  IWORK  INFO
|   |   |   |     |      |      |     |      |
CALL DGECON( '1', 9 , A , 9 , ANORM, RCOND, WORK, IWORK, INFO )

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:

```
NORM  M   N   A  LDA  WORK
```
ANORM = ZLANGE( '1', 4, 4, A, 4, RWORK )

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 × 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 132: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, aux, rcond, det</td>
<td>SGEFCD</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>SGEFCD</td>
</tr>
<tr>
<td>Long-precision real</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

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGEFCD</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgefcd</td>
</tr>
</tbody>
</table>

On Entry
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 132 on page 508.

$lda$

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; $iopt = 0, 1, 2, or 3$.

$rcond$

See On Return.

$det$

See On Return.

$aux$

has the following meaning:

If $naux = 0$ and error 2015 is unrecoverable, $aux$ is ignored.

Otherwise, it is is a storage work area used by this subroutine. Its size is specified by $naux$.

Specified as: an area of storage, containing numbers of the data type indicated in Table 132 on page 508.

$naux$

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, SGEFCD and DGEFCD dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, $naux \geq n$.

On Return

$a$

is the transformed matrix $A$ of order $n$, containing the results of the factorization. See “Function” on page 510. Returned as: an $lda$ by (at least) $n$ array, containing numbers of the data type indicated in Table 132 on page 508.

$ipvt$

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.
**rcond**

is an estimate of the reciprocal of the condition number, *rcond*, of matrix *A*. Returned as: a number of the data type indicated in Table 132 on page 508; *rcond* ≥ 0.

**det**

is the vector *det*, containing the two components, *det*₁ and *det*₂, of the determinant of matrix *A*. The determinant is:

\[ det_1 \left(10^{det_2}\right) \]

where 1 ≤ *det*₁ < 10. Returned as: an array of length 2, containing numbers of the data type indicated in Table 132 on page 508.

**Notes**

1. In your C program, argument *rcond* must be passed by reference.
2. When *iopt* = 0, these subroutines provide the same function as a call to SGEF or DGEF, respectively.
3. You have the option of having the minimum required value for *naux* dynamically returned to your program. For details, see “Using Auxiliary Storage in ESSL” on page 49.
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 [“83” on page 1257] and [“84” on page 1257].

If *n* is 0, no computation is performed. See reference [“46” on page 1255].

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 65.

**Input-Argument Errors**

1. \( \text{lda} \leq 0 \)
2. \( n < 0 \)
3. \( n > \text{lda} \)
4. \( \text{iopt} \neq 0, 1, 2, \text{ or } 3 \)
5. 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 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 \( 5.436(10^{-5}) \), which is computed by SGEICD and DGEICD. (See Example 3.) On output, the value in \( \text{det} \), \( |A| \), 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)

\( \text{IPVT} = (3, 4, 5, 6, 7, 8, 9, 8, 9) \)

\( \text{RCOND} = 0.00005436 \)

\( \text{DET} = (3.36, 2.00) \)

**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 133: 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* and SGEICD</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGETRI* and DGEICD</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGETRI*</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGETRI*</td>
</tr>
</tbody>
</table>

*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

Table 134: Syntax for supported computing languages.

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGETRI</td>
</tr>
<tr>
<td></td>
<td>CALL SGEICD</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgetri</td>
</tr>
<tr>
<td></td>
<td>sgeicd</td>
</tr>
<tr>
<td>LAPACKE</td>
<td>info = LAPACKE_sgetri</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. The value of the matrix_layout argument 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, or 3, it is matrix A of order n, whose inverse, reciprocal of condition number, and determinant are computed.

If iopt = 4, it is the transformed matrix A of order n, resulting from the factorization performed in a previous call to SGEF/SGEFCD or DGEF/DGEFCD/DGEFP, respectively, whose inverse is computed.

Specified as: an lda by (at least) n array, containing numbers of the data type indicated in Table 133 on page 512.
**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; \( 0 \leq n \leq lda \).

**iopt**

indicates the type of computation to be performed, where:

If \( iopt = 0 \), the inverse is computed for matrix \( A \).

If \( iopt = 1 \), the inverse and the reciprocal of the condition number are computed for matrix \( A \).

If \( iopt = 2 \), the inverse and the determinant are computed for matrix \( A \).

If \( iopt = 3 \), the inverse, the reciprocal of the condition number, and the determinant are computed for matrix \( A \).

If \( iopt = 4 \), the inverse is computed using the factored matrix \( A \).

Specified as: an integer; \( iopt = 0, 1, 2, 3, 4 \).

**rcond**

See On Return.

**det**

See On Return.

**aux**

has the following meaning, and its size is specified by \( naux \):

If \( iopt = 0, 1, 2, \) or \( 3 \), then if \( naux = 0 \) and error 2015 is unrecoverable, \( aux \) is ignored. Otherwise, it is the storage work area used by this subroutine.

If \( iopt = 4 \), \( aux \) has the following meaning:

- For SGEICD, the first \( n \) (32-bit integer arguments) or \( 2n \) (64-bit integer arguments) locations in \( aux \) must contain the \( ipvt \) integer vector of length \( n \), resulting from a previous call to SGEF, SGETRF, or SGEFCD.

- For DGEICD, the first \( \lceil n/2 \rceil \) (32-bit integer arguments) or \( n \) (64-bit integer arguments) locations in \( aux \) must contain the \( ipvt \) integer vector of length \( n \), resulting from a previous call to DGEF, DGETRF, DGEFCD, or DGEFP.

Specified as: an area of storage, containing numbers of the data type indicated in Table 133 on page 512.

**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 133 on page 512.

$lwork$

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 $work_1$. 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, $lwork \geq 100*n$.

$info$

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 133 on page 512.

$rcond$

is the reciprocal of the condition number, $rcond$, of matrix $A$. Returned as: a real number of the data type indicated in Table 133 on page 512; $rcond \geq 0$.

$det$

is the vector $det$, containing the two components $det_1$ and $det_2$ of the determinant of matrix $A$. The determinant is:

$$det_1 \left(10^{det_2}\right)$$

where $1 \leq det_1 < 10$.Returned as: an array of length 2, containing numbers of the data type indicated in Table 133 on page 512.

$work$

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 133 on page 512. Except for $work_1$, 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/SGEFCD or DGEF/DGEFCD/DGEPF, 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 49.
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:

$$det_1\left(10^{det_2}\right)$$

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 [“46” on page 1255], [“48” on page 1255], and [“54” on page 1256].

Error conditions

Resource Errors

1. Unable to allocate internal work area.
2. If $iopt = 0, 1, 2, \text{ or } 3$, 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 65.

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 \( naux \neq 0, \) 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 computes the inverse of matrix \( A \), where matrix \( A \) is the transformed matrix factored by SGETRF in Example 3 and the input contents of IPVT are the same as the output contents of IPVT in Example 3.

Note: Because \( lwork \) is 0, SGETRI dynamically allocates the work area used by this subroutine.

Call Statement and Input:

```
CALL SGETRI( 9 , A , 9 , IPVT , WORK , 0 , INFO )
```

- \( A = \) (same as output \( A \) in Example 3)
- \( IPVT = \) (same as output \( IPVT \) in Example 3)
Example 2

This example computes the inverse of matrix $A$, where $A$ is the transformed matrix factored by ZGETRF in Example 2 and the input contents of IPVT are the same as the output contents of IPVT in Example 2.

**Note:** Because $lwork$ is 0, ZGETRI dynamically allocates the work area used by this subroutine.

**Call Statement and Input:**

```
CALL ZGETRI( 9 , A , 9 , IPVT , WORK , 0 , INFO )
```

$A$ = (same as output $A$ in Example 2)

IPVT = (same as output IPVT in Example 2)

**Output:**

```
A    =  
      | (-0.2, -0.4) (-0.1,  0.1) (-0.1,  0.1) ( 0.0,  0.1) ( 0.1,  0.1) ( 0.1,  0.0) ( 0.1,  0.0) ( 0.0, -0.3) |
      | ( 0.2,  0.4) ( 0.0, -0.6) (-0.1,  0.0) (-0.1,  0.0) (-0.1,  0.0) (-0.1,  0.0) (-0.1,  0.0) (-0.1,  0.0) |
      | ( 0.0,  0.0) ( 0.2,  0.4) ( 0.0, -0.6) (-0.1,  0.0) (-0.1,  0.0) (-0.1,  0.0) (-0.1,  0.0) (-0.1,  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) (-0.1,  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) ( 0.2,  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) ( 0.2,  0.4) ( 0.0, -0.6) (-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) |
```

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.
Example 4

This example computes the inverse of matrix $A$, where: $iopf = 4$; 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:

```
A  LDA  N  IOPT  RCOND   DET   AUX   NAUX
|   |   |   |      |      |     |     |
CALL SGEICD( A , 9 , 9 , 4  , RCOND , DET , AUX , 300 )
```

```
A =
| 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  0.000  0.143 -0.143  0.000 |
| -1.000  1.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000 |
| -226.000 266.000  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 |
```

Output:

```
| 0.333 -0.667  0.333  0.000  0.000  0.000  0.042 -0.042  0.000 |
| 56.833 -52.167 -1.167 -0.500 -0.500 -0.357  6.836 -0.479 -0.500 |
| -55.167 51.833  0.833  0.500  0.500  0.214 -6.735  0.521  0.500 |
| -1.000  1.000  0.000  0.000  0.000  0.000  0.143 -0.143  0.000 |
| -1.000  1.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000 |
| -226.000 266.000  5.000  3.000  2.000  1.429 -27.179  1.750  2.000 |
| 560.000 -520.000 -10.000 -6.000 -4.000 -2.857  67.857 -5.000 -5.000 |
| -325.000 305.000  5.000  3.000  2.000  1.429 -39.554  3.125  3.000 |
```

RCOND    =  0.00005436
DET      =  (3.36, 2.00)

---

**SLANGE, DLANGE, CLANGE, and ZLANGE (General Matrix Norm)**

**Purpose**

SLANGE, DLANGE, CLANGE, and ZLANGE compute the norm of general matrix $A$. 

---

518 IBM Engineering and Scientific Subroutine Library for Linux on POWER  Version 6.1: ESSL Guide and Reference
Table 135: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Work, Result</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SLANGE*</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DLANGE*</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>CLANGE*</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>ZLANGE*</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>SLANGE</td>
</tr>
<tr>
<td>C and C++</td>
<td>slange</td>
</tr>
<tr>
<td>LAPACK</td>
<td>LAPACKE_slange</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. The value of the `matrix_layout` argument 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 \( \mathbf{A} \) is computed.
  
  If `norm = 'I'`, the infinity norm of \( \mathbf{A} \) is computed.
  
  If `norm = 'F'` or `'E'`, the Frobenius or Euclidean norm of \( \mathbf{A} \) is computed.
  
  If `norm = 'M'`, the absolute value of the matrix element having the largest absolute value, i.e., \( \text{max}(|\mathbf{A}|) \), is returned.

  Specified as: a single character; `norm = 'O', '1', 'I', 'F', 'E', or 'M'`.

- **m**
  
  the number of rows in matrix \( \mathbf{A} \).

  Specified as: an integer; \( m \geq 0 \).

- **n**
  
  the number of columns in matrix \( \mathbf{A} \).

  Specified as: an integer; \( n \geq 0 \).

- **a**
  
  is the general matrix \( \mathbf{A} \), with \( m \) rows and \( n \) columns.

  Specified as: an \( lda \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 135 on page 519.

- **lda**
  
  is the leading dimension of matrix \( \mathbf{A} \).
Specified as: an integer; \( lda > 0 \) and \( lda \geq m \).

\textbf{work}

is the work area used by this subroutine, where:

- When \( \text{norm} = 'I' \), 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 135 on page 519.

\textbf{On Return}

\textit{Function value}

is the result of the norm computation, returned as a number of the data type indicated in Table 135 on page 519.

If \( \text{norm} = 'O' \) or \( '1' \), the one norm of \( A \) is returned.
If \( \text{norm} = 'I' \), 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.

\textbf{Notes}

1. Declare this function in your program as returning a value of the data type indicated in Table 135 on page 519.
2. This function accepts lowercase letters for the \( \text{norm} \) argument.

\textbf{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>'I'</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.

\textbf{Error conditions}

\textbf{Resource Errors}

None.

\textbf{Computational Errors}

None.

\textbf{Input-Argument Errors}

1. \( \text{norm} \neq 'O', '1', 'I', 'F', 'E', \) 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:

```
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:

```
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>
<thead>
<tr>
<th>( A, B )</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SPPSV*</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DPPSV*</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CPPSV*</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZPPSV*</td>
</tr>
</tbody>
</table>
Table 136: Data Types. (continued)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A, B</td>
<td>Subroutine</td>
</tr>
<tr>
<td></td>
<td>△ LAPACK</td>
</tr>
</tbody>
</table>

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SPPSV</td>
</tr>
<tr>
<td>C and C++</td>
<td>sppsv</td>
</tr>
<tr>
<td>LAPACK</td>
<td>info = LAPACKE_sppsv</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. The value of the matrix_layout argument 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 ≥ 0.

nrhs

is the number of right-hand sides; that is, the number of columns of matrix B.
Specified as: an integer; nrhs ≥ 0.

ap

is an 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 (at least) length n(n+1)/2, containing numbers of the data type indicated in Table 136 on page 521.

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 136 on page 521.

ldb

is the leading dimension of the array specified for B.
Specified as: an integer; ldb > 0 and ldb ≥ n.
**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 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 136 on page 521. See “Function” on page 523.

*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 136 on page 521.

*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 ≥ 0.
- For LAPACKE_sppsv, LAPACKE_dppsv, LAPACKE_cppsv, and LAPACKE_zppsv, 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. The matrices used in this computation must have no common elements; otherwise, results are unpredictable. See “Vector concepts” on page 71.
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 84.
   - For positive definite complex Hermitian matrices, see “Positive Definite or Negative Definite Complex Hermitian Matrix” on page 85.
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:

$$ A = LL^T \text{ or } A = U^U $$

for SPPSV and DPPSV

$$ A = LL^H \text{ or } A = U^H U $$

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 [“9” on page 1253], [“46” on page 1255], and [“48” on page 1255].

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. $uplo \neq 'U' \text{ or } 'L'$
2. $n < 0$
3. $nrhs < 0$
4. $n > ldb$
5. $ldb \leq 0$

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:

```
CALL SPPSV ('L', 9, 2, AP, BX, 9, INFO)
```

AP = (same as input AP in Example 5)
BX = (same as input BX in Example 5)

Output:

```
AP = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
```

```
BX = |
    | 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 |
```

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:

```
[ 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 |
  1.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 |
  1.0 2.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0 |
  1.0 2.0 3.0 4.0 4.0 4.0 4.0 4.0 4.0 |
  1.0 2.0 3.0 4.0 5.0 5.0 5.0 5.0 5.0 |
  1.0 2.0 3.0 4.0 5.0 6.0 6.0 6.0 6.0 |
  1.0 2.0 3.0 4.0 5.0 6.0 7.0 7.0 7.0 |
  1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 8.0 |
  1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 ]
```

On output, all elements of this matrix \( A \) are 1.0.

**Note:** The AP array is formatted in a triangular arrangement for readability; however, it is stored in upper-packed storage mode.

Call Statement and Input:

```
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)
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:

$$
\begin{bmatrix}
25.0 & 0.0 & -5.0 & -5.0 & 10.0 & 10.0 \\
-5.0 & 51.0 & 0.0 & 4.0 & -6.0 \\
10.0 & -5.0 & 71.0 & 0.0 \\
\end{bmatrix}
$$

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 )
```

AP = (same as input AP in Example 7)
BX = (same as input BX in Example 7)

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))
BX = [ (2.0, -1.0) (2.0, 0.0) ]
  [ (1.0, 1.0) (-1.0, 2.0) ]
  [ (0.0, -2.0) (1.0, 1.0) ]
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:

\[
\begin{pmatrix}
(9.0, 0.0) & (3.0, 3.0) & (3.0, -3.0) \\
(3.0, -3.0) & (18.0, 0.0) & (8.0, -6.0) \\
(3.0, 3.0) & (8.0, 6.0) & (43.0, 0.0)
\end{pmatrix}
\]

**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, they are set to zero.

Call Statement and Input:

```
CALL ZPPSV ('U', 3, 2, AP, BX, 3, INFO )
```

AP = (same as input AP in Example 8)

BX = (same as input BX in Example 8)

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))
```

```
BX = \[
\begin{pmatrix}
(2.0, -1.0) & (2.0, 0.0) \\
(1.0, -1.0) & (0.0, 1.0) \\
(3.0, 0.0) & (1.0, -1.0)
\end{pmatrix}
\]
```

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>
<thead>
<tr>
<th>Table 137: Data Types.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A, B</strong></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>

\textsuperscript{*} LAPACK

Linear algebraic equations subroutines 527
### Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SPOSV</td>
</tr>
<tr>
<td>C and C++</td>
<td>sposv</td>
</tr>
<tr>
<td>LAPACK</td>
<td>info = LAPACKE_sposv</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. The value of the `matrix_layout` argument 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` ≥ 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 137 on page 527. See “Notes:” on page 529.

- **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 137 on page 527.

- **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 137 on page 527. See “Function” on page 529.

\(bx\)

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 137 on page 527.

\(info\)

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 \(\geq 0\).
• For LAPACKE_sposv, LAPACKE_dposv, LAPACKE_cposv, and LAPACKE_zposv, 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. 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 84.
   • For positive definite complex Hermitian matrices, see “Positive Definite or Negative Definite Complex Hermitian Matrix” on page 85.
7. The matrices used in this computation must have no common elements; otherwise, results are unpredictable. See “Vector concepts” on page 71.

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:
A=LL^T or A=U^T U
for SPOSV and DPOSV
A=LL^H or A=U^H U
for CPOSV and ZPOSV

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 [“9” on page 1253], [“46” on page 1255], and [“84” on page 1257].

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 ≠ 'U' or 'L'
2. n < 0
3. nrhs < 0
4. n > lda
5. lda ≤ 0
6. n > ldb
7. ldb ≤ 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 = (same as input A in Example 1)
BX = (same as input BX in Example 1)

Output:

| 1.0  | .  | .  | .  | .  | .  | .  |
| 1.0  | 1.0 | .  | .  | .  | .  | .  |
| 1.0  | 1.0 | 1.0 | .  | .  | .  | .  |
Example 2

This example shows how to solve the system $A\mathbf{x} = \mathbf{b}$, 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)
```

$A =$ (same as input $A$ in Example 2)
$B X =$ (same as input $B X$ in Example 2)

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

BX =
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

INFO = 0
```

Example 3

This example shows how to solve the system $A\mathbf{x} = \mathbf{b}$, 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)
```

$$A = \begin{bmatrix}
25.0 & 0.0 \\
-5.0 & -5.0 \\
10.0 & 5.0
\end{bmatrix} \begin{bmatrix}
10.0 \\
51.0 \\
71.0
\end{bmatrix}$$

$$BX = \begin{bmatrix}
60.0 & -55.0 \\
34.0 & 58.0 \\
13.0 & -152.0
\end{bmatrix} \begin{bmatrix}
70.0 \\
-51.0 \\
75.0
\end{bmatrix}$$

Output:

$$A = \begin{bmatrix}
5.0 & 0.0 \\
-1.0 & 1.0 \\
2.0 & -1.0
\end{bmatrix} \begin{bmatrix}
10.0 \\
4.0 \\
8.0
\end{bmatrix}$$

$$BX = \begin{bmatrix}
2.0 & -1.0 \\
1.0 & -1.0 \\
0.0 & -2.0
\end{bmatrix} \begin{bmatrix}
1.0 \\
1.0 \\
1.0
\end{bmatrix}$$

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:

```
CALL CPOSV('U', 3, 2, A, 3, BX, 3, INFO)
```

$$A = \begin{bmatrix}
9.0 & 0.0 \\
3.0 & 3.0 \\
3.0 & 3.0
\end{bmatrix} \begin{bmatrix}
3.0 \\
18.0 \\
43.0
\end{bmatrix}$$

$$BX = \begin{bmatrix}
33.0 & -18.0 \\
45.0 & -45.0 \\
152.0 & 1.0
\end{bmatrix} \begin{bmatrix}
15.0 \\
8.0 \\
43.0
\end{bmatrix}$$

Output:

$$A = \begin{bmatrix}
3.0 & 0.0 \\
3.0 & -3.0 \\
3.0 & -3.0
\end{bmatrix} \begin{bmatrix}
1.0 \\
4.0 \\
2.0
\end{bmatrix}$$

**Note:** The strict lower part of $A$ is not referenced.
SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPPF, 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, DPOF, or ZPOF 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, ZPOTRF, SPOF, or DPOF 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>
<thead>
<tr>
<th>Data Type</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SPOTRF, SPO, SPF, SPPTRF, and SPPF</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DPOTRF, DPO, DPF, DPPTRF, and DPF</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CPOTRF, CPO, and CPPTRF'</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZPOTRF, ZPO, and ZPPTRF'</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>SPF</td>
<td>SPS</td>
</tr>
<tr>
<td>DPF</td>
<td>DPP</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SPOTRF</td>
</tr>
<tr>
<td></td>
<td>CALL SPOF</td>
</tr>
<tr>
<td></td>
<td>CALL SPTRF</td>
</tr>
<tr>
<td></td>
<td>CALL SPTF</td>
</tr>
<tr>
<td>Language</td>
<td>Syntax</td>
</tr>
<tr>
<td>----------</td>
<td>--------</td>
</tr>
</tbody>
</table>
| C and C++ | `spotrf | dpotrf | cpotrf | zpotrf (uplo, n, a, lda, info);`  
          | `spof | dpof | cpof | zpof (uplo, a, lda, n);`  
          | `spptrf | dpptrf | cpptrf | zpptrf (uplo, n, a, lda, info);`  
          | `sppf | dppf (ap, n, iopt);`  
| LAPACKE  | `info = LAPACKE_spotrf | LAPACKE_dpotrf | LAPACKE_cpotrf | LAPACKE_zpotrf (matrix_layout, uplo, n, a, lda);`  
          | `nfo = LAPACKE_spptrf | LAPACKE_dpptrf | LAPACKE_cpptrf | LAPACKE_zpptrf (matrix_layout, uplo, n, a, lda);` |

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. The value of the `matrix_layout` argument 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 139 on page 534. See “Notes” on page 537.

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 139 on page 534.

**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`. 

Linear algebraic equations subroutines  535
**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, or 11.

**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 SPPTRF, DPPTRF, CPPTRF, and ZPPTRF:**

The transformed matrix is stored in upper-packed or lower-packed storage mode.

**For 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 139 on page 534.

**For SPPTRF, DPPTRF, CPPTRF, and ZPPTRF:**

The array contains at least $n(n+1)/2$ elements.

**For 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 537 and see “Function” on page 537.

**a**

is the transformed matrix $A$ of order $n$, containing the results of the factorization. See “Function” on page 537.

Returned as: a two-dimensional array, containing numbers of the data type indicated in Table 139 on page 534.

**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 SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPOF, DPOF, CPOF, ZPOF, SPPTRF, DPPTRF, CPPTRF, ZPPTRF, SPPF, and DPPF, returned as: an integer; $info \geq 0$.
- For LAPACKE_spotrf, LAPACKE_dpotrf, LAPACKE_cpotrf, LAPACKE_zpotrf, LAPACKE_spptrf, LAPACKE_zpttrf, LAPACKE_zpptrf, and LAPACKE_zpptrf, 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 \( 1 \), 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 \[“63” on page 1256\], \[“79” on page 1257\], and \[“81” on page 1257\]).

   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 84.
   - For positive definite complex Hermitian matrices, see “Positive definite or Negative Definite Complex Hermitian Matrix” on page 85.

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:

\[
A = LL^T \quad \text{or} \quad A = U^TU \\
\text{for SPOTRF, DPOTRF, SPOF, and DPOF}
\]

\[
A = LL^H \quad \text{or} \quad A = U^HU \\
\text{for CPOTRF, ZPOTRF, CPOF, and ZPOF}
\]

where:

\( L \) is a lower triangular matrix.
\( U \) is an upper triangular matrix.

If \( n \) is 0, no computation is performed. See references \[“9” on page 1253\] and \[“46” on page 1255\].

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:

\[
A = LL^T \quad \text{or} \quad A = U^TU \\
\text{for SPPTRF and DPPTRF}
\]
\[ A = LL^H \text{ or } A = U^H U \]
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 [“9” on page 1253], [“46” on page 1255], and [“80” on page 1257].

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 [“9” on page 1253] and [“46” on page 1255].

**Error conditions**

**Resource Errors**
Unable to allocate internal work area.

**Computational Errors**

1. Matrix \( A \) is not positive definite (for SPOTRF, DPOTRF, CPOTRF, ZPOTRF, SPTRF, DPTRF, CPTRF, and ZPTRF).
   - 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.
   - \( i \) can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2104 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see “What Can You Do about ESSL Computational Errors?” on page 65.

3. Matrix \( A \) is not positive definite (for SPPF and DPPF when \( iopt = 1 \) or 11 and for SPOF, DPOF, CPOF, and ZPOF).
   - Processing stops at the first occurrence of a nonpositive definite diagonal element.
• The order \( i \) of the first minor encountered having a nonpositive determinant is identified in the computational error message.
• The return code is set to 1.
• \( i \) can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2115 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see “What Can You Do about ESSL Computational Errors?” on page 65.

**Input-Argument Errors**

1. \( \text{uplo} \neq 'U' \) or \( 'L' \)
2. \( n < 0 \)
3. \( n > \text{lda} \)
4. \( \text{lda} \leq 0 \)
5. \( \text{iopt} \neq 0, 1, 10, \) or \( 11 \)

**Examples**

**Example 1**

This example shows a factorization of the same positive definite symmetric matrix \( A \) of order 9 used in Example 9, but stored in lower storage mode.

**Call Statement and Input:**

```
UPLO N A LDA INFO
```
```
CALL SPOTRF( 'L' , 9 , A , 9 , INFO )
```

**or**

```
UPLO A LDA N
```
```
CALL SPOF( 'L' , A , 9 , 9 )
```

**Output:**

```
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 | 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 |
```

INFO = 0
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:

```
UPLO N A LDA INFO
CALL SPOTRF( 'U' , 9 , A , 9 , INFO )
```

or

```
UPLO A LDA N
CALL SPOF( 'U' , A , 9 , 9 )
```

```
1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
| .   2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0 |
| .    .   3.0  3.0  3.0  3.0  3.0  3.0  3.0 |
| .    .    .   4.0  4.0  4.0  4.0  4.0  4.0 |
| .    .    .    .   5.0  5.0  5.0  5.0  5.0 |
| .    .    .    .    .   6.0  6.0  6.0  6.0 |
| .    .    .    .    .    .   7.0  7.0  7.0 |
| .    .    .    .    .    .    .   8.0  8.0 |
| .    .    .    .    .    .    .    .   9.0 |
```

Output:

```
1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
| .   1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
| .    .   1.0  1.0  1.0  1.0  1.0  1.0  1.0 |
| .    .    .   1.0  1.0  1.0  1.0  1.0  1.0 |
| .    .    .    .   1.0  1.0  1.0  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:

```
| (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) |
```

**Note:** On input, the imaginary parts of the diagonal elements of the complex Hermitian matrix $A$ are assumed to be zero, so you do not have to set these values. On output, they are set to zero.

Call Statement and Input:

```
UPLO N A LDA INFO
CALL CPOTRF( 'L' , 3 , A , 3 , INFO )
```
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:

$$
\begin{pmatrix}
9.0 & 3.0 & 3.0 \\
3.0 & 3.0 & -3.0 \\
3.0 & 3.0 & 43.0
\end{pmatrix}
$$

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, they are set to zero.

Call Statement and Input:

```
UPLO N A LDA INFO
|   |   |   |    |
CALL CPOTRF( 'U' , 3 , A , 3 , INFO )
```

Output:

$$
\begin{pmatrix}
3.0 & 1.0 & 1.0 \\
\cdot & 4.0 & 0.0 \\
\cdot & \cdot & 6.0
\end{pmatrix}
$$

INFO = 0

Example 5

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.

```
UPLO A LDA N
|   |   |   |
CALL CPOF( 'L' , A , 3 , 3 )
```

Output:

$$
\begin{pmatrix}
3.0 & 0.0 & \cdot \\
\cdot & 1.0 & 1.0 \\
\cdot & \cdot & 6.0
\end{pmatrix}
$$

INFO = 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:

```
UPLO  N   AP   INFO
|    |   |     |
CALL SPPTRF( '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:

```
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:

```
UPLO  N   AP   INFO
|    |   |    |
CALL SPPTRF( 'U',  9,  AP, INFO )
```

```
AP = (                                         1.0,
       1.0, 1.0,
       1.0, 2.0, 2.0,
       1.0, 2.0, 3.0, 3.0,
       1.0, 2.0, 3.0, 4.0, 4.0,
       1.0, 2.0, 3.0, 4.0, 5.0, 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, 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:

```
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, -5.0), (51.0, .), (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 = ((9.0, .), (3.0, 3.0), (18.0, .), (3.0, -3.0), (8.0, -6.0), (43.0, .))

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:

```
| 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 |
| 1.0 2.0 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 |
| 1.0 2.0 3.0 4.0 5.0 5.0 5.0 5.0 |
| 1.0 2.0 3.0 4.0 5.0 6.0 6.0 6.0 |
| 1.0 2.0 3.0 4.0 5.0 6.0 7.0 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 |
```

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.
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.
**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>
<thead>
<tr>
<th>Table 140: Data Types.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A, B, X</strong></td>
</tr>
<tr>
<td><strong>Subroutine</strong></td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>SPOTRS*, SPOSM, and SPPTRS*</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
<tr>
<td>DPOTRS*, DPOSM, and DPPTRS*</td>
</tr>
<tr>
<td>Short-precision complex</td>
</tr>
<tr>
<td>CPOTRS*, CPOSM, and CPPTRS*</td>
</tr>
<tr>
<td>Long-precision complex</td>
</tr>
<tr>
<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**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fortran</strong></td>
<td>CALL SPOTRS</td>
</tr>
<tr>
<td></td>
<td>CALL SPOSM</td>
</tr>
<tr>
<td></td>
<td>CALL SPPTRS</td>
</tr>
<tr>
<td><strong>C and C++</strong></td>
<td>spotrs</td>
</tr>
<tr>
<td></td>
<td>sposm</td>
</tr>
<tr>
<td></td>
<td>spptrs</td>
</tr>
<tr>
<td><strong>LAPACKE</strong></td>
<td>info = LAPACKE_spotrs</td>
</tr>
<tr>
<td></td>
<td>info = LAPACKE_spptrs</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. The value of the `matrix_layout` argument 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 140 on page 545.

**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 140 on page 545.

**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 \( A \) and the number of rows of matrix \( B \).
Specified as: an integer; \( n \geq 0 \).

**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 140 on page 545.

**ldb**
is the leading dimension of the array specified for \( b \).
Specified as: an integer; \( ldb > 0 \) and \( ldb \geq n \).

**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 140 on page 545.

**info**

**info** has the following meaning:

If $info = 0$, the solve completed successfully.

Returned as:

- For SPOTRS, DPOTRS, CPOTRS, ZPOTRS, SPOSM, DPOSM, CPOSM, ZPOMS, SPPTRS, DPPTRS, CPPTRS, and ZPPTRS, returned as: an integer; $info \geq 0$.
- For LAPACKE_spotrs, LAPACKE_dpotrs, LAPACKE_cpotrs, LAPACKE_zpotrs, LAPACKE_spptrs, LAPACKE_dpptrs, LAPACKE_cpptrs, and LAPACKE_zpptrs, 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/SPPTRF, DPOTRF/DPOF/DPOFCD/DPPTRF, CPOTRF/CPOF/CPPTRF, and ZPOTRF/ZPOF/ZPPTRF, 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 SPPTRF, DPPTRF, CPPTRF, and ZPPTRF, respectively.
6. The matrices used in this computation must have no common elements; otherwise, results are unpredictable. See “Vector concepts” on page 71.
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 84.
   - For positive definite complex Hermitian matrices, see “Positive Definite or Negative Definite Complex Hermitian Matrix” on page 85.

**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/SPPTRS and DPOTRS/DPOSM/DPPTRS, and a positive definite complex Hermitian matrix for CPOTRS/CPOSM/CPPTRS and ZPOTRS/ZPOSM/ZPPTRS. These subroutines use the results of the factorization of matrix $A$, produced by a preceding call to SPOTRF/SPOF/SPOFCD/SPPTRF, DPOTRF/DPOF/DPOFCD/DPPTRF, CPOTRF/CPOF/CPPTRF, or ZPOTRF/ZPOF/ZPPTRF, respectively. For a description of how $A$ is factored, see “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 533.

If $n$ or $nrhs$ is 0, no computation is performed. See references [“9” on page 1253] and [“46” on page 1255].

**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. uplo ≠ 'U' or 'L'
2. n < 0
3. nrhs < 0
4. n > lda
5. lda ≤ 0
6. n > ldb
7. ldb ≤ 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:

```
CALL SPOTRS( 'L' , 9 , 2 , A , 9 , BX , 9 , INFO  )
```

or

```
CALL SPOSM( 'L' , A , 9 , 9 , BX , 9 ,  2  )
```

\( A = \) (same as output \( A \) in Example 1)

```
|  9.0 |  45.0 |
| 17.0 |  89.0 |
| 24.0 | 131.0 |
| 38.0 | 176.0 |
| 35.0 | 205.0 |
| 39.0 | 235.0 |
| 42.0 | 259.0 |
| 44.0 | 276.0 |
| 45.0 | 285.0 |
```

BX =

```
|  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 |
```

Output:

```
|  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 |
```

INFO = 0

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:

```
CALL SPOTRS('U', 9, 2, A, 9, BX, 9, INFO)
```

or

```
CALL SPOSM('U', A, 9, 9, BX, 9, 2)
```

$A = \text{(same as output } A \text{ in Example 2)}$

\[
A = \begin{bmatrix}
9.0 & 45.0 \\
17.0 & 89.0 \\
24.0 & 131.0 \\
30.0 & 170.0 \\
35.0 & 285.0 \\
39.0 & 235.0 \\
42.0 & 259.0 \\
44.0 & 276.0 \\
45.0 & 285.0
\end{bmatrix}
\]

$BX = \begin{bmatrix}
35.0 & 205.0 \\
39.0 & 235.0 \\
42.0 & 259.0 \\
44.0 & 276.0 \\
45.0 & 285.0
\end{bmatrix}$

Output:

\[
BX = \begin{bmatrix}
1.0 & 1.0 \\
1.0 & 2.0 \\
1.0 & 3.0 \\
1.0 & 4.0 \\
1.0 & 5.0 \\
1.0 & 6.0 \\
1.0 & 7.0 \\
1.0 & 8.0 \\
1.0 & 9.0
\end{bmatrix}
\]

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 the 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)
```

$A = \text{(same as output } A \text{ in Example 3)}$

\[
A = \begin{bmatrix}
(60.0, -55.0) & (70.0, 10.0) \\
(34.0, 58.0) & (-51.0, 110.0) \\
(13.0, -152.0) & (75.0, 63.0)
\end{bmatrix}
\]

$BX = \begin{bmatrix}
(60.0, -55.0) & (70.0, 10.0) \\
(34.0, 58.0) & (-51.0, 110.0) \\
(13.0, -152.0) & (75.0, 63.0)
\end{bmatrix}$
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:

```fortran
CALL CPOTRS( 'U' , 3 , 2 , A , 3 , BX , 3 , INFO  )
```

or

```fortran
CALL CPOSM( 'U' , A , 3 , 3 , BX , 3 ,  2  )
```

$A = \text{(same as output } A \text{ in Example 4)}$

$B X = \begin{bmatrix}
(33.0, -18.0) & (15.0, -3.0) \\
(45.0, -45.0) & (8.0, -2.0) \\
(152.0, 1.0) & (43.0, -29.0)
\end{bmatrix}$

Output:

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

$INFO = 0$

Example 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 SPPTRF and DPPTRF.

Call Statement and Input:

```fortran
CALL SPPTRS( 'L' , 9 , 2 , AP , BX , 9 , INFO )
```

$A = \text{(same as output } A \text{ in Example 5)}$

$B X = \begin{bmatrix}
9.0 & 45.0 \\
17.0 & 89.0 \\
24.0 & 131.0 \\
30.0 & 178.0 \\
35.0 & 205.0 \\
39.0 & 235.0 \\
42.0 & 259.0 \\
44.0 & 276.0 \\
45.0 & 285.0
\end{bmatrix}$

Output:

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

$INFO = 0$
Example 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:

```
CALL SPPTRS( 'U' , 9 , 2 ,  AP , BX , 9 , INFO  )
```

$A = $ (same as output $A$ in Example 6)

```
[ 9.0  45.0 ]
[ 17.0  89.0 ]
[ 24.0 131.0 ]
[ 30.0 170.0 ]
BX   =  [ 35.0 205.0 ]
[ 39.0 235.0 ]
[ 42.0 259.0 ]
[ 44.0 276.0 ]
[ 45.0 285.0 ]
```

Output:

```
[ 1.0  1.0 ]
[ 1.0  2.0 ]
[ 1.0  3.0 ]
[ 1.0  4.0 ]
BX   =  [ 1.0  5.0 ]
[ 1.0  6.0 ]
[ 1.0  7.0 ]
[ 1.0  8.0 ]
[ 1.0  9.0 ]
```

INFO =  0

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 )
```

Output:

```
[ 1.0  1.0 ]
[ 1.0  2.0 ]
[ 1.0  3.0 ]
[ 1.0  4.0 ]
BX   =  [ 1.0  5.0 ]
[ 1.0  6.0 ]
[ 1.0  7.0 ]
[ 1.0  8.0 ]
[ 1.0  9.0 ]
```

INFO =  0
AP = (same as output AP in Example 7)

\[
\begin{bmatrix}
(60.0, & -55.0) & (70.0, & 10.0) \\
(34.0, & 58.0) & (-51.0, & 110.0) \\
(13.0, & -152.0) & (75.0, & 63.0)
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
(2.0, & -1.0) & (2.0, & 0.0) \\
(1.0, & -1.0) & (-1.0, & 2.0) \\
(0.0, & -2.0) & (1.0, & 1.0)
\end{bmatrix}
\]

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 )
```

AP = (same as output AP in Example 8)

\[
\begin{bmatrix}
(33.0, & -18.0) & (15.0, & -3.0) \\
(45.0, & -45.0) & (8.0, & -2.0) \\
(152.0, & 1.0) & (43.0, & -29.0)
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
(2.0, & -1.0) & (2.0, & 0.0) \\
(1.0, & -1.0) & (0.0, & 1.0) \\
(3.0, & 0.0) & (1.0, & -1.0)
\end{bmatrix}
\]

INFO = 0

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/SPPFC or DPPF/DPPFP/DPPFC, respectively.

Table 141: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A, b, x )</td>
<td>SPPS</td>
</tr>
<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>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SPPS</td>
</tr>
<tr>
<td>C and C++</td>
<td>spps</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 141 on page 552, 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 141 on page 552.
- **iopt** indicates the type of factorization that was performed on matrix $A$, where:
  - If $iopt = 0$, the matrix was factored using the $\text{LDL}^T$ 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 141 on page 552.

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 $iopt$</th>
<th>SPPS/DPPS Input $iopt$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 or 10</td>
<td>0</td>
</tr>
<tr>
<td>SPPF/DPPF Input ( iopt )</td>
<td>SPPS/DPPS Input ( iopt )</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------</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 “Vector concepts” on page 71.

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 80.

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 [“46” on page 1255] and [“48” on page 1255].

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:

```
       AP   N   BX   IOPT
       |   |   |   |
    CALL SPPS ( AP , 9 , BX ,  0 )
```

\( AP = \) (same as output \( AP \) in Example 9 for SPPF and DPPF)

\( BX = (9.0, 17.0, 24.0, 30.0, 35.0, 39.0, 42.0, 44.0, 45.0) \)

Output:

\( BX = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0) \)

Example 2

This example shows how to solve the same system as in Example 1, where matrix \( A \) is the same matrix factored in the Example 10 for SPPF and DPPF.
Call Statement and Input:

```
AP   N   BX  IOPT
|    |   |    |
CALL SPPS( AP , 9 , BX , 1 )
```

AP = (same as output AP in Example 10 for SPPF and DPPF)
BX = (9.0, 17.0, 24.0, 30.0, 35.0, 39.0, 42.0, 44.0, 45.0)

Output:

```
BX       =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
```

**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, SPP, and DPPF (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization)” on page 533.

**Table 142: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$, work</td>
<td>SPOCON*, SPPCON*</td>
</tr>
<tr>
<td>Short-precision real</td>
<td></td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DPOCON*, DPPCON*</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CPOCON*, CPPCON*</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZPOCON*, ZPPCON*</td>
</tr>
</tbody>
</table>
**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SPOCON</td>
</tr>
<tr>
<td></td>
<td>CALL CPOCON</td>
</tr>
<tr>
<td></td>
<td>CALL SPPCON</td>
</tr>
<tr>
<td></td>
<td>CALL CPPCON</td>
</tr>
<tr>
<td>C and C++</td>
<td>spocon</td>
</tr>
<tr>
<td></td>
<td>cpocon</td>
</tr>
<tr>
<td></td>
<td>sppcon</td>
</tr>
<tr>
<td></td>
<td>cppcon</td>
</tr>
<tr>
<td>LAPACK</td>
<td>info = LAPACKE_spocon</td>
</tr>
<tr>
<td></td>
<td>info = LAPACKE_cpocon</td>
</tr>
<tr>
<td></td>
<td>info = LAPACKE_sppcon</td>
</tr>
<tr>
<td></td>
<td>info = LAPACKE_cppcon</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.

- **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.

- **n**
  - the order of the factored matrix A used in the computation.

- **ap**
  - is an array, referred to as AP, containing the factorization of the positive definite matrix A produced by a preceding call to SPTRF, DPTRF, CPTRF, or ZPTRF, respectively, stored in upper-packed or lower-packed storage mode.

- **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.

- **lda**
  - is the leading dimension of matrix A.
**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, CLANSY, ZLANSY, CLANHE, ZLANHE, SLANSP, DLANSP, CLANHP, and ZLANHP (Real Symmetric or Complex Hermitian Matrix Norm)” on page 579.

Specified as: a number $\geq 0.0$, of the data type indicated in Table 142 on page 555.

**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 142 on page 555.

**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 142 on page 555.

**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 $anorm = 0.0$, the subroutines return with $rcond = 0.0$.

Returned as: a number $\geq 0.0$, of the data type indicated in Table 142 on page 555.

**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 SPPTRF/DPPTRF/CPPTRF/ZPPTRF.
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 SPPTRF/DPPTRF/CPPTRF/ZPPTRF.
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 84.
   • For positive definite complex Hermitian matrices, see “Positive Definite or Negative Definite Complex Hermitian Matrix” on page 85.

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 ≠ 0 and anorm = 0.0, the subroutines return with rcond = 0.0.

See reference [“84” on page 1257].

Error conditions
Resource Errors
None.
Computational Errors
None.
Input-Argument Errors
1. uplo ≠ 'U' or 'L'
2. n < 0
3. n > lda
4. \( \text{lda} \leq 0 \)
5. \( \text{anorm} < 0 \)
6. \( \text{anorm} \neq 0 \) and \( \text{anorm} > \text{big} \) or \( \text{anorm} < \text{tiny} \)

Where:

For \text{SPOCON}, \text{SPPCON}, \text{CPOCON}, \text{and CPPCON} \\
big and tiny have the following values:
\[
\text{big} = 2^{127} \times (1 - \text{ulp}) \\
\text{tiny} = 2^{126} \times 2^{21}
\]

For \text{DPOCON}, \text{DPPCON}, \text{ZPOCON}, \text{and ZPPCON} \\
big and tiny have the following values:
\[
\text{big} = 2^{1023} \times (1 - \text{ulp}) \\
\text{tiny} = 2^{1022} \times 2^{50}
\]

Where 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 positive definite real symmetric matrix \( A \) stored in lower storage mode. The input matrix \( A \) to \text{DLANSY} and \text{DPOTRF} is the same as input matrix \( A \) in Example 1.

Call Statements and Input:

<table>
<thead>
<tr>
<th>NORM</th>
<th>UPLO</th>
<th>N</th>
<th>A</th>
<th>LDA</th>
<th>WORK</th>
</tr>
</thead>
</table>
ANORM = \text{DLANSY}( '1', 'L', 9 , A , 9 , WORK )

<table>
<thead>
<tr>
<th>UPLO</th>
<th>N</th>
<th>A</th>
<th>LDA</th>
<th>INFO</th>
</tr>
</thead>
</table>
CALL \text{DPOTRF}( 'L', 9 , A , 9 , INFO )

<table>
<thead>
<tr>
<th>UPLO</th>
<th>N</th>
<th>A</th>
<th>LDA</th>
<th>ANORM</th>
<th>RCOND</th>
<th>WORK</th>
<th>IWORK</th>
<th>INFO</th>
</tr>
</thead>
</table>
CALL \text{DPOCON}( 'L', 9 , A , 9 , ANORM, RCOND, WORK, IWORK, INFO )

\( A = \) (same as output \( A \) in Example 1)
ANORM = (same as output ANORM in Example 1)

Output:

\( \text{RCOND} = 5.56 \times 10^{-3} \)
INFO = 0

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 \text{DLANSY} and \text{DPOTRF} is the same as input matrix \( A \) in Example 2.
Call Statements and Input:

\begin{verbatim}
NORM  UPLO  N   A  LDA  WORK
ANORM = DLANSY( '1', 'U', 9, A, 9, WORK )

UPLO  N   A  LDA  INFO
CALL DPOTRF( 'U', 9, A, 9, INFO )

UPLO  N   A  LDA  ANORM  RCOND  WORK  IWORK  INFO
CALL DPOCON( 'U', 9, A, 9, ANORM, RCOND, WORK, IWORK, INFO )
\end{verbatim}

\begin{verbatim}
A = (same as output A in Example 2)
ANORM = (same as output ANORM in Example 2)
\end{verbatim}

Output:

\begin{verbatim}
RCOND = 5.56 \times 10^{-3}
INFO = 0
\end{verbatim}

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 3.

Call Statements and Input:

\begin{verbatim}
NORM  UPLO  N   A  LDA  WORK
ANORM = ZLANHE( '1', 'L', 3, A, 3, RWORK )

UPLO  N   A  LDA  INFO
CALL ZPOTRF( 'L', 3, A, 3, INFO )

UPLO  N   A  LDA  ANORM  RCOND  WORK  RWORK  INFO
CALL ZPOCON( 'L', 3, A, 3, ANORM, RCOND, WORK, RWORK, INFO )
\end{verbatim}

\begin{verbatim}
A = (same as output A in Example 3)
ANORM = (same as output ANORM in Example 3)
\end{verbatim}

Output:

\begin{verbatim}
RCOND = 1.85 \times 10^{-1}
INFO = 0
\end{verbatim}

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.
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 DLANSP and DPPTRF is the same as input matrix A in Example 5.

Call Statements and Input:

```
Call Statements and Input:

| NORM | UPLO | N   | A   | LDA | WORK
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ANORM = ZLANHE( '1', 'U', 3, A, 3, RWORK )

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
CALL ZPOTRF( 'U', 3, A, 3, INFO )

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
CALL ZPOCON( 'U', 3, A, 3, ANORM, RCOND, WORK, RWORK, INFO )
```

A = (same as output A in Example 3)
ANORM = (same as output ANORM in Example 4)

Output:

RCOND = 1.01 × 10⁻¹
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 DLANSP and DPPTRF is the same as input matrix A in Example 6.

Call Statements and Input:

```
Call Statements and Input:

| NORM | UPLO | N   | A   | WORK
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ANORM = DLANSP( '1', 'L', 9, AP, WORK )

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
CALL DPPTRF( 'L', 9, AP, INFO )

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
CALL DPPOCON( 'U', 9, AP, ANORM, RCOND, WORK, IWORK, INFO )
```

AP = (same as output AP in Example 5)
ANORM = (same as output ANORM in Example 5)

Output:

RCOND = 5.56 × 10⁻³
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 DLANSP and DPPTRF is the same as input matrix A in Example 6.
Call Statements and Input:

```
NORM  UPLO  N  AP  WORK
ANORM = DLANS( '1', 'U', 9, AP, WORK )

UPLO  N  AP  INFO
CALL DPTRF( 'U', 9, AP, INFO )

UPLO  N  AP  ANORM  RCOND  WORK  IWORK  INFO
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:

\[
\text{RCOND} = 5.56 \times 10^{-3}
\]
\[
\text{INFO} = 0
\]

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:

```
NORM  UPLO  N  AP  WORK
ANORM = ZLANHP( '1', 'L', 3, AP, RW )

UPLO  N  AP  INFO
CALL ZPPTRF( 'L', 3, AP, INFO )

UPLO  N  AP  ANORM  RCOND  WORK  RW  INFO
CALL ZPPCON( 'L', 3, AP, ANORM, RCOND, WORK, RW, INFO )
```

AP = (same as output AP in Example 7)
ANORM = (same as output ANORM in Example 7)

Output:

\[
\text{RCOND} = 1.85 \times 10^{-1}
\]
\[
\text{INFO} = 0
\]

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:

```
<table>
<thead>
<tr>
<th>NORM</th>
<th>UPLO</th>
<th>N</th>
<th>AP</th>
<th>WORK</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ANORM = ZLANHP( '1', 'U', 3, AP, RWORK )
```

```
<table>
<thead>
<tr>
<th>UPLO</th>
<th>N</th>
<th>AP</th>
<th>INFO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CALL ZPPTRF( 'U', 3, AP, INFO )
```

```
<table>
<thead>
<tr>
<th>UPLO</th>
<th>N</th>
<th>AP</th>
<th>ANORM</th>
<th>RCOND</th>
<th>WORK</th>
<th>RWORK</th>
<th>INFO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CALL ZPPCON( 'U', 3, AP, ANORM, RCOND, WORK, RWORK, INFO )
```

AP = (same as output AP in Example 8)
ANORM = (same as output ANORM in Example 8)

**Output:**

RCOND = $1.01 \times 10^{-1}$
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 143: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$, aux, rcond, det</td>
<td>SPPFCD and SPOFCD</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>SPPFCD and SPOFCD</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DPPFCD and DPOFCD</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

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
</table>
| Fortran  | CALL SPPFCD | DPPFCD (ap, n, iopt, rcond, det, aux, naux)  
CALL SPOFCD | DPOFCD (uplo, a, lda, n, iopt, rcond, det, aux, naux) |
| C and C++ | sppfcd | dppfcd (ap, n, iopt, rcond, det, aux, naux);  
spofcd | dpofcd (uplo, a, lda, n, iopt, rcond, det, aux, naux); |

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 143 on page 563.

$a$
is the positive definite symmetric matrix $A$, to be factored.
Specified as: an $lda$ by (at least) $n$ array, containing numbers of the data type indicated in Table 143 on page 563.

$lda$
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 143 on page 563.

\( naux \)

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, SPOFC, and DPOFC 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 566. Returned as: a one-dimensional array of (at least) length \( n(n+1)/2+n \), containing numbers of the data type indicated in Table 143 on page 563.

\( a \)

is the transformed matrix \( A \) of order \( n \), containing the results of the factorization. See “Function” on page 566. Returned as: a two-dimensional array, containing numbers of the data type indicated in Table 143 on page 563.

\( rcond \)

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 143 on page 563; \( rcond \geq 0 \).

\( det \)

is the vector \( det \), containing the two components \( det_1 \) and \( det_2 \) of the determinant of matrix \( A \). The determinant is:

\[
\begin{align*}
det_1 \left(10^{det_2}\right) \\
\end{align*}
\]

where \( 1 \leq det_1 < 10 \). Returned as: an array of length 2, containing numbers of the data type indicated in Table 143 on page 563.

Notes

1. All subroutines accept lowercase letters for the \( uplo \) argument.
2. In your C program, argument \( rcond \) must be passed by reference.
3. When \( iopt = 0 \), SPPFCD and DPPFCD provide the same function as a call to SPPF or DPPF, respectively. When \( iopt = 0 \), SPOFC and DPOFC 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 [“63” on page 1256], [“79” on page 1257], and [“81” on page 1257]).

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 553 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 80. 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 84.

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 49.

**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 ["83" on page 1257] and ["84" on page 1257].

If \( n \) is 0, no computation is performed. See references ["46" on page 1255] and ["48" on page 1255].

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:

\[
A = LL^T \quad \text{or} \quad A = U^TU
\]

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 ["83" on page 1257] and ["84" on page 1257].

If \( n \) is 0, no computation is performed. See references ["9" on page 1253] and ["46" on page 1255].

**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 65.

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 65.

Input-Argument Errors

1. $uplo \neq 'U' \text{ or } 'L'$
2. $lda \leq 0$
3. $lda < n$
4. $n < 0$
5. $iopt \neq 0, 1, 2, \text{ or } 3$
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 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 $\|A\| = 4.9$
On output, the value in \( \text{det}, |A| \), is equal to 1.

Call Statement and Input:

```c
CALL DPPFCD( AP , 9 , 3  , RCOND , DET , AUX , 9  )
```

\( \text{AP} = (\text{same as input AP in Example 9}) \)

Output:

\( \text{AP} = (\text{same as output AP in Example 9}) \)

\( \text{RCOND} = 0.00555555 \)

\( \text{DET} = (1.0, 0.0) \)

**Example 2**

This example computes the factorization, reciprocal of the condition number, and determinant of matrix \( A \). The input is the same as used in **Example 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 \( \|A\| = 4.0 \)

On output, the value in \( \text{det}, |A| \), is equal to 1.

Call Statement and Input:

```c
CALL SPOFCD( 'L', A , 9 , 9 , 3 , RCOND , DET , AUX , 9  )
```

\( \text{A} = (\text{same as input A in Example 1}) \)

Output:

\( \text{A} = (\text{same as output A in Example 1}) \)

\( \text{RCOND} = 0.00555555 \)

\( \text{DET} = (1.0, 0.0) \)

**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 \( \|A\| = 4.0 \)

On output, the value in \( \text{det}, |A| \), is equal to 1.

Call Statement and Input:

```c
CALL SPOFCD( 'U', A , 9 , 9 , 3 , RCOND , DET , AUX , 9  )
```

\( \text{A} = (\text{same as input A in Example 2}) \)
Output:
A = (same as output A 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>
<thead>
<tr>
<th>Table 144: Data Types.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A, \text{aux}, \text{rcond}, \text{det} )</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>

`LAPACK`

**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 SPPF, DPPF, SPOF/SPOFCD, or DPOF/DPOFCD, respectively, where Cholesky factorization was performed.

**Syntax**

<table>
<thead>
<tr>
<th>Table 145: Syntax for supported computing languages.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
</tr>
</tbody>
</table>
| Fortran | CALL SPOTRI | DPOTRI | CPOTRI | ZPOTRI (uplo, n, a, lda, info)  
          CALL SPOICD | DPOICD (uplo, a, lda, n, iopt, rcond, det, aux, aux)  
          CALL SPPTRI | DPPTRI | CPPTRI | ZPPTRI (uplo, n, ap, info)  
          CALL SPPICD | DPPICD (ap, n, iopt, rcond, det, aux, aux) |
| C and C++ | spotri | dpotri | cpotri | zpotri (uplo, n, a, lda, info);  
          spoicd | dpoicd (uplo, a, lda, n, iopt, rcond, det, aux, aux);  
          spptri | dpptri | cpptri | zpptri (uplo, n, ap, info);  
          sppicd | dppicd (ap, n, iopt, rcond, det, aux, aux); |
Table 145: Syntax for supported computing languages. (continued)

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPACKE</td>
<td>info = LAPACKE_spotri</td>
</tr>
<tr>
<td></td>
<td>(matrix_layout, uplo, n, a, lda);</td>
</tr>
<tr>
<td></td>
<td>info = LAPACKE_spptri</td>
</tr>
<tr>
<td></td>
<td>(matrix_layout, uplo, n, ap);</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. The value of the `matrix_layout` argument 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 $iopt = 0, 1, 2, \text{ or } 3$, then $AP$ contains the positive definite real symmetric matrix $A$, whose inverse, condition number reciprocal, and determinant are computed, where matrix $A$ is stored in lower-packed storage mode.

If $iopt = 4$, then $AP$ contains the transformed matrix $A$ of order $n$, resulting from the Cholesky factorization performed in a previous call to SPPF or DPPF, respectively, whose inverse is computed.

Specified as: a one-dimensional array of (at least) length $n(n+1)/2$, containing numbers of the data type indicated in Table 144 on page 569.

**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 $iopt = 0, 1, 2, \text{ or } 3$, 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 144 on page 569.

**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**

indicates the type of computation to be performed, where:

If *iopt* = 0, the inverse is computed for matrix *A*.

If *iopt* = 1, the inverse and the reciprocal of the condition number are computed for matrix *A*.

If *iopt* = 2, the inverse and the determinant are computed for matrix *A*.

If *iopt* = 3, the inverse, the reciprocal of the condition number, and the determinant are computed for matrix *A*.

If *iopt* = 4, the inverse is computed for the Cholesky factored matrix *A*.

Specified as: an integer; *iopt* = 0, 1, 2, 3, or 4.

**rcond**

See On Return.

**det**

See On Return.

**aux**

has the following meaning:

If *naux* = 0 and error 2015 is unrecoverable, *aux* is ignored.

Otherwise, it is the storage work area used by this subroutine. Its size is specified by *naux*. Specified as: an area of storage, containing numbers of the data type indicated in Table 144 on page 569.

**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, 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, *naux* ≥ *n*.

**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 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 144 on page 569.
**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 144 on page 569.

**rcond**
is the reciprocal of the condition number, $rcond$, of matrix $A$. Returned as: a real number of the data type indicated in Table 144 on page 569; $rcond \geq 0$.

**det**
is the vector $det$, containing the two components $det_1$ and $det_2$ of the determinant of matrix $A$. The determinant is:

$$det_1 \left(10^{det_2}\right)$$

where $1 \leq det_1 < 10$. Returned as: an array of length 2, containing numbers of the data type indicated in Table 144 on page 569.

**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; 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; $info \geq 0$.
- 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; $info \geq 0$.

Specified as: an integer; $info \geq 0$.

**Notes:**

1. In your C program, the arguments $info$ and $rcond$ 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 $uplo$, $lda$, 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 $uplo$ argument.

5. SPPICD and DPPICD in some cases utilize algorithms based on recursive packed storage format. (See references [“63” on page 1256], [“79” on page 1257], and [“81” on page 1257]).

6. The way _POTRI and _PPTRI 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 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 80.

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 49.
**Function**

These subroutines find the inverse of positive definite matrix $A$, where:

- $A^{-1}$ is the inverse of matrix $A$, where $AA^{-1} = A^{-1}A = I$.
- For positive definite real symmetric matrix $A$:
  
  $A = LL^T$ or $U^TU$
  $A^{-1} = L^{-T}L^{-1}$ or $U^{-1}U^T$

- For positive definite complex Hermitian matrix $A$:
  
  $A = LL^H$ or $U^HU$
  $A^{-1} = L^{-H}L^{-1}$ or $U^{-1}U^H$

**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:

- $1/\left(\|A\|_1(\|A^{-1}\|_2)\right)$ 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:

$$det_1\left(10^{det_2}\right)$$

- 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 [“46” on page 1255], [“48” on page 1255], and [“54” on page 1256].

**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 $i_0pt = 1$, 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 2115 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 2. For details, see “What Can You Do about ESSL Computational Errors?” on page 65.

The inverse of matrix $A$ could not be computed.

- For $i_0pt = 4$, 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 65.

Input-Argument Errors

1. $uplo \neq 'U' \text{ or } 'L'$
2. $n < 0$
3. $lda \leq 0$
4. $lda < n$
5. $i_0pt \neq 0, 1, 2, 3, \text{ or } 4$
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 $i_0pt = 4$, and matrix $A$ is the transformed matrix factored by SPOTRF in Example 9.

Call Statement and Input:

```
<table>
<thead>
<tr>
<th>UPLO</th>
<th>N</th>
<th>A</th>
<th>LDA</th>
<th>INFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>'U'</td>
<td>9</td>
<td>A</td>
<td>9</td>
<td>INFO</td>
</tr>
</tbody>
</table>
```

CALL SPOTRI( 'U' , 9 , A , 9 , INFO )

A = (same as output A in Example 2 for SPOTRF)
Output:

\[
A = \begin{bmatrix}
2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 2.0 & -1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 2.0 & -1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 2.0 & -1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0
\end{bmatrix}
\]

INFO = 0

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:

\[
CALL \text{CPOTRI('L', 3, A, 3, INFO)}
\]

\[A = \text{(same as output A in Example 3 for CPOTRF)}\]

Output:

\[
A = \begin{bmatrix}
(0.05, 0.00) & . & . & . \\
(0.00, -0.01) & (0.02, 0.00) & . & . \\
(-0.01, 0.00) & (0.00, 0.00) & (0.02, 0.00) & . \\
\end{bmatrix}
\]

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:

\[
CALL \text{CPOTRI('U', 3, A, 3, INFO)}
\]

\[A = \text{(same as output A in Example 4 for CPOTRF)}\]

Output:

\[
A = \begin{bmatrix}
(0.13, 0.00) & (-0.02, -0.03) & (0.00, 0.01) & . \\
. & (0.07, 0.00) & (-0.01, 0.01) & . \\
. & . & (0.03, 0.00) & . \\
\end{bmatrix}
\]

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:

\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 & 2.0 \\
1.0 & 2.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 & 3.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 4.0 & 4.0 & 4.0 & 4.0 & 4.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 5.0 & 5.0 & 5.0 & 5.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 6.0 & 6.0 & 6.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 7.0 & 7.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 8.0 & 8.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 8.0 & 9.0
\end{bmatrix}
\]

The values used to compute the reciprocal of the condition number in this example are obtained with the following values:

$\| A \|_1 = \max(9.0, 17.0, 24.0, 30.0, 35.0, 39.0, 42.0, 44.0, 45.0) = 45.0$  
$\| A^{-1} \|_1 = 4.0$

On output, the value in $\det |A|$, is equal to 1, and $\text{RCOND} = 1/180$.

Call Statement and Input:

```
CALL SPOICD( 'U' , A , 9 , 9 , 3 , RCOND , DET , AUX , 9 )
```

Output:

```
\[
\begin{bmatrix}
2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
. & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
. & . & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
. & . & . & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
. & . & . & . & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 \\
. & . & . & . & . & 2.0 & -1.0 & 0.0 & 0.0 \\
. & . & . & . & . & . & 2.0 & -1.0 & 0.0 \\
. & . & . & . & . & . & . & 2.0 & -1.0 \\
. & . & . & . & . & . & . & . & 1.0
\end{bmatrix}
\]
```

$\text{RCOND} = 0.005555556$  
$\text{DET} = (1.0, 0.0)$

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:

```
CALL SPPTRI( 'L' , 9 , AP , INFO )
```

Output:

```
\[
\begin{bmatrix}
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 \\
. & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
. & . & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
. & . & . & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
. & . & . & . & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 \\
. & . & . & . & . & 2.0 & -1.0 & 0.0 & 0.0 \\
. & . & . & . & . & . & 2.0 & -1.0 & 0.0 \\
. & . & . & . & . & . & . & 2.0 & -1.0 \\
. & . & . & . & . & . & . & . & 1.0
\end{bmatrix}
```

$\text{RCOND} = 0.005555556$  
$\text{DET} = (1.0, 0.0)$
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:

```fortran
CALL SPPTRI( 'U' , 9 , AP , INFO )
```

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:

```fortran
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 )
```

AP = (same as output AP in Example 8 for ZPPTRF)

Output:

```
AP = ((0.13, 0.0) (-0.02, -0.03) (0.07, 0.00) (0.00, 0.01) (-0.01, 0.01) (0.03, 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, 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 = (2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
    2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
    2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
    2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
    2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
    2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
    2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
    2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
    2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
    1.0)
RCOND = 0.005556
DET = (1.0, 0.0)
```

**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:

```
| AP | N | IOPT | RCOND | DET | AUX | NAUX |
-----------------------|---|------|--------|-----|-----|------|
CALL SPPICD(AP, 9, 4, RCOND, DET, AUX, 9)
```

AP = (same as output AP in Example 10 for SPPF)

Output:

```
AP = (2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
     2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
     2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
     2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
     2.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
     2.0, -1.0, 0.0, 0.0, 0.0,
     2.0, -1.0, 0.0,
     2.0, -1.0,
     1.0)
```

SLANSY, DLANSY, CLANSY, ZLANSY, CLANHE, ZLANHE, SLANSP, DLANSP, CLANHP, and ZLANHP (Real Symmetric or Complex Hermitian Matrix Norm)

**Purpose**

These subprograms compute the norm of matrix $A$ as explained below:

**SLANSY, DLANSY, CLANSY, ZLANSY, CLANHE, ZLANHE, SLANSP, DLANSP, CLANHP, and ZLANHP**

These subprograms compute the norm of matrix $A$, stored in upper or lower storage mode, where:

- For SLANSY and DLANSY, $A$ is a real symmetric matrix.
- For CLANSY and ZLANSY, $A$ is a complex symmetric matrix.
- For CLANHE, ZLANHE, $A$ is a complex Hermitian matrix.

**Table 146: 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*, &gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CLANSY*&lt;</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>ZLANHE*, ZLANHP*, &gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZLANSY*&lt;</td>
</tr>
</tbody>
</table>

Δ LAPACK
### Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fortran</strong></td>
<td>SLANSY</td>
</tr>
<tr>
<td></td>
<td>SLANSP</td>
</tr>
<tr>
<td><strong>C and C++</strong></td>
<td>slansy</td>
</tr>
<tr>
<td></td>
<td>slanhp</td>
</tr>
<tr>
<td><strong>LAPACKE</strong></td>
<td>LAPACKE_slansy</td>
</tr>
<tr>
<td></td>
<td>LAPACKE_slanhp</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. The value of the `matrix_layout` argument 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 `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', 'I', '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; $n \geq 0$.

**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 146 on page 579.

**a**
is the matrix $A$, stored in upper or lower storage mode.
Specified as: an \( \text{lda} \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 146 on page 579.

\( \text{lda} \)  
is the leading dimension of matrix \( A \).  
Specified as: an integer; \( \text{lda} > 0 \) and \( \text{lda} \geq n \).  

\( \text{work} \)  
is the work area used by this subroutine, where:  
- When \( \text{norm} = 'O', '1', \) or \( 'I' \), the size the of \( \text{work} \) argument is (at least) of length \( n \).  
- Otherwise, \( \text{work} \) is not referenced.  
Specified as: an area of storage containing numbers of data type indicated in Table 146 on page 579.

On Return  
**Function value**  
is the result of the norm computation, returned as a number of the data type indicated in Table 146 on page 579.

If \( \text{norm} = 'O' \) or \( '1' \), the one norm of \( A \) is returned.  
If \( \text{norm} = 'I' \), 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 \( n = 0 \), the function returns zero.

**Notes**  
1. Declare this function in your program as returning a value of the data type indicated in Table 146 on page 579.  
2. This function accepts lowercase letters for the \( \text{norm} \) and \( \text{uplo} \) 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 the \( \text{norm} \) argument:

<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>'I'</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, that is, ( \max(</td>
</tr>
</tbody>
</table>

If \( n = 0 \), the function returns zero.

**Error conditions**

**Resource Errors**
None.

**Computational Errors**
None.
Input-Argument Errors

1. \( \text{norm} \neq 'O', '1', 'I', 'F', 'E', \text{or} 'M' \)
2. \( \text{uplo} \neq 'U' \text{or} 'L' \)
3. \( n < 0 \)
4. \( n > \text{lda} \)
5. \( \text{lda} \leq 0 \)

Examples

Example 1

This example computes the one norm of real symmetric matrix \( \mathbf{A} \) stored in lower storage mode.

Call Statements and Input:

\[
\begin{array}{cccccc}
\text{NORM} & \text{UPLO} & N & A & LDA & \text{WORK} \\
\hline
\end{array}
\]

\[
\text{ANORM} = \text{DLANSY}( '1', 'L', 9, A, 9, \text{WORK} )
\]

\( \mathbf{A} = \) (same as input matrix \( \mathbf{A} \) in Example 1)

Output:

\( \text{ANORM} = 45.0 \)

Example 2

This example computes the one norm of real symmetric matrix \( \mathbf{A} \) stored in upper storage mode.

Call Statements and Input:

\[
\begin{array}{cccccc}
\text{NORM} & \text{UPLO} & N & A & LDA & \text{WORK} \\
\hline
\end{array}
\]

\[
\text{ANORM} = \text{DLANSY}( '1', 'U', 9, A, 9, \text{WORK} )
\]

\( \mathbf{A} = \) (same as input matrix \( \mathbf{A} \) in Example 2)

Output:

\( \text{ANORM} = 45.0 \)

Example 3

This example computes the one norm of complex symmetric matrix \( \mathbf{A} \) stored in upper storage mode.

Call Statements and Input:

\[
\begin{array}{cccccc}
\text{NORM} & \text{UPLO} & N & A & LDA & \text{WORK} \\
\hline
\end{array}
\]

\[
\text{ANORM} = \text{ZLANSY}( '1', 'U', 3, A, 3, \text{WORK} )
\]

\[
\begin{bmatrix}
(9.0, 0.0) & (3.0, 3.0) & (3.0, -3.0) \\
6.0 & (18.0, 0.0) & (8.0, -6.0) \\
& & (43.0, 0.0)
\end{bmatrix}
\]

Output:

\( 582 \)
Example 4
This example computes the one norm of complex Hermitian matrix $A$ stored in lower storage mode.

Call Statements and Input:

\[
\begin{array}{c|c|c|c|c|c}
\text{NORM} & \text{UPLO} & \text{N} & \text{A} & \text{LDA} & \text{WORK} \\
\hline
\text{ANORM} & & & & & \\
\end{array}
\]
\[
\text{ANORM} = \text{ZLANHE}( \ '1', \ 'L', \ 3, \ A, \ 3, \ \text{WORK} )
\]

$A = $ (same as input matrix $A$ in Example 3)

Output:
ANORM = 89.39

Example 5
This example computes the one norm of complex Hermitian matrix $A$ stored in upper storage mode.

Call Statements and Input:

\[
\begin{array}{c|c|c|c|c|c}
\text{NORM} & \text{UPLO} & \text{N} & \text{A} & \text{LDA} & \text{WORK} \\
\hline
\text{ANORM} & & & & & \\
\end{array}
\]
\[
\text{ANORM} = \text{ZLANHE}( \ '1', \ 'U', \ 3, \ A, \ 3, \ \text{WORK} )
\]

$A = $ (same as input matrix $A$ in Example 4)

Output:
ANORM = 57.24

Example 6
This example computes the one norm of real symmetric matrix $A$ stored in lower-packed storage mode.

Call Statements and Input:

\[
\begin{array}{c|c|c|c|c|c}
\text{NORM} & \text{UPLO} & \text{N} & \text{AP} & \text{WORK} \\
\hline
\text{ANORM} & & & & & \\
\end{array}
\]
\[
\text{ANORM} = \text{DLANSP}( \ '1', \ 'L', \ 9, \ AP, \ \text{WORK} )
\]

$AP = $ (same as input matrix $AP$ in Example 5)

Output:
ANORM = 45.0

Example 7
This example computes the one norm of real symmetric matrix $A$ stored in upper-packed storage mode.

Call Statements and Input:

\[
\begin{array}{c|c|c|c|c|c}
\text{NORM} & \text{UPLO} & \text{N} & \text{AP} & \text{WORK} \\
\hline
\text{ANORM} & & & & & \\
\end{array}
\]
\[
\text{ANORM} = \text{DLANSP}( \ '1', \ 'U', \ 9, \ AP, \ \text{WORK} )
\]

$AP = $ (same as input matrix $AP$ in Example 6)
Example 8
This example computes the one norm of complex Hermitian matrix $A$ stored in lower-packed storage mode.

Call Statements and Input:

```
ANORM = ZLANHP( '1', 'L', 3, AP, WORK )
```

$AP =$ (same as input matrix AP in Example 7)

Output:
ANORM = 89.39

Example 9
This example computes the one norm of complex Hermitian matrix $A$ stored in upper-packed storage mode.

Call Statements and Input:

```
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>
<thead>
<tr>
<th>$A, B, work$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SSYSV$^\Delta$, SSPSV$^\Delta$</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DSYSV$^\Delta$, DSPSV$^\Delta$</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CSYSV$^\Delta$, CHESV$^\Delta$, CSPSV$^\Delta$, CHPSV$^\Delta$</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZSYSV$^\Delta$, ZHESV$^\Delta$, ZSPSV$^\Delta$, ZHPSV$^\Delta$</td>
</tr>
<tr>
<td>$^\Delta$LAPACK</td>
<td></td>
</tr>
</tbody>
</table>
### Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SSYSV</td>
</tr>
<tr>
<td></td>
<td>CALL SSPSV</td>
</tr>
<tr>
<td>C and C++</td>
<td>ssysv</td>
</tr>
<tr>
<td></td>
<td>sspsv</td>
</tr>
<tr>
<td>LAPACK</td>
<td>info = LAPACKE_ssysv</td>
</tr>
<tr>
<td></td>
<td>info = LAPACKE_sspsv</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. The value of the `matrix_layout` argument 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 147 on page 584.

**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 147 on page 584.

**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 147 on page 584.

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 147 on page 584.

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 $work_1$. 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 596.

Returned as: an $lda$ by (at least) $n$ array, containing numbers of the data type indicated in Table 147 on page 584.

ap
is the transformed matrix $A$ containing the results of the factorization. See “Function” on page 596.

Returned as: a one-dimensional array of (at least) length $n(n + 1)/2$, containing numbers of the data type indicated in Table 147 on page 584.

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 $D_{k+1,k+1}$ is a $2 \times 2$ diagonal block.

Returned as: a one-dimensional integer array of (at least) length $n$, containing integers.
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 147 on page 584.

**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, CSPSV, ZSPSV, CHPSV, and ZHPSV, returned as: an integer; \( info \geq 0 \).
- For LAPACKE_ssysv, LAPACKE_dsysv, LAPACKE_csysv, LAPACKE_zsysv, LAPACKE_chesv, LAPACKE_zhesv, LAPACKE_sspsv, LAPACKE_dspsv, LAPACKE_cspsv, LAPACKE_zspsv, LAPACKE_chpsv, and LAPACKE_zhpvs, 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. \( a, ap, b, ipiv, \) 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 83 or “Upper Storage Mode” on page 83, respectively.

   For a description of how complex Hermitian matrices are stored in lower or upper storage mode, see “Complex Hermitian Matrix” on page 84.

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 80 or “Upper-Packed Storage Mode” on page 82, respectively.

   For a description of how complex Hermitian matrices are stored in lower- or upper-packed storage mode, see “Complex Hermitian Matrix” on page 84.

6. 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.

7. 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 [“9” on page 1253] and [“19” on page 1254].

**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 65.

**Input-Argument Errors**

1. $uplo \neq 'U' \text{ or } 'L'$
2. $n < 0$
3. $nrhs < 0$
4. $lda \leq 0$
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 $\text{lwork} = 0$, the subroutine dynamically allocates $\text{WORK}$.

Call Statement and Input:

```
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:

```
| 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 |
```

$IPIV$ = ( -2 -2 3 4 -6 -6 7 8 )

```
| 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 |
```

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 $\text{lwork} = 0$, the subroutine dynamically allocates $\text{WORK}$.
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.
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:

```
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:

```
AP = (   3.0   5.0   1.0  -1.0   1.0   0.0   1.0  -1.0,
3.0  -1.0   1.0   0.0  -1.0  -1.0   0.0,
4.0  -1.0   0.0   1.0   1.0   1.0,
8.0  -1.0  -1.0  -1.0  -1.0  -1.0  -1.0,
   1.0   3.0  -1.0   1.0,
   1.0   1.0   0.0,
   2.0   1.0,
 16.0 )
```

IPIV = ( 1 2 -2 -2 5 6 7 8 )

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:

```
UPLO  N  NRHS  AP   IPIV   B  LDB  INFO
|    |    |    |     |    |   |     |
CALL ZSPSV( 'L' , 4 ,  3 , AP , IPIV , B , 4 , INFO )
```

Output:

```
B = |
  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   |
```

INFO = 0
AP = (same as input AP in Example 5)
B = (same as input B in Example 5)

Output:

\[
\begin{align*}
\text{AP} & = \left( (0.368, -0.319), (-0.062, 0.006), (0.625, 0.257), (-0.462, 0.314), \\
& \hspace{1cm} (0.258,-0.147), (1.085, -0.335), (-0.447, -1.386), \\
& \hspace{1cm} (0.333, 0.315), (-0.437, 1.248), \\
& \hspace{1cm} (0.841, 0.431) \right) \\
IPIV & = (1 2 4 4) \\
B & = \left[ \begin{array}{c}
(0.409,-0.663) (-0.582,-1.410) (2.484, 2.216) \\
(-1.664,-0.552) (-1.503,-4.837) (-3.577, 2.575) \\
(2.388, 4.010) (1.260,-0.430) (-1.273, 0.177) \\
(1.562, 0.164) (6.213, 1.471) (-0.980,-2.551)
\end{array} \right]
\end{align*}
\]

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:

\[
\begin{align*}
\text{UPLO} & , \ N , \ NRHS , \ AP , \ IPIV , \ B , \ LDB , \ INFO \\
\text{CALL ZHPSV( 'L', 4 , 3 , AP , IPIV , B , 4 , INFO )}
\end{align*}
\]

AP = (same as input AP in Example 6)
B = (same as input B in Example 6)

Output:

\[
\begin{align*}
\text{AP} & = \left( (-0.550, 0.000), (-0.027, 0.476), (0.062, 0.244), (-0.249, 0.022), \\
& \hspace{1cm} (-0.484, 0.000), (-0.082, -0.269), (0.152, -0.091), \\
& \hspace{1cm} (-0.496, 0.000), (0.245, -0.002), \\
& \hspace{1cm} (-0.479, 0.000) \right) \\
IPIV & = (1 2 3 4) \\
B & = \left[ \begin{array}{c}
(-1.623,-4.385) (2.635,-1.111) (-2.436,-0.306) \\
(-3.533,-0.212) (1.865,-2.830) (-0.866,-0.308) \\
(-1.742,-1.724) (-1.576, 1.575) (-0.478,-1.172) \\
(-1.537,-2.116) (-1.047, 1.668) (3.093, 0.365)
\end{array} \right]
\end{align*}
\]

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>
<thead>
<tr>
<th>$A$, $B$, work</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SSYTRF, SSPTRF</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DSYTRF, DSPTRF</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CSYTRF, CHETRF, CSPTRF, CHPTRF</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZSYTRF, ZHETRF, ZSPTRF, ZHPTRF</td>
</tr>
</tbody>
</table>

**Note:**
- The output from the SSYTRF, DSYTRF, CSYTRF, ZSYTRF, 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. The value of the $matrix\_layout$ argument must be $LAPACK\_ROW\_MAJOR$ or $LAPACK\_COL\_MAJOR$. 

---

**Table 148: Data Types.**

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Short-precision real</th>
<th>Long-precision real</th>
<th>Short-precision complex</th>
<th>Long-precision complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSYTRF, SSPTRF</td>
<td>SSYTRF</td>
<td>DSYTRF, DSPTRF</td>
<td>CSYTRF, CHETRF, CSPTRF, CHPTRF</td>
<td>ZSYTRF, ZHETRF, ZSPTRF, ZHPTRF</td>
</tr>
</tbody>
</table>

**Syntax**

**Language** | **Syntax**
|----------------|--------------------------------------------------|
| Fortran        | CALL SSYTRF | DSYTRF | CSYTRF | ZSYTRF | CHETRF | ZHETRF (uplo, n, a, lda, ipiv, work, lwork, info)  
|               | CALL SSPTRF | DSPTRF | CSPTRF | ZSPTRF | CHPTRF | ZHPTRF (uplo, n, ap, ipiv, info) |
| C and C++      | ssysyr | dsytrf | cssyr | zsyr | chetrf | zhesyr (uplo, n, a, lda, ipiv, work, lwork, info);  
|               | sspsyr | dsptrf | cspyr | zspyr | chptrf | zhptrf (uplo, n, ap, ipiv, info); |
| LAPACK         | info = LAPACKE_ssysyr | LAPACKE_dsysyr | LAPACKE_cssyr | LAPACKE_zsyr | LAPACKE_chetrf | LAPACKE_zhetrf (matrix\_layout, uplo, n, a, lda, ipiv);  
|               | info = LAPACKE_sspyr | LAPACKE_dsptrf | LAPACKE_cspyr | LAPACKE_zspyr | LAPACKE_chptrf | LAPACKE_zhptrf (matrix\_layout, uplo, n, ap, ipiv); |
**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; $n \geq 0$.

**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 148 on page 593.

**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 148 on page 593.

**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.

**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 148 on page 593.

**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 $work_{\text{1}}$. 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 596.

Returned as: an \( lda \) by (at least) \( n \) array, containing numbers of the data type indicated in Table 148 on page 593.

\( ap \)

is the transformed matrix \( A \) containing the results of the factorization. See “Function” on page 596.

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 593.

\( ipiv \)

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 \( D_{k-1,k-1,k}; k \) 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 \( D_{k,k+1,k+1,k}; k+1 \) 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, CSPTRF, ZSPTRF, CHPTRF, and ZHPTRF, returned as: an integer; \( info \geq 0 \).
- For LAPACKE_ssytrf, LAPACKE_dsytrf, LAPACKE_csytrf, LAPACKE_zsytrf, LAPACKE_chetrf, LAPACKE_zhetrf, 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 83 or “Upper Storage Mode” on page 83, respectively.

For a description of how complex Hermitian matrices are stored in lower or upper storage mode, see “Complex Hermitian Matrix” on page 84.
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 80 or “Upper-Packed Storage Mode” on page 82, respectively.

For a description of how complex Hermitian matrices are stored in lower- or upper-packed storage mode, see “Complex Hermitian Matrix” on page 84.

6. 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.

7. For best performance, specify $lwork = 0$.

**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 [“9” on page 1253] and [“19” on page 1254].

**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 65.

**Input-Argument Errors**

1. uplo ≠ 'U' or 'L'
2. $n < 0$
3. $lda ≤ 0$
4. $n > lda$
5. $lwork ≠ 0$ and $lwork ≠ -1$ 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:**

```
UPLO  N   A  LDA  IPIV   WORK LWORK INFO
CALL DSYTRF( 'L' , 8 , A , 8 , IPIV , WORK,  0 , INFO )
```

```
A   =  |
     |  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    .     .     .  |
     |-10.0  -3.0   2.0 -10.0   6.0 16.0    .     .  |
     | -2.0   2.0  -8.0  -8.0  8.0  8.0  6.0    .  |
     |-14.0  -5.0  6.0 -14.0  6.0 20.0 18.0 34.0 |
```

**Output:**

```
A   =  |
     |  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 1 |
```

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:

$$
\begin{pmatrix}
(0.368, -0.319) & -0.021, 0.022) & (0.312, -0.105) & (-0.070, 0.263) \\
(-0.021, 0.022) & (0.259, -0.148) & (0.212, -0.237) & (0.071, 0.370) \\
(0.312, -0.105) & (0.212, -0.237) & (0.273, -0.041) & (0.384, -0.056) \\
(-0.070, 0.263) & (0.071, 0.370) & (0.384, -0.056) & (-0.230, 0.085)
\end{pmatrix}
$$

**Note:** Because $lwork = 0$, the subroutine dynamically allocates WORK.

Call Statement and Input:

```
UPLO N A LDA IPIV WORK LWORK INFO
|    |   |   |    |       |    |     |
CALL ZSYTRF( 'L' , 4 , A , 4 , IPIV , WORK, 0 , INFO )
```

Output:

```
A = | (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) |
```

IPIV = (1 2 4 4)

INFO = 0

**Example 3**

This example shows a factorization of the indefinite complex Hermitian matrix $A$ of order 4.

Matrix $A$ is:

```
| (-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) |
```

**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:

```
UPLO  N   A  LDA  IPIV   WORK LWORK INFO
|    |    |    |    |    |    |
CALL ZHETRF( 'L' , 4 , A , 4 , IPIV , WORK , 0 , INFO )
```

```
A = |
  | (-0.550, .   ) | (-0.609, .   ) | .   | .   | .   |
  | ( 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, .   ) |
```

Output:

```
A = |
  | (-0.550, 0.000) | .   | .   | .   | .   |
  | ( 0.027, 0.476) | (-0.483, 0.000) | .   | .   | .   |
  | (-0.027, 0.476) | (-0.483, 0.000) | .   | .   | .   |
  | ( 0.137, -0.012) | (-0.083, -0.021) | (-0.126, -0.053) | (-0.479, 0.000) |
```

```
IPIV = ( 1 2 4 4 )
INFO  =   0
```

**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:

```
UPLO  N   AP   IPIV   INFO
|    |    |    |    |
CALL DSPTRF( 'L' , 8 , AP , IPIV , INFO )
```

```
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:

```
AP = ( 3.0  5.0 -1.0  1.0  0.0  1.0 -1.0,
   3.0 -1.0  1.0  0.0 -1.0 -1.0  0.0,
   4.0 -1.0  0.0  1.0  1.0 -1.0  1.0,
   8.0 -1.0 -1.0 -1.0 -1.0,
   1.0  3.0 -1.0  1.0,
   1.0  1.0  0.0,
```

Linear algebraic equations subroutines 599
IPIV = ( -2 -2 3 4 -6 -6 7 8 )
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:

```
CALL ZSPTRF( 'L' , 4 , AP , IPIV , INFO )
```

AP = ( ( 0.368, -0.319), (-0.021,  0.022), (0.312, -0.105),(-0.070,  0.263),
      ( 0.259, -0.148), (0.212, -0.237),( 0.071,  0.370),
      (0.273, -0.041),( 0.384, -0.056),
      (-0.230,  0.085) )

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

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:

```
CALL ZHPTRF( 'L' , 4 , AP , IPIV , INFO )
```

AP = ( (-0.550,   .   ), ( 0.015,-0.262), (-0.034,-0.134), ( 0.137,-0.012),
      (-0.609,  .   ), (-0.062, 0.150), (-0.083,-0.021),
      (-0.560,  .   ), (-0.126,-0.053),
      (-0.558,  .   ) )

Output:

AP = ( (-0.550, 0.000), (-0.027, 0.476), ( 0.062, 0.244), (-0.249, 0.022),
      (-0.484, 0.800),(-0.092, -0.269), ( 0.152, -0.091),
IPIV = ( 1 2 3 4 )
INFO  =  0

### SSYTRS, DSYTRS, CSYTRS, ZSYTRS, CHETRS, ZHERTS, 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, ZHERTS use the results of the factorization of matrix \( A \), produced by a preceding call to SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, or ZHERTF, 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 149: Data Types.

<table>
<thead>
<tr>
<th>( A, B )</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SSYTRS( ^A ), SSPTRS( ^A )</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DSYTRS( ^A ), DSPTRS( ^A )</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CSYTRS( ^A ), CHETRS( ^A ), CSPTRS( ^A ), CHPTRS( ^A )</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZSYTRS( ^A ), ZHERTS( ^A ), ZSPTRS( ^A ), ZHPTRS( ^A )</td>
</tr>
</tbody>
</table>

\( ^A \) LAPACK

#### Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SSYTRS</td>
</tr>
<tr>
<td></td>
<td>CALL SSPTRS</td>
</tr>
<tr>
<td>C and C++</td>
<td>ssysys</td>
</tr>
<tr>
<td></td>
<td>ssptsys</td>
</tr>
<tr>
<td>LAPACK</td>
<td>info = LAPACKE_ssysys</td>
</tr>
<tr>
<td></td>
<td>info = LAPACKE_sspptsys</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. The value of the matrix_layout argument 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 149 on page 601.

`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 149 on page 601.

`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 149 on page 601.

`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 149 on page 601.

\( \text{info} \) has the following meaning:

If \( \text{info}=0 \), the factorization completed successfully.

Returned as:

- For SSYTRS, DSYTRS, CSYTRS, ZSYTRS, CHETRS, ZHETRS, SSPTRS, DSPTRS, CSPTRS, ZSPTRS, CHPTRS, and ZHPTRS, returned as: an integer; \( \text{info} \geq 0 \).
- 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; \( \text{info} \geq 0 \).

Notes

1. These subroutines accept lowercase letters for the \( \text{uplo} \) argument.
2. In your C program, argument \( \text{info} \) must be passed by reference.
3. \( a, ap, b \) and \( \text{ipiv} \) 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 83 or “Upper Storage Mode” on page 83, respectively.
   For a description of how complex Hermitian matrices are stored in lower or upper storage mode, see “Complex Hermitian Matrix” on page 84.
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 80 or “Upper-Packed Storage Mode” on page 82, respectively.
   For a description of how complex Hermitian matrices are stored in lower- or upper-packed storage mode, see “Complex Hermitian Matrix” on page 84.

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 [“9” on page 1253] and [“19” on page 1254].

Error conditions

Computational Errors

None

\textbf{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:

```
UPLO N NRHS A LDA IPIV B LDB INFO
CALL DSYTRS( 'L' , 8 , 3 , A , 8 , IPIV , B , 8 , INFO )
```

$A =$ (same as output $A$ in Example 1)

$IPIV =$ (same as output $IPIV$ in Example 1)

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>-38.0</td>
<td>47.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.0</td>
<td>-18.0</td>
<td>73.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.0</td>
<td>52.0</td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

$B =$

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-30.0</td>
<td>-228.0</td>
<td>-42.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32.0</td>
<td>183.0</td>
<td>105.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34.0</td>
<td>297.0</td>
<td>9.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32.0</td>
<td>244.0</td>
<td>44.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>62.0</td>
<td>497.0</td>
<td>61.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Output:

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.0</td>
<td>8.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>2.0</td>
<td>7.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>3.0</td>
<td>6.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

$B =$

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>4.0</td>
<td>5.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>5.0</td>
<td>4.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>6.0</td>
<td>3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>7.0</td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>8.0</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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:

```
UPLO N NRHS A LDA IPIV B LDB INFO
CALL ZSYTRS( 'L' , 4 , 3 , A , 4 , IPIV , B , 4 , INFO )
```

INFO = 0
Example 3

This example shows how to solve the system $\mathbf{A}\mathbf{x} = \mathbf{b}$ for three right-hand sides, where indefinite complex Hermitian matrix $\mathbf{A}$ is the same matrix factored in the Example 3 for ZHETRF.

Call Statement and Input:

```
UPLO  N  NRHS  A  LDA  IPIV  B  LDB  INFO
```

```call
CALL ZHETRS('L', 4, 3, A, 4, IPIV, B, 4, INFO)
```

A = (same as output A in Example 3)

IPIV = (same as output IPIV in Example 3)

Output:

```
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 4

This example shows how to solve the system $\mathbf{A}\mathbf{x} = \mathbf{b}$ for three right-hand sides, where indefinite real symmetric matrix $\mathbf{A}$ is the same matrix factored in the Example 1 for DSYTRF.
Call Statement and Input:

```
|    |   |    |     |    |   |     |
|------------------------|
CALL DSPTRS( 'L' , 8 , 3 , AP , IPIV , B , 8 , INFO )
```

AP = (same as output AP in Example 4)

IPIV = (same as output IPIV in Example 4)

```
|   1.0  -38.0  47.0 |
|   7.0  -10.0  73.0 |
|   6.0   52.0   2.0 |
```

B =

```
|   1.0  -38.0  47.0 |
|   7.0  -10.0  73.0 |
|   6.0   52.0   2.0 |
```

Output:

```
|   1.0   1.0   8.0 |
|   1.0   2.0   7.0 |
|   1.0   3.0   6.0 |
```

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 , 4 , INFO )
```

AP = (same as output AP in Example 5)

IPIV = (same as output IPIV in Example 5)

```
| ( 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) |
```

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:

```
| ( 0.409,-0.663) (-0.582,-1.410) ( 2.484, 2.216) |
| (-1.664,-0.552) (-1.503,-4.837) (-3.577, 2.575) |
```
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:

```plaintext
CALL ZHPTRS( 'L' , 4 , 3 , AP , IPIV , B , 4 , INFO )
```

$AP = \text{(same as output AP in Example 6)}$

$IPIV = \text{(same as output IPIV in Example 6)}$

Output:

```plaintext
B = ((-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) )
```

$INFO = 0$

**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:

$A = UDU^T$ 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.
Table 150: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>ipvt</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, B</td>
<td>Integer</td>
<td>DBSSV</td>
</tr>
<tr>
<td>Long-precision real</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL DBSSV (uplo, n, nrhs, ap, ipvt, bx, ldb, nsinfo)</td>
</tr>
<tr>
<td>C and C++</td>
<td>dbssv (uplo, n, nrhs, ap, ipvt, bx, ldb, nsinfo);</td>
</tr>
</tbody>
</table>

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 ≥ 0.

nrhs
is the number of right-hand sides; i.e., the number of columns of matrix B.
Specified as: an integer; nrhs ≥ 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 150 on page 608. See “Notes” on page 610.

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 ldb by (at least) nrhs array, containing numbers of the data type indicated in Table 150 on page 608.

ldb
is the leading dimension of the array specified for B.
Specified as: an integer; ldb > 0 and ldb ≥ n.

nsinfo
is the number of elements in array, AP.
If n ≤ nco, nsinfo = n(n + 1) / 2
Where:
\[ nco = \sqrt{2.0(\text{ncs})} \]
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:

\[
\begin{align*}
    n0 &= 100 \\
    ns &= (n + n0) (n + n0 + 1) / 2 + n(n0) \\
    \text{For } uplo = 'L', \\
    nsinfo &\geq ns \\
    \text{For } uplo = 'U', \\
    n1 &= (n + 1) / 2 \\
    nt &= n((n + 1) / 2) \\
    nt1 &= n1(n1 + 1) \\
    ns1 &= nt + nt1 \\
    nsinfo &\geq \max(ns, ns1)
\end{align*}
\]

To determine the minimal amount of storage see “Notes” on page 610.

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 610 and “Function” on page 610.

  Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 150 on page 608.

- **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 150 on page 608.

- **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 150 on page 608.

- **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, \(AP\) 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 64.
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 vectors and matrices used in this computation must have no common elements; otherwise, results are unpredictable. See “Vector concepts” on page 71.
5. On return, if nsinfo ≥ 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:

   ```
   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 80.

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:

$$A = UDU^T$$ 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.

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 [“9” on page 1253] and [“78” on page 1257].

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
Input-Argument Errors

1. uplo ≠ 'U' or 'L'
2. n < 0
3. n > ldb
4. ldb ≤ 0
5. nrhs < 0
6. 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 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:

\[
A = \begin{bmatrix}
3.0 & 5.0 & -2.0 & 2.0 & 3.0 & -5.0 & -2.0 & -3.0 \\
5.0 & 3.0 & 2.0 & -2.0 & 5.0 & -3.0 & 2.0 & -5.0 \\
-2.0 & 2.0 & 0.0 & 0.0 & -2.0 & 2.0 & 0.0 & 6.0 \\
2.0 & -2.0 & 0.0 & 0.0 & -6.0 & -10.0 & -8.0 & -14.0 \\
3.0 & 5.0 & -2.0 & -6.0 & 12.0 & 6.0 & 8.0 & 6.0 \\
-5.0 & -3.0 & 2.0 & -10.0 & 6.0 & 16.0 & 8.0 & 20.0 \\
-2.0 & 2.0 & 0.0 & -5.0 & 8.0 & 8.0 & 6.0 & 18.0 \\
-3.0 & -5.0 & 6.0 & -14.0 & 6.0 & 20.0 & 18.0 & 34.0
\end{bmatrix}
\]

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:

```
UPLO N NRHS AP IPVT BX LDB NSINFO
|    |    |     |   |     |   |      |
CALL DBSSV ( 'L', 8, 3, AP, IPVT, BX, 8, 36 )
```

\[
AP = ( 3.0, 5.0, -2.0, 2.0, 3.0, -5.0, -2.0, -3.0, \\
3.0, 2.0, -2.0, 5.0, -3.0, 2.0, -5.0, \\
0.0, 0.0, -2.0, 2.0, 0.0, 6.0, \\
8.0, -6.0, -10.0, -8.0, -14.0, \\
12.0, 6.0, 8.0, 6.0, \\
16.0, 8.0, 20.0, \\
6.0, 6.0, 18.0, 34.0 )
\]

\[
BX = \begin{bmatrix}
1.0 & -38.0 & 47.0 \\
7.0 & -18.0 & 73.0 \\
6.0 & 52.0 & 2.0 \\
-30.0 & -228.0 & -42.0 \\
32.0 & 183.0 & 105.0 \\
34.0 & 297.0 & 9.0 \\
32.0 & 244.0 & 44.0 \\
62.0 & 497.0 & 61.0
\end{bmatrix}
\]
Output:

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

\[
\text{BX} = \begin{array}{ccc}
  1.0 & 1.0 & 8.0 \\
  1.0 & 2.0 & 7.0 \\
  1.0 & 3.0 & 6.0 \\
  1.0 & 4.0 & 5.0 \\
  1.0 & 5.0 & 4.0 \\
  1.0 & 6.0 & 3.0 \\
\end{array}
\]

\[
\text{NSINFO} = 0
\]

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:

\[
\begin{align*}
  & 34.0 & 18.0 & 17.0 & 6.0 & -14.0 & 6.0 & -5.0 & -3.0 \\
  & 18.0 & 6.0 & 6.0 & 8.0 & -8.0 & 0.0 & 2.0 & -2.0 \\
  & 17.0 & 6.0 & 9.0 & 9.0 & -8.0 & 0.0 & 2.0 & -2.0 \\
  & 6.0 & 8.0 & 9.0 & 12.0 & -6.0 & -2.0 & 5.0 & 3.0 \\
  & -14.0 & -8.0 & -8.0 & -6.0 & 8.0 & 0.0 & -2.0 & 2.0 \\
  & 6.0 & 0.0 & 0.0 & -2.0 & 0.0 & 0.0 & 2.0 & -2.0 \\
  & -5.0 & 2.0 & 2.0 & 5.0 & -2.0 & 2.0 & 3.0 & 5.0 \\
  & -3.0 & -2.0 & -2.0 & 3.0 & 2.0 & -2.0 & 5.0 & 3.0 \\
\end{align*}
\]

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:

\[
\begin{array}{|c|c|c|c|c|c|c|c|}
\hline
\text{UPLO} & \text{N} & \text{NRHS} & \text{AP} & \text{IPVT} & \text{BX} & \text{LDB} & \text{NSINFO} \\
\hline
\text{CALL} & \text{DBSSV} & ( & 'U', & 8, & 3, & \text{AP}, & \text{IPVT}, & \text{BX}, & 8, & 36 & ) \\
\hline
\end{array}
\]

\[
\begin{align*}
\text{AP} = & ( & 34.0, & 18.0, & 6.0, \\
& 17.0, & 6.0, & 9.0, & 9.0, & 12.0, \\
& -14.0, & -8.0, & -8.0, & -6.0, & 8.0, & 0.0, & 2.0, & -2.0, & 5.0, & 3.0, \\
& -5.0, & -5.0, & -2.0, & -2.0, & 3.0, & 2.0, & -2.0, & 5.0, & 3.0) \\
\end{align*}
\]

\[
\begin{align*}
\text{BX} = & ( & 59.0 & 52.0 & 479.0 \\
& 30.0 & 38.0 & 232.0 \\
& 33.0 & 50.0 & 247.0 \\
& 35.0 & 114.0 & 201.0 \\
& -28.0 & -36.0 & 216.0 \\
& 4.0 & -4.0 & 40.0 \\
& 12.0 & 88.0 & 20.0 \\
& 4.0 & 56.0 & -20.0 \\
\end{align*}
\]

Output:

\[
\begin{align*}
\begin{array}{ccc}
  1.0 & 1.0 & 8.0 \\
  1.0 & 2.0 & 7.0 \\
  1.0 & 3.0 & 6.0 \\
  1.0 & 4.0 & 5.0 \\
  1.0 & 5.0 & 4.0 \\
  1.0 & 6.0 & 3.0 \\
\end{array}
\end{align*}
\]
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 151: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>ipvt</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Integer</td>
</tr>
</tbody>
</table>

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL DBSTRF (uplo, n, ap, ipvt, nsinfo)</td>
</tr>
<tr>
<td>C and C++</td>
<td>dbstrf (uplo, n, ap, ipvt, nsinfo);</td>
</tr>
</tbody>
</table>

**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 \( nsinfo \), containing numbers of the data type indicated in Table 151 on page 613. See “Notes” on page 615.

**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 \( uplo = 'U' \),
\[
 n1 = (n + 1) / 2 \\
 nt = n((n + 1) / 2) \\
 nt1 = n1(n1 + 1) \\
 ns1 = nt + nt1 \\
 nsinfo \geq \max(ns, ns1)
\]

To determine the minimal amount of storage see “Notes” on page 615.

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 615 and “Function” on page 615.

Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 151 on page 613.
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 151 on page 613.

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 64.

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:

   ```
   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 80.

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 [“9” on page 1253] and [“78” on page 1257].

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 65.

Input-Argument Errors

1. $uplo \neq 'U' \text{ or } 'L'$
2. $n < 0$
3. $nsinfo < (\text{minimum value})$.

- 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:

```
UPLO  N   AP  IPVT  NSINFO
|    |    |   |       |
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, 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:

\[
\begin{pmatrix}
34.0 & 18.0 & 17.0 & 6.0 & -14.0 & 6.0 & -5.0 & -3.0 \\
18.0 & 6.0 & 6.0 & 8.0 & -8.0 & 0.0 & 2.0 & -2.0 \\
17.0 & 6.0 & 9.0 & 9.0 & -8.0 & 0.0 & 2.0 & -2.0 \\
6.0 & 8.0 & 9.0 & 12.0 & -6.0 & -2.0 & 5.0 & 3.0 \\
-14.0 & -8.0 & -8.0 & -6.0 & 8.0 & 0.0 & -2.0 & 2.0 \\
6.0 & 0.0 & 0.0 & -2.0 & 0.0 & 0.0 & 2.0 & -2.0 \\
-5.0 & 2.0 & 2.0 & 5.0 & -2.0 & 2.0 & 3.0 & 5.0 \\
-3.0 & -2.0 & -2.0 & 3.0 & 2.0 & -2.0 & 5.0 & 3.0 \\
\end{pmatrix}
\]

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 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 152: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$, $B$</td>
<td>ipvt</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Integer</td>
</tr>
</tbody>
</table>

Note: The input to this solve subroutine must be the output from the factorization subroutine DBSTRF.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL DBSTRS (uplo, n, nrhs, ap, ipvt, bx, ldb, info)</td>
</tr>
<tr>
<td>C and C++</td>
<td>dbstrs (uplo, n, nrhs, ap, ipvt, bx, ldb, info);</td>
</tr>
</tbody>
</table>
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 ≥ 0`.

`nrhs` is the number of right-hand sides; i.e., the number of columns of matrix `B`.

Specified as: an integer; `nrhs ≥ 0`.

`ap` is the factored matrix `A` produced by a preceding call to DBSTRF.

Specified as: a one-dimensional array of length `n*info`, containing numbers of the data type indicated in Table 152 on page 617. See “Notes” on page 619 and “DBSTRF (Symmetric Indefinite Matrix Factorization)” on page 613.

`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 152 on page 617. See “Notes” on page 619.

`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 152 on page 617.

`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 152 on page 617.

`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 “Vector concepts” on page 71.
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 80.
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 [“9” on page 1253] and [“78” on page 1257].

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 )
```
**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:**

```fortran
CALL DBSTRS ( 'U', 8, 3, AP, IPVT, BX, 8, INFO )
```

**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.)

**Output:**

\[
\begin{bmatrix}
1.0 & 59.0 \\
52.0 & 30.0 \\
479.0 & 33.0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
1.0 & 1.0 \\
2.0 & 3.0 \\
4.0 & 6.0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
1.0 & 1.0 \\
2.0 & 3.0 \\
4.0 & 6.0 \\
\end{bmatrix}
\]

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

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

\[
\begin{bmatrix}
1.0 & 5.0 \\
6.0 & 7.0 \\
8.0 & 9.0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
1.0 & 6.0 \\
7.0 & 8.0 \\
9.0 & 10.0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
1.0 & 7.0 \\
2.0 & 3.0 \\
3.0 & 4.0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
1.0 & 8.0 \\
2.0 & 3.0 \\
3.0 & 4.0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
1.0 & 9.0 \\
2.0 & 10.0 \\
3.0 & 11.0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
1.0 & 10.0 \\
2.0 & 11.0 \\
3.0 & 12.0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
1.0 & 11.0 \\
2.0 & 12.0 \\
3.0 & 13.0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
1.0 & 12.0 \\
2.0 & 13.0 \\
3.0 & 14.0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
1.0 & 13.0 \\
2.0 & 14.0 \\
3.0 & 15.0 \\
\end{bmatrix}
\]
INFO = 0

STRTRI, DTRTRI, CTRTRI, ZTRTRI, STPTRI, DTPTRI, CTPTRI, and ZTPTRI (Triangular Matrix Inverse)

Purpose
These subroutines find the inverse of triangular matrix A:

\[ A \leftarrow A^{-1} \]

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 153: Data Types.

<table>
<thead>
<tr>
<th>A</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>STRTRI (^<em>) and STPTRI (^</em>)</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DTRTRI (^<em>) and DTPTRI (^</em>)</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CTRTRI (^<em>) and CTPTRI (^</em>)</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZTRTRI (^<em>) and ZTPTRI (^</em>)</td>
</tr>
</tbody>
</table>

\(^*\) LAPACK

Syntax

Table 154: Syntax for supported computing languages.

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
</table>
| Fortran  | CALL STRTRI | DTRTRI | CTRTRI | ZTRTRI (uplo, diag, n, a, lda, info)  
 | CALL STPTRI | DTPTRI | CTPTRI | ZTPTRI (uplo, diag, n, ap, info) |
| C and C++ | strtri | dtrtri | ctrtri | ztrtri (uplo, diag, n, a, lda, info);  
 | stptri | dtptri | ctptri | ztptri (uplo, diag, n, ap, info); |
| LAPACKE | info = LAPACKE_strtri | LAPACKE_dtrtri | LAPACKE_ctrtri | LAPACKE_ztrtri (matrix_layout, uplo, diag, n, a, lda);  
 | info = LAPACKE_stptri | LAPACKE_dtptri | LAPACKE_ctptri | LAPACKE_ztptri (matrix_layout, 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. The value of the matrix_layout argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

\textit{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'.

\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'.

\textit{a} \\
is the upper or lower triangular matrix A of order \(n\), stored in upper- or lower-triangular storage mode, respectively. Specified as: an \textit{lda} by (at least) \(n\) array, containing numbers of the data type indicated in Table 153 on page 621.

\textit{lda} \\
is the leading dimension of the arrays specified for \(a\). 
Specified as: an integer; \(lda > 0\) and \(lda \geq n\).

\textit{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 153 on page 621.

\textit{n} \\
is the order of matrix A. 
Specified as: an integer; \(n \geq 0\).

\textit{info} \\
See On Return.

\textbf{On Return} \\
\textit{a} \\
is the inverse of the upper or lower triangular matrix A of order \(n\), stored in upper- or lower-triangular storage mode, respectively. Returned as: an \textit{lda} by (at least) \(n\) array, containing numbers of the data type indicated in Table 153 on page 621.

\textit{ap} \\
is the inverse of the upper or lower triangular matrix A of order \(n\), stored in upper- or lower-triangular-packed storage mode, respectively. Returned as: a one-dimensional array of (at least) length \(n(n+1)/2\), containing numbers of the data type indicated in Table 153 on page 621.

\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 ≥ 0.

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 88.

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 [“9” on page 1253] and [“46” on page 1255].

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 ≠ 'U' or 'L'
2. diag ≠ 'U' or 'N'
3. n < 0
4. lda ≤ 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:**

```
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:**

\[
A = \begin{bmatrix}
1.00 & -1.50 & 2.00 & 3.75 & 35.00 \\
. & 0.50 & -1.00 & -1.75 & -14.00 \\
. & . & 0.25 & 1.00 & 7.00 \\
. & . & . & -0.50 & -3.00 \\
. & . & . & . & -1.00
\end{bmatrix}
\]

\( \text{INFO} = 0 \)

**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:

\[
\begin{bmatrix}
1.0 & 0.0 & 0.0 & 0.0 & 0.0
\end{bmatrix}
\]
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 )
```

Output:

```
A = [\[
    \begin{pmatrix}
    3.0 & . & . & . & . \\
    4.0 & 8.0 & . & . & . \\
    5.0 & 9.0 & 8.0 & . & . \\
    6.0 & 1.0 & 4.0 & 6.0 & . \\
    . & . & . & . & .
    \end{pmatrix}
\]
```

```
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:

```
| (-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) |
```

Matrix $A^{-1}$ is:

```
| (-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) |
```
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:

$$
\begin{bmatrix}
1.00 & 0.00 & 0.00 & 0.00 & 0.00 \\
4.00 & -3.00 & 1.00 & 0.00 & 0.00 \\
-1.00 & 3.00 & -3.00 & 1.00 & 0.00 \\
0.00 & 0.00 & -2.00 & -1.00 & 1.00 \\
-1.00 & 0.00 & 4.00 & 3.00 & -5.00 \\
\end{bmatrix}
$$

Matrix $A^{-1}$ is:

$$
\begin{bmatrix}
1.00 & 0.00 & 0.00 & 0.00 & 0.00 \\
-4.00 & 3.00 & 1.00 & 0.00 & 0.00 \\
-14.00 & 2.00 & 3.00 & 1.00 & 0.00 \\
-59.00 & -34.00 & 8.00 & 13.00 & 3.00 \\
64.00 & -8.00 & -10.00 & -9.00 & -1.00 \\
\end{bmatrix}
$$

Note: Because matrix $A$ is unit triangular, the diagonal elements are not referenced. ESSL assumes a value of one for the diagonal elements.
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:

```
CALL STPTRI( 'U' , 'N' , 5 , AP, INFO )
```

Output:

```
AP   =  (1.00, -1.50, 0.50, 2.00, -1.00, 0.25, 3.75, -1.75, 1.00, -0.50, 35.00, -14.00, 7.00, -3.00, -1.00)
```

Example 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:

```
CALL STPTRI( 'L' , 'U' , N , AP, INFO )
```

Output:

```
AP   =  (. , -3.0, 20.0, -138.0, 745.0, . , -8.0, 55.0, -299.0, . , -8.0, 44.0, . , -6.0, . )
```

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.

Call Statement and Input:

```
UPLO  DIAG  N   AP  INFO
|     |    |   |    |
CALL ZTPTRI( 'U', 'N', 5, AP, INFO )
```

```
AP   =  ( (-4.00,  1.00),
( 4.00, -3.00), (-2.00,  0.00),
(-1.00,  3.00), (-3.00, -1.00), (-5.00,  3.00),
( 0.00,  0.00), (-2.00, -1.00), (-3.00, -3.00), (4.00, -4.00),
(-1.00,  0.00), ( 4.00,  3.00), (-5.00, -5.00), (2.00,  0.00), (2.00, -1.00) )
```

Output:

```
AP   =  ( (-0.24, -0.06),
(-0.56,  0.24), (-0.50,  0.00),
( 0.41,  0.09), ( 0.18,  0.21), (-0.15, -0.09),
(-0.22,  0.13), (-0.22, -0.06), ( 0.07, -0.11), ( 0.12,  0.12),
( 1.32,  2.12), ( 0.21,  1.87), ( 0.02, -0.47), (-0.05, -0.15), (0.40, 0.20) )
```

```
INFO =    0
```

### 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.

Call Statement and Input:

```
UPLO  DIAG  N   AP  INFO
|     |    |   |    |
CALL ZTPTRI( 'L', 'U', 5, AP, INFO )
```

```
AP   =  ( ., ( 4.00, -3.00), (-1.00,  3.00), (0.00, 0.00), (-1.00, 0.00),
., (-3.00, -1.00), (-2.00, -1.00), (4.00, 3.00),
., (-3.00, -3.00), (-5.00, -5.00),
., ( 2.00,  0.00),
.
```

Output:

```
AP   =  ( ., (-4.00, 3.00), (-14.00,  2.00), (-59.00, -34.00), (64.00, 8.00),
., ( 3.00,  1.00), ( 8.00, 13.00), (-18.00, -9.00),
., ( 3.00,  3.00), (-1.00, -1.00),
., (-2.00,  0.00),
.
```

```
INFO =    0
```

**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>
<thead>
<tr>
<th>Table 155: 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>
</tbody>
</table>

$^\Delta$LAPACK

Syntax

<table>
<thead>
<tr>
<th>Table 156: Syntax for supported computing languages.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
</tr>
<tr>
<td>Fortran</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>C and C++</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>LAPACK</td>
</tr>
<tr>
<td></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. The value of the matrix_layout argument 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 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 $\mathbf{A}$, where:

**For SLANTR, DLANTR, CLANTR, and ZLANTR**
- If $\text{uplo}=\text{'U'}$, $\mathbf{A}$ is stored in upper-trapezoidal storage mode.
- If $\text{uplo}=\text{'L'}$, $\mathbf{A}$ is stored in lower-trapezoidal storage mode.

**For SLANTP, DLANTP, CLANTP, and ZLANTP**
- If $\text{uplo}=\text{'U'}$, $\mathbf{A}$ is stored in upper-triangular-packed storage mode.
- If $\text{uplo}=\text{'L'}$, $\mathbf{A}$ 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 $\mathbf{A}$, where:

**For SLANTR, DLANTR, CLANTR, and ZLANTR**
- If $\text{diag} = \text{'U'}$, $\mathbf{A}$ is a unit trapezoidal matrix.
- If $\text{diag} = \text{'N'}$, $\mathbf{A}$ is not a unit trapezoidal matrix.

**For SLANTP, DLANTP, CLANTP, and ZLANTP**
- If $\text{diag} = \text{'U'}$, $\mathbf{A}$ is a unit triangular matrix.
- If $\text{diag} = \text{'N'}$, $\mathbf{A}$ 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 $\mathbf{A}$.
Specified as: an integer; $m \geq 0$.

$n$

**For SLANTR, DLANTR, CLANTR, and ZLANTR**
$n$ is the number of columns in matrix $\mathbf{A}$.

**For SLANTP, DLANTP, CLANTP, and ZLANTP**
$n$ is the order of matrix $\mathbf{A}$
Specified as: an integer; $n \geq 0$.

$\text{ap}$ is the matrix $\mathbf{A}$ 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 Table 155 on page 629.

$\mathbf{a}$ is the trapezoidal matrix $\mathbf{A}$, stored in upper- or lower-trapezoidal storage mode.
Specified as: an $\text{lda}$ by (at least) $n$ array, containing numbers of the data type indicated in Table 155 on page 629.

$\text{lda}$ is the leading dimension of matrix $\mathbf{A}$.
Specified as: an integer; $\text{lda} \geq m$.

$\text{work}$ is the work area used by this subroutine, where:
- When $\text{norm} = \text{'I'}$, '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, work is not referenced.

Specified as: an area of storage containing numbers of data type indicated in Table 155 on page 629.

On Return

Function value

is the result of the norm computation, returned as a number of the data type indicated in Table 155 on page 629.

If norm = 'O' or '1', the one norm of \( A \) is returned.

If norm = 'I', 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 \( m = 0 \) 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 155 on page 629.
2. This function accepts lowercase letters for the norm, uplo, and 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 88.
4. For a description of trapezoidal matrices and how they are stored in upper- and lower-trapezoidal storage mode, see “Trapezoidal Matrix” on page 91.
5. For SLANTR, DLANTR, CLANTR, and ZLANTR, the following cases are extensions to the LAPACK standard:
   • uplo = 'U' and \( m > n \)
   • uplo = 'L' and \( n > m \)

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>'I'</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. \( norm \neq 'O', '1', 'I', 'F', 'E', \) or 'M'
2. \( uplo \neq 'U' \) 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 )
```

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 )
```

Output:
ANORM = 632
Example 3

This example computes the infinity norm of complex trapezoidal matrix $A$ stored in lower-trapezoidal storage mode.

Call Statements and Input:

```
ANORM = ZLANTR('I', 'L', 'N', 5, 4, A, 5, WORK )
```

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

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:

```
ANORM = ZLANTR('M', 'U', 'U', 4, 5, A, 4, WORK )
```

```
| (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) |
```

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:

```
ANORM = DLANTP('I', 'L', 'N', 9, AP, WORK )
```

```
| 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    .        .        .        .        |
```

Output:

ANORM = 45.0

Example 6

This example computes the Frobenius norm of real triangular matrix $A$, stored in upper-triangular-packed storage mode.

Call Statements and Input:

```
NORM UPLO DIAG N   AP   WORK
ANORM = DLANTP( 'F', 'U', 'U', 9 , AP , WORK )
```

```
| 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 = 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:

```
NORM UPLO DIAG N   AP   WORK
ANORM = ZLANTP( 'I', 'L', 'N', 5 , AP , WORK )
```

```
| (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:

```
NORM  UPLO  DIAG  N   AP   WORK
|     |    |    |    |     |
ANORM = ZLANTP( 'M', 'U', 'U', 5, AP, WORK )
```

```
| (1.0, 0.0)(1.0, 1.0)(1.0, 1.0)(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) |

AP =
```

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 157: Data Types.**

<table>
<thead>
<tr>
<th>( A, B )</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGBSV^A</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGBSV^A</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGBSV^A</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGBSV^A</td>
</tr>
</tbody>
</table>

^ALAPACK
### Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGBSV</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgbsv</td>
</tr>
<tr>
<td>LAPACK</td>
<td>info = LAPACKE_sgbsv</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. The value of the `matrix_layout` argument 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 ≥ 0.

**kl**
- is the lower band width kl of the matrix A.
- Specified as: an integer; kl ≥ 0.

**ku**
- is the upper band width ku of the matrix A.
- Specified as: an integer; ku ≥ 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 general band matrix A of order n.
- Specified as: an lda by (at least) n array, containing numbers of the data type indicated in Table 157 on page 635.

**lda**
- is the leading dimension of the array specified for a.
- Specified as: an integer; lda > 0 and lda ≥ 2kl+ku+1.

**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 (at least) nrhs array, containing numbers of the data type indicated in Table 157 on page 635.

**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. See “Function” on page 637.
Returned as: an lda by (at least) n array, containing numbers of the data type indicated in Table 157 on page 635.

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 ≤ ipiv ≤ n for all j.

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 matrix X.
Returned as: an ldb by (at least) nrhs array, containing numbers of the data type indicated in Table 157 on page 635.

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 solution has not been computed.
Returned as:
• For SGBSV, DGBSV, CGBSV, and ZGBSV, returned as: an integer; info ≥ 0.
• For LAPACKE_sgbsv, LAPACKE_dgbsv, LAPACKE_cgbsv, and LAPACKE_zgbsv, returned as an integer function value; info ≥ 0.

Notes:
1. In your C program, argument info must be passed by reference.
2. a, 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 94.
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 [“9” on page 1253] and [“48” on page 1255].

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:

```
| N  KL  KU  NRHS  A  LDA  IPIV  B  LDB  INFO |
CALL DGBSV(  9 , 2, 3 , 3 , A , 8 , IPIV , B , 9 , INFO )
```

A = (same as input A in Example 1)
B = (same as input B in Example 1)

Output:

```
| .  .  .  .  4.000  4.000  4.000  4.000 |
| .  .  .  3.000  3.000  3.000  3.000  -2.271 |
| .  .  2.000  2.000  2.000  2.000  2.000  -4.074 -1.747 |
| .  1.000  1.000  1.000  1.000  -4.691 -4.177  1.000 |
| .  .  1.000  1.000  1.000  1.000  1.000  1.000 |
A = | 2.000  2.000  2.000  2.000  -4.419 -3.174  2.000  0.927 |
| 3.000  3.000  3.000  3.000  -5.617 -5.095  3.000  -1.546  3.037 |
| 0.666  0.444  0.518  0.567 -0.334  0.326  0.043 -0.790   .   |
| 0.333 -0.111  0.592  0.246 -0.829 -0.588 -0.617   .      .   |
| .  .  0.043  0.496   .      .  0.043 -0.790   .      .   |
| .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .|
```

IPIV = ( 3, 4, 5, 6, 5, 6, 9, 8, 9 )

```
| 0.629  1.149  4.713 |
| -0.025  1.876  -0.726 |
| 0.523  0.615  2.946 |
| -0.286  1.434  2.774 |
| -0.184 -0.524  1.067 |
| -0.118  0.591  0.970 |
| 0.220  0.896  2.207 |
| -0.079  1.604  0.340 |
| -0.496  0.480  3.599 |
```

B = ... (output values) ...
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( 5 , 1 , 2 , 3 , A , 5 , IPIV , B , 5 , INFO )
```

$A =$ (same as input $A$ in Example 2)

$B =$ (same as input $B$ in Example 3)

Output:

```
| . . . ( 2.000, 4.000) ( 3.000, 5.000) | . . ( 2.000, 3.000) ( 3.000, 4.000) ( 4.000, 5.000) |
A = | ( 2.000, 1.000) ( 3.000, 2.000) ( 4.000, 3.000) ( 5.000, 4.000) |
    | ( 0.200, 0.200) ( 0.300, 0.300) ( 0.400, 0.400) ( 0.500, 0.500) |
    | (-0.339, -0.596) |
    | ( 0.060, 0.020) ( 0.534, 0.305) ( 0.443, 0.300) |
    | (-0.412, -0.396) |
```

```
IPIV = ( 2, 3, 4, 5, 5 )
```

```
| (-0.039, -0.250)( 0.237, -0.265)(-0.247, -0.723) |
B = | ( 0.161, -0.014)( 0.158, 0.163)(-0.092, -0.192) |
    | ( 0.265, 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) |
```

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>SGBTRFΔ</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGBTRFΔ</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGBTRFΔ</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGBTRFΔ</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

Table 160: Syntax for supported computing languages.

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGBTRF</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgbtrf</td>
</tr>
<tr>
<td>LAPACKE</td>
<td>info = LAPACKE_sgbtrf</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. The value of the matrix_layout argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

m
is the number of rows in matrix A.
Specified as: an integer; m ≥ 0.

n
is the number of columns in matrix A.
Specified as: an integer; n ≥ 0.

kl
is the lower band width kl of the matrix A.
Specified as: an integer; kl ≥ 0.

ku
is the upper band width ku of the matrix A.
Specified as: an integer; ku ≥ 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 640.
**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” on page 641.

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 \( \text{SGBTRF, DGBTRF, CGBTRF and ZGBTRF} \), returned as: an integer; \( info \geq 0 \).
- For \( \text{LAPACKE_sgbtrf, LAPACKE_dgbtrf, LAPACKE_cgbtrf and LAPACKE_zgbtrf} \), returned as an integer function value; \( info \geq 0 \).

**Note:**

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 94.

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 \( \text{SGBTRS, DGBTRS, CGBTRS, and ZGBTRS} \) respectively.

If \( m \) or \( n \) is 0, no computation is performed and the subroutine returns after doing some parameter checking.

See references [“9” on page 1253] and [“48” on page 1255].
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. $lda < 0$
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:

$$
\begin{bmatrix}
1.0 & 2.0 & 3.0 & 4.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
2.0 & 1.0 & 2.0 & 3.0 & 4.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 3.0 & 2.0 & 1.0 & 2.0 & 3.0 & 4.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 3.0 & 2.0 & 1.0 & 2.0 & 3.0 & 4.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 3.0 & 2.0 & 1.0 & 2.0 & 3.0 & 4.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 3.0 & 2.0 & 1.0 & 2.0 & 3.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 3.0 & 2.0 & 1.0 & 2.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 3.0 & 2.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 3.0 & 2.0 & 1.0 \\
\end{bmatrix}
$$

Call Statement and Input:

```c
CALL DGBTRF( 9 , 9 , 2 , 3 , A , 8 , IPIV , INFO)
```

Output:

$$
\begin{bmatrix}
1.0000 & 4.0000 & 4.0000 & 4.0000 & 4.0000 \\
2.0000 & 2.0000 & 2.0000 & 2.0000 & 2.0000 \\
3.0000 & 3.0000 & 3.0000 & 3.0000 & -2.2710 \\
2.0000 & 2.0000 & 2.0000 & 2.0000 & -4.0744 \\
3.0000 & 3.0000 & 3.0000 & 3.0000 & -4.1771 \\
0.6659 & 0.4444 & 0.5184 & 0.5674 & -0.3345 \\
3.0000 & 3.0000 & 3.0000 & -3.6172 & -5.0952 \\
3.0000 & 3.0000 & -3.1744 & -0.7908 & -0.7908 \\
3.0000 & 3.0000 & -4.1771 & -4.1771 & -4.1771 \\
\end{bmatrix}
$$
Example 2

This example shows a factorization of the following complex general band matrix of order 5. Matrix $A$ is:

$$
\begin{pmatrix}
(0.100, 0.100) & (1.000, 2.000) & (1.000, 3.000) & (0.000, 0.000) & (0.000, 0.000) \\
(2.000, 1.000) & (0.200, 0.200) & (2.000, 3.000) & (2.000, 4.000) & (0.000, 0.000) \\
(0.000, 0.000) & (3.000, 2.000) & (0.300, 0.300) & (3.000, 4.000) & (3.000, 5.000) \\
(0.000, 0.000) & (0.000, 0.000) & (4.000, 3.000) & (0.400, 0.400) & (4.000, 5.000) \\
(0.000, 0.000) & (0.000, 0.000) & (0.000, 0.000) & (5.000, 4.000) & (0.500, 0.500) \\
\end{pmatrix}
$$

Call Statement and Input:

```
CALL ZGBTRF(5, 5, 2, 2, A, 5, IPIV, INFO)
```

Output:

```
IPIV = ( 2, 3, 4, 5, 5 )
INFO = 0
```

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

**Purpose**

SGBTRS and DGBTRS solve one of the following systems of equations for multiple right-hand sides:

1. $AX=B$
2. $A^TX=B$

CGBTRS and ZGBTRS solve one of the following systems of equations for multiple right-hand sides:

1. $AX=B$
2. $A^TX=B$
3. $A^{iH}X=B$

In the formulas above:
• $A$ represents the general band matrix $A$ stored in BLAS-general-band storage mode, 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 matrix $A$ and vector $ipiv$, produced by a preceding call to SGBTRF, DGBTRF, CGBTRF, and ZGBTRF, respectively.

### Table 161: Data Types.

<table>
<thead>
<tr>
<th>$A, B$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGBTRS$^\Delta$</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGBTRS$^\Delta$</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGBTRS$^\Delta$</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGBTRS$^\Delta$</td>
</tr>
</tbody>
</table>

$^\Delta$LAPACK

**Note:** The input to these solve subroutines must be the output from the factorization subroutines SGBTRF, DGBTRF, CGBTRF, and ZGBTRF, respectively.

### Syntax

### Table 162: Syntax for supported computing languages.

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGBTRS</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgbtrs</td>
</tr>
<tr>
<td>LAPACK</td>
<td>info = LAPACKE_sgbtrs</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. The value of the $matrix_layout$ argument 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; $trans = 'N'$, '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 161 on page 644.

$lda$
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 161 on page 644.

$ldb$
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 161 on page 644.

$info$
has the following meaning:
If $info = 0$, the subroutine completed successfully.
GBTRS, 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; $info \geq 0$.

Note:
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 _GTBRF 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 94.

Function

One of the following systems of equations is solved for multiple right-hand sides:

1. \( AX = B \)
2. \( A^TX = B \) (only for CGBTRS and ZGBTRS)

where \( A \) is a general band matrix and \( B \) and \( X \) are general matrices. These subroutines uses 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 639.

If \( n \) or \( nrhs \) is 0, no computation is performed.

See reference [“48” on page 1255].

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. \( \text{trans} \neq 'N', 'T', \text{or} 'C' \)
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:

```plaintext
TRANS N KL KU NRHS A LDA IPIV B LDB INFO)
CALL DGBTRS( 'N' , 9 , 2 , 3 , 3 , A , 8 , IPIV , B , 9 , INFO)
```

A = (same as output A in Example 1)

IPIV = (same as output IPIV in Example 1)

B =

```
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  
```

Output:

```
0.629  1.149  4.713  
-0.025  1.870 -0.726  
 0.523  0.615  2.946  
-0.286 -1.434 -2.774  
-0.184 -0.524 -1.667  
-0.118 -0.591 -0.970  
 0.220 -0.896  2.027  
-0.079  1.604 -0.340  
 0.496  0.480  3.599  
```

INFO = 0

**Example 2**

This example shows how to solve the system $A^T X = B$, where real general band matrix $A$ is the same matrix factored in Example 1 for DGBTRF.

Call Statement and Input:

```plaintext
TRANS N KL KU NRHS A LDA IPIV B LDB INFO)
CALL DGBTRS( 'T' , 9 , 2 , 3 , 3 , A , 8 , IPIV , B , 9 , INFO)
```

A = (same as output A in Example 1)

IPIV = (same as output IPIV in Example 1)

B =

```
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  
```

Output:

```
0.496  0.480  1.362  
-0.079  1.604 -0.450  
 0.220 -0.896  0.179  
```

Linear algebraic equations subroutines 647
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:

```
TRANS N KL KU NRHS A LDA IPIV B LDB INFO)
CALL ZGBTRS('N', 5, 1, 2, 3, A, 5, IPIV, B, 5, INFO)
```

A = (same as output A in Example 2)

IPIV = (same as output IPIV in Example 2)

Output:

```
| ( 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) |
```

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 = (same as output A in Example 2)

IPIV = (same as output IPIV in Example 2)

Output:

```
| ( 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) |
```

INFO = 0

---

IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and Reference
Output:

\[
\begin{pmatrix}
( \ 0.015, -0.126) & ( \ 0.150, -0.096) & ( \ 0.029, \ 0.017) \\
( -0.158, \ 0.428) & ( -0.607, \ 0.209) & ( -0.002, \ 0.200) \\
B = & ( -0.094, \ 0.283) & ( -0.392, \ 0.149) & ( -0.054, \ 0.230) \\
( -0.057, -0.111) & ( -0.070, -0.161) & ( -0.114, \ 0.109) \\
( -0.088, -0.469) & ( \ 0.367, -0.460) & ( -0.111, -0.050)
\end{pmatrix}
\]

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>
<thead>
<tr>
<th>Table 163: Data Types.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Types</strong></td>
</tr>
<tr>
<td>( A, b, x )</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>Long-precision real</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>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGBS</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgbs</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 163 on page 649, 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 163 on page 649.

**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 163 on page 649.

**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 “Vector concepts” on page 71.

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 94.

**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 694. See reference [“48” on page 1255].

**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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
CALL SGBS( AGB , 23 , 9 , 2 , 3 , IPVT , BX )
```

IPVT = (2, -65534, -131070, -196606, -262142, -327678, -327678, -327680, -327680)
BX = (4.0000, 5.0000, 9.0000, 10.0000, 11.0000, 12.0000, 12.0000, 12.0000, 33.0000)
AGB = (same as output AGB in Example)

Output:

```
<table>
<thead>
<tr>
<th>BX</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1.0000 , 1.0000 , 1.0000 , 1.0000 , 1.0000 , 1.0000 , 0.9999 , 1.0001)</td>
</tr>
</tbody>
</table>
```

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 164: Data Types.

<table>
<thead>
<tr>
<th>$A, B$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SPBSVA</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DPBSVA</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CPBSVA</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZPBSVA</td>
</tr>
</tbody>
</table>

LAPACK

Syntax

Table 165: Syntax for supported computing languages.

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SPBSV</td>
</tr>
<tr>
<td>C and C++</td>
<td>spbsv</td>
</tr>
</tbody>
</table>
Table 165: Syntax for supported computing languages. (continued)

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPACKE</td>
<td>`info = LAPACKE_spbsv</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. The value of the `matrix_layout` argument 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 ≥ 0`.

- **k**
  - Is the half band width `k` of the matrix `A`. Specified as: an integer; `0 ≤ k ≤ 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 ≥ 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 164 on page 651.

- **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 651.

- **ldb**
  - Is the leading dimension of the array specified for `B`. Specified as: an integer; `ldb > 0` and `ldb ≥ n`.

**On Return**

- **a**
  - If `info = 0`, `a` is the updated matrix `A` containing the results of the Cholesky factorization. See “Function” on page 653.
  - Returned as: an `lda` by (at least) `n` array, containing numbers of the data type indicated in Table 164 on page 651.
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 164 on page 651.

$info$ 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 100 or “Lower-Band-Packed Storage Mode” on page 100, 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 102.

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 “Vector concepts” on page 71.

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 a 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 [“9” on page 1253], [“46” on page 1255], and [“75” on page 1257].

**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. \( uplo \neq 'U' \) 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 \( A \) 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:

\[
A = \begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & . \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & . & . \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & . & . & . \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
1.0 & 1.0 & 1.0 \\
1.0 & -1.0 & . \\
1.0 & 1.0 & -1.0 \\
1.0 & -1.0 & 1.0 \\
1.0 & 1.0 & 0.0 \\
1.0 & -1.0 & -1.0 \\
\end{bmatrix}
\]
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 $U^T U$.

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 )
```

Output:

```
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:

```
CALL ZPBSV ( 'L' , 6 , 3 , 3 , A , 4 , B , 6 , INFO )
```

Output:

```
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 \( U^*U \).

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:

\[
\text{UPLO} \quad \text{N} \quad \text{K} \quad \text{NRHS} \quad \text{A} \quad \text{LDA} \quad \text{B} \quad \text{LDB} \quad \text{INFO} \\
\]

\[
\text{CALL ZPBSV( 'U' , 6 , 2 , 3 , A , 3 , B , 6 , INFO )} \\
\]

\[ A = \text{(same as input A in Example 4)} \]

\[ B = \text{(same as input B in Example 4)} \]

Output:

\[
A = \begin{pmatrix}
(1.0, -1.0) & (1.0, 1.0) & (1.0, -1.0) & (1.0, 1.0) \\
(1.0, 1.0) & (1.0, -1.0) & (1.0, 1.0) & (1.0, -1.0) \\
(1.0, 0.0) & (1.0, 0.0) & (1.0, 0.0) & (1.0, 0.0) \\
(5.0, 13.0) & (1.0, -1.0) & (1.0, 1.0) & (1.0, -1.0) \\
\end{pmatrix}
\]

\[
B = \begin{pmatrix}
(5.0, -1.0) & (1.0, -1.0) & (1.0, 1.0) \\
(-3.0, 7.0) & (1.0, -1.0) & (1.0, 1.0) \\
(3.0, 7.0) & (1.0, -1.0) & (1.0, 1.0) \\
(1.0, 1.0) & (1.0, -1.0) & (1.0, 2.0) \\
\end{pmatrix}
\]

\[ \text{INFO} = 0 \]

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 166: Data Types. (continued)

<table>
<thead>
<tr>
<th>A</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long-precision real</td>
<td>DPBTRFΔ</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CPBTRFΔ</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZPBTRFΔ</td>
</tr>
<tr>
<td>ΔLAPACK</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 167: Syntax for supported computing languages.

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SPBTRF</td>
</tr>
<tr>
<td>C and C++</td>
<td>spbtrf</td>
</tr>
<tr>
<td>LAPACKE</td>
<td>info = LAPACKE_spbtrf</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. The value of the matrix_layout argument 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 ≥ 0.

k
is the half band width k of the matrix A. Specified as: an integer; 0 ≤ k ≤ 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 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 166 on page 656.

lda
is the leading dimension of the array specified for A. Specified as: an integer; lda > 0 and lda > k.
On Return

If $info = 0$, $a$ is the updated matrix $A$ containing the results of the Cholesky factorization. See “Function” on page 658.

Returned as: an lda by (at least) $n$ array, containing numbers of the data type indicated in Table 166 on page 656.

$info$

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 ZPBTRF, returned as: an integer; info $\geq 0$.
- For LAPACKE_spbtrf, LAPACKE_dpbtrf, LAPACKE_cpbtrf, and LAPACKE_zpbtrf, returned as an integer function value; info $\geq 0$.

Notes:

1. These subroutines accept lower case letters for the $uplo$ 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 100 or “Lower-Band-Packed Storage Mode” on page 100, 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 102.
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 an 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. $uplo \neq 'U$ 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 $A = LL^T$:

$$
\begin{bmatrix}
  1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
  1.0 & 2.0 & 2.0 & 2.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
  1.0 & 2.0 & 3.0 & 3.0 & 2.0 & 1.0 & 0.0 & 0.0 & 0.0 \\
  1.0 & 2.0 & 3.0 & 4.0 & 3.0 & 2.0 & 1.0 & 0.0 & 0.0 \\
  0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 3.0 & 2.0 & 1.0 & 0.0 \\
  0.0 & 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 3.0 & 2.0 & 1.0 \\
  0.0 & 0.0 & 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 3.0 & 2.0 \\
  0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 3.0 \\
  0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & 3.0 & 4.0 \\
\end{bmatrix}
$$

Call Statement and Input:

```
UPLO  N   K    A  LDA  INFO
|    |   |    |   |    |
CALL DPBTRF( 'L' , 9 , 3 ,  A , 4 , INFO )
```

Output:

```
A = 
\begin{bmatrix}
  1.0 & 2.0 & 3.0 & 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 \\
  1.0 & 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 \\
\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 \\
\end{bmatrix}
```

---

Linear algebraic equations subroutines 659
Example 3

This example shows a factorization of the following positive definite complex Hermitian band matrix $A$ in the form $A = LL^H$:

\[
\begin{bmatrix}
1.0, 0.0 & 1.0, -1.0 & 1.0, -1.0 & 1.0, -1.0 & 1.0, 0.0 & 1.0, 0.0 \\
1.0, 1.0 & 3.0, 0.0 & 3.0, -1.0 & 3.0, -1.0 & 1.0, -1.0 & 1.0, -1.0 \\
1.0, 1.0 & 3.0, 1.0 & 5.0, 0.0 & 5.0, -1.0 & 3.0, -1.0 & 1.0, -1.0 \\
0.0, 0.0 & 1.0, 1.0 & 3.0, 1.0 & 5.0, 1.0 & 7.0, 0.0 & 5.0, -1.0 \\
0.0, 0.0 & 0.0, 0.0 & 1.0, 1.0 & 3.0, 1.0 & 5.0, 1.0 & 7.0, 0.0 \\
0.0, 0.0 & 0.0, 0.0 & 0.0, 0.0 & 1.0, 1.0 & 3.0, 1.0 & 5.0, 1.0 \\
\end{bmatrix}
\]
Example 4

This example shows a factorization of the following positive definite complex Hermitian band matrix \( A \) in the form \( A = U^H U \):

\[
\begin{bmatrix}
(1.0, 0.0) & (1.0, 1.0) & (1.0, -1.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) \\
(1.0, -1.0) & (3.0, 0.0) & (1.0, -3.0) & (1.0, 1.0) & (0.0, 0.0) & (0.0, 0.0) \\
(1.0, 1.0) & (1.0, 1.0) & (1.0, 3.0) & (1.0, 1.0) & (1.0, -1.0) & (0.0, 0.0) \\
(0.0, 0.0) & (1.0, -1.0) & (1.0, -3.0) & (5.0, 0.0) & (1.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (0.0, 0.0) & (1.0, 1.0) & (1.0, 3.0) & (5.0, 0.0) & (1.0, 0.0) \\
(0.0, 0.0) & (0.0, 0.0) & (1.0, 0.0) & (1.0, 0.0) & (1.0, -1.0) & (1.0, -3.0) & (5.0, 0.0)
\end{bmatrix}
\]

Call Statement and Input:

```
UPLO  N   K    A  LDA  INFO
   |   |    |   |    |
CALL ZPBTRF( 'U' , 6 , 2 , A , 3 , INFO )
```

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, -3.0) (1.0, 3.0) (1.0, 3.0) (1.0, 3.0) |
(1.0, 0.0) (3.0, 0.0) (5.0, 0.0) (5.0, 0.0) (5.0, 0.0) |
```

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>
<thead>
<tr>
<th>Table 168: Data Types.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A, B</strong></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>

\(\Delta\) LAPACK

Note: The input to these solve subroutines must be the output from the factorization subroutines SPBTRF, DPBTRF, CPBTRF, and ZPBTRF, respectively.

Linear algebraic equations subroutines 661
### Syntax

Table 169: Syntax for supported computing languages.

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SPBTRS</td>
</tr>
<tr>
<td>C and C++</td>
<td>spbtrs</td>
</tr>
<tr>
<td>LAPACK</td>
<td>info = LAPACKE_spbtrs</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. The value of the `matrix_layout` argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

- **uplo** indicates whether the factored 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 $A$ and the number of rows of matrix $B$. Specified as: an integer; $n \geq 0$.

- **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 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 168 on page 661.

- **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 168 on page 661.

- **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 168 on page 661.

**info**

\(info\) has the following meaning:

If \(info = 0\), the solve completed successfully.

\(S\)

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 100 or “Lower-Band-Packed Storage Mode” on page 100, 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 102.
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 “Vector concepts” on page 71.

**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 656.

If \(n\) or \(nrhs\) is 0, no computation is performed. See references [“9” on page 1253] and [“46” on page 1255].

**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. \(uplo \neq 'U'\) or \('L'\)
2. \(n < 0\)
3. $k < 0$
4. $k > \max(0,n-1)$
5. $\text{nrhs} < 0$
6. $\text{lda} \leq 0$
7. $k \geq \text{lda}$
8. $\text{ldb} \leq 0$
9. $n > \text{ldb}$

**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 )
```

$A = \text{(same output A as in Example 1)}$

```
| 4.0  0.0  1.0 |
| 8.0  0.0  1.0 |
|12.0  0.0  0.0 |
|16.0  0.0  1.0 |
|16.0  0.0  0.0 |
|16.0  0.0 -1.0 |
|15.0  1.0  0.0 |
|13.0  1.0 -2.0 |
|10.0  2.0 -3.0 |
```

$B =$

```
| 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 |
```

Output:

```
| 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 |
```

$\text{INFO} = 0$

**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:

```
UPLO  N   K  NRHS A  LDA  B  LDB  INFO
|    |   |   |   |   |   |   |    |
CALL DPBTRS( 'U' , 9 , 2 , 3 , A , 3 , B , 9 , INFO )
```

$A = \text{(same as output A in Example 2)}$

```
| 1.0  3.0  0.0 |
| 0.0 -6.0  2.0 |
| 1.0  9.0 -4.0 |
| 1.0 -9.0  4.0 |
```

664 IBM Engineering and Scientific Subroutine Library for Linux on POWER  Version 6.1: ESSL Guide and Reference
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:

```
      UPLO  N   K  NRHS A  LDA  B  LDB  INFO
      |    |   |   |   |   |   |   |    |
      CALL ZPBTRS('L', 6 , 3 , 3 , A , 4 , B , 6 , INFO )
```

$A = \text{(same as output $A$ in Example 3)}$

```
| (  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) |
```

$B = \text{(same as output $B$ in Example 3)}$

```
| (  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) |
```

Output:

```
| (  7.0,  33.0)  (  1.0,  -1.0)  (  1.0,  1.0) |
| ( 19.0,  -1.0)  (  1.0,  -1.0)  (  1.0,  1.0) |
| (-11.0,  -5.0)  (  1.0,  -1.0)  (  2.0,  1.0) |
```

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:

```
      UPLO  N   K  NRHS A  LDA  B  LDB  INFO
      |    |   |   |   |   |   |   |    |
      CALL ZPBTRS('U', 6 , 2 , 3 , A , 4 , B , 6 , INFO )
```

Output:

```
| (  7.0,  33.0)  (  1.0,  -1.0)  (  1.0,  1.0) |
| ( 19.0,  -1.0)  (  1.0,  -1.0)  (  1.0,  1.0) |
| (-11.0,  -5.0)  (  1.0,  -1.0)  (  2.0,  1.0) |
```

INFO = 0

Linear algebraic equations subroutines 665
\[ A = \begin{pmatrix} 1.0 & 1.0 \\ 1.0 & 1.0 \\ 1.0 & 1.0 \end{pmatrix} \begin{pmatrix} 3.0 & -3.0 \\ 3.0 & -9.0 \\ 15.0 & -3.0 \end{pmatrix} \begin{pmatrix} 6.0 & 0.0 \\ 13.0 & -7.0 \\ 22.0 & 15.0 \end{pmatrix} \]

\[ B = \begin{pmatrix} 1.0 & 1.0 \\ 1.0 & 1.0 \\ 1.0 & 1.0 \end{pmatrix} \begin{pmatrix} 3.0 & -15.0 \\ 3.0 & -1.0 \\ 3.0 & -11.0 \end{pmatrix} \begin{pmatrix} 25.0 & -14.0 \\ 18.0 & 19.0 \\ 13.0 & -14.0 \end{pmatrix} \]

Output:

\[ B = \begin{pmatrix} 5.0 & 13.0 \\ 1.0 & -1.0 \\ 1.0 & 1.0 \end{pmatrix} \begin{pmatrix} 1.0 & -1.0 \\ 1.0 & -1.0 \\ 1.0 & -2.0 \end{pmatrix} \begin{pmatrix} 1.0 & 1.0 \\ 1.0 & 1.0 \\ 1.0 & 1.0 \end{pmatrix} \begin{pmatrix} 1.0 & -1.0 \\ 1.0 & -1.0 \\ 1.0 & -1.0 \end{pmatrix} \begin{pmatrix} 1.0 & 1.0 \\ 1.0 & 1.0 \\ 1.0 & 1.0 \end{pmatrix} \begin{pmatrix} 1.0 & -1.0 \\ 1.0 & -1.0 \\ 1.0 & -1.0 \end{pmatrix} \begin{pmatrix} 1.0 & 1.0 \\ 1.0 & 1.0 \\ 1.0 & 1.0 \end{pmatrix} \]

INFO = 0

\section*{SGTSV, DGTSS, CGTSV, and ZGTSV (General Tridiagonal Matrix Factorization and Multiple Right-Hand Side Solve)}

\textbf{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.

\textbf{Table 170: Data Types.}

\begin{tabular}{|l|l|}
\hline
\textit{dl, d, du, B} & \textit{Subroutine} \\
\hline
\text{Short-precision real} & \text{SGTSV}^{\Delta} \\
\text{Long-precision real} & \text{DGTSS}^{\Delta} \\
\text{Short-precision complex} & \text{CGTSV}^{\Delta} \\
\text{Long-precision complex} & \text{ZGTSV}^{\Delta} \\
\hline
\end{tabular}

\text{\textsuperscript{\Delta}LAPACK}

\textbf{Syntax}

\textbf{Table 171: Syntax for supported computing languages.}

\begin{tabular}{|l|l|}
\hline
\textbf{Language} & \textbf{Syntax} \\
\hline
\text{Fortran} & \text{CALL SGTSV | DGTSS | CGTSV | ZGTSV \( (n, nrhs, dl, d, du, b, ldb, info) \)} \\
\text{C and C++} & \text{sgtsv | dtgsv | cgtsv | ztgsv \( (n, nrhs, dl, d, du, b, ldb, info) \)}; \\
\text{LAPACKE} & \text{info = LAPACKE\_sgtsv \ LAPACKE\_dgtss \ LAPACKE\_cgtsv \ LAPACKE\_ztgsv (matrix\_layout, n, nrhs, dl, d, du, b, ldb);} \\
\hline
\end{tabular}

\textbf{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. The value of the *matrix_layout* argument 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* ≥ 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* ≥ 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 170 on page 666.

*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 170 on page 666.

*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 170 on page 666.

*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 170 on page 666.

*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**

*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 170 on page 666.

*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 170 on page 666.

*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 170 on page 666.
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 170 on page 666.

$info$

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, DGTSV, 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 106.
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 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 [“9” on page 1253].

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

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:

```
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:

```
   0.609   0.478   4.597
  0.098   0.130  -0.899
 -0.231  -0.641  -2.723
  0.234   0.312   2.105
  0.364   0.153   2.516
 -0.017  -0.023  -0.958
 -0.019  -0.359  -0.147
  0.267   0.357   2.505
  0.198  -0.070   1.484
```

INFO = 0

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:

\[ B = \begin{bmatrix}
(-0.247, 0.0) & (0.119, 0.0) & (0.0, 0.247) \\
(0.311, 0.0) & (0.220, 0.0) & (0.0, -0.311) \\
(0.357, 0.0) & (-0.394, 0.0) & (0.0, -0.357) \\
(-0.073, 0.0) & (0.183, 0.0) & (0.0, 0.073)
\end{bmatrix} \]

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 172: 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( ^{\Delta} )</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGTTRF( ^{\Delta} )</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGTTRF( ^{\Delta} )</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGTTRF( ^{\Delta} )</td>
</tr>
</tbody>
</table>

\( ^{\Delta} \) 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**

**Table 173: Syntax for supported computing languages.**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGTTRF</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgttrf</td>
</tr>
<tr>
<td>LAPACKE</td>
<td>info = LAPACKE_sgttrf</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. The value of the \( \text{matrix\_layout} \) argument 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 172 on page 670.

$d$
- 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 172 on page 670.

$du$
- 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 172 on page 670.

$du2$
- 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 172 on page 670.

$d$
- 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 172 on page 670.

$du$
- 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 172 on page 670.

$du2$
- 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 172 on page 670.

$ipiv$
- 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 $U_{ii}$ is zero. The factorization has been completed.
Returned as:
- For SGTTTRF, DGTTTRF, CGTTRF, and ZGTTRF, returned as: an integer; $info \geq 0$.
- For LAPACKE_sgtttrf, LAPACKE_dgttrf, LAPACKE_cgttrf, and LAPACKE_zgttrf, returned as an integer function value; $info \geq 0$.

Notes:
1. In your C program, argument $info$ must be passed by reference.
2. $dl$, $d$, $du$, $du2$, 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 106.
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 [“9” on page 1253].

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:

| 1.0 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 3.0 1.0 4.0 0.0 0.0 0.0 0.0 0.0 |
Call Statement and Input:

\[
\begin{array}{cccccccc}
N & DL & D & DU & DU2 & IPIV & INFO \\
\hline
9 & DL & D & DU & DU2 & IPIV & INFO \\
\end{array}
\]

\[
\text{CALL DGTTRF}( 9 , DL , D , DU , DU2 , IPIV , INFO )
\]

\[
\begin{align*}
DL &= (3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0) \\
D &= (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0) \\
DU &= (4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0)
\end{align*}
\]

Output:

\[
\begin{align*}
DL &= (0.333, 0.818, 0.696, 0.908, 0.849, -0.799, 0.625, -0.332) \\
D &= (3.000, 3.666, 3.000, 3.303, 3.532, 3.000, 4.799, 3.000, 4.332) \\
DU &= (1.000, -1.333, 1.000, -2.787, 4.000, 1.000, 3.196, 1.000) \\
DU2 &= (4.000, 0.000, 4.000, 0.000, 4.000, 0.000) \\
IPIV &= (2, 2, 4, 4, 5, 7, 7, 9, 9) \\
INFO &= 0
\end{align*}
\]

Example 2

This example shows a factorization of a complex general tridiagonal matrix of order 4.

Matrix \( A \) is:

\[
\begin{pmatrix}
(1.0, 1.0) & (4.0, 4.0) & (0.0, 0.0) & (0.0, 0.0) \\
(3.0, 3.0) & (1.0, 1.0) & (4.0, 4.0) & (0.0, 0.0) \\
(0.0, 0.0) & (3.0, 3.0) & (1.0, 1.0) & (4.0, 4.0) \\
(0.0, 0.0) & (0.0, 0.0) & (3.0, 3.0) & (1.0, 1.0)
\end{pmatrix}
\]

Call Statement and Input:

\[
\begin{array}{cccccccc}
N & DL & D & DU & DU2 & IPIV & INFO \\
\hline
4 & DL & D & DU & DU2 & IPIV & INFO \\
\end{array}
\]

\[
\text{CALL ZGTTRF}( 4 , DL , D , DU , DU2 , IPIV , INFO )
\]

\[
\begin{align*}
DL &= (3.0, 3.0, 3.0, 3.0) \\
D &= (1.0, 1.0, 1.0, 1.0) \\
DU &= (4.0, 4.0, 4.0, 4.0)
\end{align*}
\]

Output:

\[
\begin{align*}
DL &= (0.0333, 0.0, 0.0818, 0.0, 0.0696, 0.0) \\
D &= (3.0, 3.0, 3.0666, 3.0666, 3.0, 3.0, 3.0303, 3.0303) \\
DU &= (1.0, 1.0, -1.0333, -1.0333) \\
DU2 &= (4.0, 4.0, 0.0, 0.0) \\
IPIV &= (2, 2, 4, 4) \\
INFO &= 0
\end{align*}
\]
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. $A^H X = B$

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 174: Data Types.

<table>
<thead>
<tr>
<th>$dl$, $d$, $du$, $du2$, $B$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGTTRS</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGTTRS</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGTTRS</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGTTRS</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 175: Syntax for supported computing languages.

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGTTRS</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgttrs</td>
</tr>
<tr>
<td>LAPACK</td>
<td>info = LAPACKE_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 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. The value of the matrix_layout argument 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; $trans = 'N'$, 'T', or 'C'.

$n$ is the order of factored matrix $A$ and the number of rows in matrix $B$.
Specified as: an integer; $n \geq 0$.

**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, DGTTRF, 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 174 on page 674.

**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, DGTTRF, CGTTRF, or ZGTTRF, respectively.
Specified as: a one-dimensional array of (at least) length $n$, containing numbers of the data type indicated in Table 174 on page 674.

**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, DGTTRF, 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 174 on page 674.

**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, DGTTRF, 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 174 on page 674.

**ipiv** is the array containing the pivot indices produced by a preceding call to SGTTRF, DGTTRF, CGTTRF, and ZGTTRF, respectively.
Specified as: a one-dimensional array of (at least) length $n$, containing integers; $1 \leq ipiv_i \leq n$.

**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 174 on page 674.

**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 174 on page 674.

\(info\) has the following meaning:

If \(info = 0\), the subroutine completed successfully.

Returned as:

- For \(>\text{SGTTRS, DGTTRS, CGTTRS, and ZGTTRS}\), returned as: an integer; \(info \geq 0\).
- For \(\text{LAPACKE_sgttrs, LAPACKE_dgttrs, LAPACKE_cgttrs, and LAPACKE_zgttrs}\), 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 \(\text{SGTTRS and DGTTRS}\), 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 \(\text{GTTRF}\) and \(\text{GTTRS}\).
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 \(\text{SGTTRF, DGTTRF, CGTTRF, and ZGTTRF}\), respectively.
7. For a description of how general tridiagonal matrices are stored, see “General Tridiagonal Matrix” on page 106.

Function

One of the following systems of equations is solved for multiple right-hand sides:

1. \(AX=B\)
2. \(A^TX=B\) (only for \(\text{CGTTRS and ZGTTRS}\))

where \(A\) is a general tridiagonal matrix and \(B\) and \(X\) are general matrices. These subroutines uses the results of the factorization of matrix \(A\), produced by a preceding call to \(\text{SGTTRF, DGTTRF, CGTTRF, or ZGTTRF}\), respectively. For details on the factorization, see “\(\text{SGTTRF, DGTTRF, CGTTRF, and ZGTTRF}\) (General Tridiagonal Matrix Factorization)” on page 670.

If \(n\) is 0 or \(nrhs\) is 0, no computation is performed and the subroutine returns after doing some parameter checking.

See reference [“9" on page 1253].

Error conditions

Resource Errors
None

Computational Errors
None

\textbf{Note:} If the factorization performed by \(\text{SGTTRF, DGTTRF, CGTTRF, or ZGTTRF}\) 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 DGTTRF.

Call Statement and Input:

```fortran
CALL DGTTRS( 'N', 9, 3, DL, D, DU, DU2, IPIV, B, 9, INFO)
```

\( DL = \) (same as output DL in Example 1)
\( D = \) (same as output D in Example 1)
\( DU = \) (same as output DU in Example 1)
\( DU2 = \) (same as output DU2 in Example 1)

\( IPIV = \) (same as output IPIV in Example 1)

```
B =
| 1.0  1.0  1.0 |
| 1.0 -1.0  2.0 |
| 1.0  1.0  3.0 |
| 1.0 -1.0  4.0 |
```

Output:

```
B =
| 0.609  0.478  4.597 |
| 0.098  0.130 -0.899 |
| -0.231 -0.641 -2.723 |
| 0.234  0.312  2.105 |
```

\( INFO = 0 \)

**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:

```fortran
CALL DGTTRS( 'N', 9, 3, DL, D, DU, DU2, IPIV, B, 9, INFO)
```

\( DL = \) (same as output DL in Example 1)
\( D = \) (same as output D in Example 1)
\( DU = \) (same as output DU in Example 1)
\( DU2 = \) (same as output DU2 in Example 1)

\( IPIV = \) (same as output IPIV in Example 1)

```
B =
| 1.0  1.0  1.0 |
| 1.0 -1.0  2.0 |
| 1.0  1.0  3.0 |
| 1.0 -1.0  4.0 |
```

Output:

```
B =
| 0.609  0.478  4.597 |
| 0.098  0.130 -0.899 |
| -0.231 -0.641 -2.723 |
| 0.234  0.312  2.105 |
```

\( INFO = 0 \)
CALL DGTTRS( 'T',  9,  3,  DL, D, DU, DU2, IPIV, B,  9, INFO)

DL = (same as output DL in Example 1)
D = (same as output D in Example 1)
DU = (same as output DU in Example 1)
DU2 = (same as output DU2 in Example 1)
IPIV = (same as output IPIV in Example 1)

\[
\begin{pmatrix}
1.0 & 1.0 & 1.0 \\
1.0 & -1.0 & 2.0 \\
1.0 & 1.0 & 3.0 \\
1.0 & -1.0 & 4.0 \\
1.0 & 1.0 & 5.0 \\
1.0 & -1.0 & 6.0 \\
1.0 & 1.0 & 7.0 \\
1.0 & -1.0 & 8.0 \\
1.0 & 1.0 & 9.0 \\
\end{pmatrix}
\]

| 0.198 | -0.070 | 0.491 |
| 0.267 | 0.356 | 0.170 |
| -0.017 | -0.035 | -0.045 |
| 0.365 | 0.153 | 1.130 |
| 0.234 | 0.312 | 0.238 |
| -0.230 | -0.641 | 0.414 |
| 0.979 | 0.130 | 1.878 |
| 0.609 | 0.478 | 1.489 |

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:

```latex
CALL ZGTTRS( 'N',  4,  3,  DL, D, DU, DU2, IPIV, B,  4, INFO)
```

DL = (same as output DL in Example 2)
D = (same as output D in Example 2)
DU = (same as output DU in Example 2)
DU2 = (same as output DU2 in Example 2)
IPIV = (same as output IPIV in Example 2)

\[
\begin{pmatrix}
(1.0, 1.0) & (1.0, 1.0) & (1.0, -1.0) \\
(1.0, 1.0) & (-1.0, -1.0) & (1.0, -1.0) \\
(1.0, 1.0) & (1.0, 1.0) & (1.0, -1.0) \\
(1.0, 1.0) & (-1.0, -1.0) & (1.0, -1.0) \\
\end{pmatrix}
\]
Example 4

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 ZGTRTF.

Call Statement and Input:

\[
\begin{array}{cccccccc}
\text{TRANS} & \text{N} & \text{NRHS} & \text{DL} & \text{D} & \text{DU} & \text{DU2} & \text{IPIV} & \text{B} & \text{LDB} & \text{INFO} \\
\end{array}
\]

\[
\text{CALL ZGTTRS( 'T', 4, 3, DL, D, DU, DU2, IPIV, B, 4, INFO) }
\]

DL = (same as output DL in Example 2)

D = (same as output D in Example 2)

DU = (same as output DU in Example 2)

DU2 = (same as output DU2 in Example 2)

IPIV = (same as output IPIV in Example 2)

Output:

\[
\begin{array}{cccccccc}
\text{TRANS} & \text{N} & \text{NRHS} & \text{DL} & \text{D} & \text{DU} & \text{DU2} & \text{IPIV} & \text{B} & \text{LDB} & \text{INFO} \\
\end{array}
\]

\[
\text{CALL ZGTTRS( 'T', 4, 3, DL, D, DU, DU2, IPIV, B, 4, INFO) }
\]

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

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 subroutine solves \( A^T X = B \).
### Table 176: 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(^A)</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DPTSV(^A)</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>Short-precision complex</td>
<td>CPTSV(^A)</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision complex</td>
<td>ZPTSV(^A)</td>
</tr>
</tbody>
</table>

\(^A\) LAPACK

### Syntax

#### Table 177: Syntax for supported computing languages.

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SPTSV</td>
</tr>
<tr>
<td>C and C++</td>
<td>sptsv</td>
</tr>
<tr>
<td>LAPACK</td>
<td>info = LAPACKE_sptsv</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. The value of the \(matrix\_layout\) argument 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 176 on page 680.

- **\(e\)**
  - is the vector \(e\) containing the subdiagonal or superdiagonal of matrix \(A\) in positions 1 through \(n\) in an array referred to as \(E\). Specified as: a one-dimensional array, of (at least) length \(n\) 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 176 on page 680.

- **\(b\)**
  - 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 176 on page 680.

- **\(ldb\)**
  - is the leading dimension of the array specified for \(B\). Specified as: an integer; \(ldb > 0\) and \(ldb \geq n\).

**On Return**
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 176 on page 680.

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 176 on page 680. It has the same length as $E$ on entry.

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 176 on page 680.

$info$ 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 LAPACKE_sptsv, LAPACKE_dptsv, LAPACKE_cptsv, and LAPACKE_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 108. 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 109.
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 $A^T X = B$ and $A = U^H DU$.

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 [“9” on page 1253],[“46” on page 1255], and [“75” on page 1257].

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 )
```

D = (same as output D in Example 1)
E = (same as output E in Example 1)
B = (same as input B in Example 1)

Output:

```
D = ( 1.0, 1.0, 2.0, 0.5 )
E = ( 1.0, 1.0, 0.5 )
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 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:
\[
\begin{array}{cccccc}
N & NRHS & D & E & B & LDB & INFO \\
\mid & | & | & | & | & | \\
\end{array}
\]
\[
\text{CALL ZPTSV( 4, 3, D, E, B, 4, INFO )}
\]

\[
\begin{array}{c}
D = \text{(same as output D in Example 2)} \\
E = \text{(same as output E in Example 2)} \\
B = \text{(same as input B in Example 2)}
\end{array}
\]

Output:
\[
\begin{array}{c}
D = (1.0 \ 2.0 \ 3.0 \ 4.0) \\
E = (1.0, 1.0) (1.0, 1.0) (1.0, 1.0) \\
\mid (1.0, 0.0) (1.0, 1.0) (1.0, -1.0) \\
B = (1.0, 0.0) (1.0, 1.0) (1.0, -1.0) \\
\mid (1.0, 0.0) (1.0, 1.0) (1.0, -1.0) \\
INFO = 0
\end{array}
\]

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:
\[
\begin{array}{cccccc}
N & NRHS & D & E & B & LDB & INFO \\
\mid & | & | & | & | & | \\
\end{array}
\]
\[
\text{CALL ZPTSV( 4, 3, D, E, B, 4, INFO )}
\]

\[
\begin{array}{c}
D = \text{(same as output D in Example 3)} \\
E = \text{(same as output E in Example 3)} \\
B = \text{(same as input B in Example 3)}
\end{array}
\]

Output:
\[
\begin{array}{c}
D = (1.0 \ 2.0 \ 3.0 \ 4.0) \\
E = (1.0, -1.0) (1.0, -1.0) (1.0, 1.0) \\
\mid (1.0, 0.0) (1.0, 1.0) (1.0, -1.0) \\
B = (1.0, 0.0) (1.0, 1.0) (1.0, -1.0) \\
\mid (1.0, 0.0) (1.0, 1.0) (1.0, -1.0) \\
INFO = 0
\end{array}
\]
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 \( A = UU^D \)

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>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SPTTRFΔ</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DPTTRFΔ</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>CPTTRFΔ</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>ZPTTRFΔ</td>
</tr>
</tbody>
</table>

Note: The output from these factorization subroutines should be used only as input to the solve subroutines SPTTRS, DPTTRS, CPTTRS, or ZPTTRS, respectively.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SPTTRF</td>
</tr>
<tr>
<td>C and C++</td>
<td>spttrf</td>
</tr>
<tr>
<td>LAPACK</td>
<td>info = LAPACKE_spttrf</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. The value of the `matrix_layout` argument 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 178 on page 684.

\( e \) 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 178 on page 684.

**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 178 on page 684.

\( e \)

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 178 on page 684. 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 ≥ 0.
- For LAPACKE_spttrf, LAPACKE_dpttrf, LAPACKE_cpttrf, and LAPACKE_zpttrf, returned as an integer function value; info ≥ 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 108. 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 109.

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 \( U^TDU \) 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 \( A = U^HDU \)

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 [“9” on page 1253], [“46” on page 1255], and [“75” on page 1257].

**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:**

```
N   D   E   INFO
|  |  |  |
CALL DPTTRF( 4 , D , E , INFO )
```

\( D = ( 1.0, 2.0, 3.0, 1.0 ) \)

\( E = ( 1.0, 1.0, 1.0 ) \)

**Output:**

\( D = ( 1.0, 1.0, 2.0, 0.5 ) \)

\( E = ( 1.0, 1.0, 0.5 ) \)

\( INFO = 0 \)
Example 2
This example shows a factorization of the positive definite complex Hermitian tridiagonal matrix $A$, in the form $A = LDL^H$:

\[
\begin{pmatrix}
1.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 1.0 & 0.0 & 0.0 \\
0.0 & 2.0 & 2.0 & 0.0 \\
0.0 & 0.0 & 3.0 & 3.0 \\
\end{pmatrix}
\begin{pmatrix}
1.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 1.0 & 0.0 & 0.0 \\
0.0 & 2.0 & 2.0 & 0.0 \\
0.0 & 0.0 & 3.0 & 3.0 \\
\end{pmatrix}
\]

Call Statement and Input:

\[
\begin{aligned}
N & \quad D & \quad E & \quad INFO \\
\end{aligned}
\]

\[
\begin{aligned}
\text{CALL ZPTTRF( 4 , D , E , INFO )} \\
D &= ( 1.0 4.0 7.0 10.0 ) \\
E &= ( ( 1.0, 1.0) ( 2.0, 2.0) ( 3.0, 3.0) ) \\
\end{aligned}
\]

Output:

\[
\begin{aligned}
D &= ( 1.0 2.0 3.0 4.0 ) \\
E &= ( ( 1.0, 1.0) ( 1.0, 1.0) ( 1.0, 1.0) ) \\
\text{INFO} &= 0 \\
\end{aligned}
\]

Example 3
This example shows a factorization of the positive definite complex Hermitian tridiagonal matrix $A$, in the form $A = U^HDU$:

\[
\begin{pmatrix}
1.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 1.0 & 0.0 & 0.0 \\
0.0 & 2.0 & 2.0 & 0.0 \\
0.0 & 0.0 & 3.0 & 3.0 \\
\end{pmatrix}
\begin{pmatrix}
1.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 1.0 & 0.0 & 0.0 \\
0.0 & 2.0 & 2.0 & 0.0 \\
0.0 & 0.0 & 3.0 & 3.0 \\
\end{pmatrix}
\]

Call Statement and Input:

\[
\begin{aligned}
N & \quad D & \quad E & \quad INFO \\
\end{aligned}
\]

\[
\begin{aligned}
\text{CALL ZPTTRF( 4 , D , E , INFO )} \\
D &= ( 1.0 4.0 7.0 10.0 ) \\
E &= ( ( 1.0, -1.0) ( 2.0, -2.0) ( 3.0, -3.0) ) \\
\end{aligned}
\]

Output:

\[
\begin{aligned}
D &= ( 1.0 2.0 3.0 4.0 ) \\
E &= ( ( 1.0, -1.0) ( 1.0, -1.0) ( 1.0, -1.0) ) \\
\text{INFO} &= 0 \\
\end{aligned}
\]

**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 $\text{uplo} = 'L'$, then this subroutine solves $AX = B$.
- If $\text{uplo} = 'U'$, then this subroutine solves $A^TX = B$.

If, in the call to CPTTRF or ZPTTRF, you specified the superdiagonal of $A$ in $e$:

- If $\text{uplo} = 'L'$, then this subroutine solves $A^TX = B$.
- If $\text{uplo} = 'U'$, 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 179: 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$^\Delta$</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DPTTRS$^\Delta$</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>Short-precision complex</td>
<td>CPTTRS$^\Delta$</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision complex</td>
<td>ZPTTRS$^\Delta$</td>
</tr>
</tbody>
</table>

$^\Delta$ LAPACK

**Note:** The input to these solve subroutines must be the output from the factorization subroutines SPTTRF, DPTTRF, CPTTRF, or ZPTTRF respectively.

### Syntax

### Table 180: Syntax for supported computing languages.

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SPTTRS</td>
</tr>
<tr>
<td></td>
<td>CALL CPTTRS</td>
</tr>
<tr>
<td>C and C++</td>
<td>spttrs</td>
</tr>
<tr>
<td></td>
<td>cpttrs</td>
</tr>
<tr>
<td>LAPACK</td>
<td>info = LAPACKE_spttrs</td>
</tr>
<tr>
<td></td>
<td>info = LAPACKE_cpttrs</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. The value of the $\text{matrix_layout}$ argument must be \text{LAPACK\_ROW\_MAJOR} or \text{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 $\text{uplo} = 'L'$, $e$ is the subdiagonal of the unit bidiagonal lower triangular factor $L$.
  - If $\text{uplo} = 'U'$, $e$ 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 \(D\). Specified as: a one-dimensional array of (at least) length \(n\), containing numbers of the data type indicated in Table 179 on page 688.

\(e\)

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 179 on page 688.

\(b\)

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 179 on page 688.

\(ldb\)

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 solutions to the system.

Returned as: an \(ldb\) by (at least) \(nrhs\) array, containing numbers of the data type indicated in Table 179 on page 688.

\(info\)

\(info\) has the following meaning:

If \(info = 0\), the solve completed successfully.

Returned as:

• For SPTTRS, DPTTRS, CPTTRS, and ZPTTRS, returned as: an integer; \(info \geq 0\).

• For LAPACKE_spttrs, LAPACKE_dpttrs, LAPACKE_cpttrs, and LAPACKE_zpttrs, 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 tridiagonal matrices are stored in LAPACK-symmetric-tridiagonal storage mode, see “LAPACK-Symmetric-Tridiagonal Storage Mode” on page 108. 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 109.
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 \( A^\top X = B \).

- If, in the call to CPTTRF or ZPTTRF, you specified the superdiagonal of \( A \) in \( e \):
  - If \( uplo = 'L' \), then this subroutine solves \( A^\top X = B \).
  - 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 684.

If \( n \) or \( nrhs \) is 0, no computation is performed. See references [“9” on page 1253] and [“46” on page 1255].

**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. \( uplo \neq 'U' \) or \( 'L' \)
2. \( n < 0 \)
3. \( nrhs < 0 \)
4. \( ldb \leq 0 \)
5. \( n > ldb \)

**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 \( LDL^\top \).

Call Statement and Input:

```
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)

\[
B = \begin{bmatrix}
2.0 & -2.0 \\
4.0 & -3.0 \\
5.0 & 0.0 \\
2.0 & 1.0
\end{bmatrix}
\]

Output:

\[
B = \begin{bmatrix}
1.0 & -1.0 \\
1.0 & -1.0 \\
1.0 & 0.0 \\
1.0 & 1.0
\end{bmatrix}
\]

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:

```fortran
CALL ZPTTRS('L', 4, 3, D, E, B, 4, INFO)
```

\[
D = \text{(same as output D in Example 2)}
\]

\[
E = \text{(same as output E in Example 2)}
\]

\[
B = \begin{bmatrix}
(2.0, -1.0) & (3.0, 1.0) & (1.0, -3.0) \\
(7.0, -1.0) & (8.0, 6.0) & (6.0, -8.0) \\
(12.0, -1.0) & (13.0, 11.0) & (11.0, -13.0) \\
(13.0, 3.0) & (10.0, 16.0) & (16.0, -10.0)
\end{bmatrix}
\]

Output:

\[
B = \begin{bmatrix}
(1.0, 0.0) & (1.0, 1.0) & (1.0, -1.0) \\
(1.0, 0.0) & (1.0, 1.0) & (1.0, -1.0) \\
(1.0, 0.0) & (1.0, 1.0) & (1.0, -1.0) \\
(1.0, 0.0) & (1.0, 1.0) & (1.0, -1.0)
\end{bmatrix}
\]

INFO = 0

**Example 3**

This example shows how to solve the system of linear equations \(A^TX = B\) 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:

```fortran
CALL ZPTTRS('U', 4, 3, D, E, B, 4, INFO)
```

\[
D = \text{(same as output D in Example 2)}
\]

\[
E = \text{(same as output E in Example 2)}
\]

\[
B = \begin{bmatrix}
(2.0, -1.0) & (3.0, 1.0) & (1.0, -3.0) \\
(7.0, -1.0) & (8.0, 6.0) & (6.0, -8.0) \\
(12.0, -1.0) & (13.0, 11.0) & (11.0, -13.0) \\
(13.0, 3.0) & (10.0, 16.0) & (16.0, -10.0)
\end{bmatrix}
\]

Output:

\[
B = \begin{bmatrix}
(1.0, 0.0) & (1.0, 1.0) & (1.0, -1.0) \\
(1.0, 0.0) & (1.0, 1.0) & (1.0, -1.0) \\
(1.0, 0.0) & (1.0, 1.0) & (1.0, -1.0) \\
(1.0, 0.0) & (1.0, 1.0) & (1.0, -1.0)
\end{bmatrix}
\]

INFO = 0
Example 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 $U^HDU$.

Call Statement and Input:

```
UPLO N NRHS D   E   B  LDB  INFO
|   |   |   |   |   |   |    |
CALL ZPTTRS( 'U', 4 , 3 , D , E , B , 4 , INFO )
```

$D = \text{(same as output D in Example 3)}$

$E = \text{(same as output E in Example 3)}$

$B = \begin{bmatrix}
(2.0, -1.0) & (3.0, 1.0) & (1.0, -3.0) \\
(7.0, -1.0) & (8.0, 6.0) & (6.0, -8.0) \\
(12.0, -1.0) & (13.0, 11.0) & (11.0, -13.0) \\
(13.0, 3.0) & (10.0, 16.0) & (16.0, -10.0)
\end{bmatrix}$

Output:

```
| (3.00, -3.33) & (6.33, -0.33) & (-0.33, -6.33) |
| (0.66, 1.66) & (-1.00, 2.33) & (2.33, 1.00) |
B = \begin{bmatrix}
(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

Example 5
This example shows how to solve the system of linear equations $A^TX = B$ where positive definite complex Hermitian tridiagonal matrix $A$ is the same matrix factored in Example 3 for ZPTTRF in the form $U^HDU$.

Call Statement and Input:

```
UPLO N NRHS D   E   B  LDB  INFO
|   |   |   |   |   |   |    |
CALL ZPTTRS( 'U', 4 , 3 , D , E , B , 4 , INFO )
```

$D = \text{(same as output D in Example 3)}$

$E = \text{(same as output E in Example 3)}$

$B = \begin{bmatrix}
(2.0, -1.0) & (3.0, 1.0) & (1.0, -3.0) \\
(7.0, -1.0) & (8.0, 6.0) & (6.0, -8.0) \\
(12.0, -1.0) & (13.0, 11.0) & (11.0, -13.0) \\
(13.0, 3.0) & (10.0, 16.0) & (16.0, -10.0)
\end{bmatrix}$

Output:

```
| (1.0, 0.0) & (1.0, 1.0) & (1.0, -1.0) |
| (1.0, 0.0) & (1.0, 1.0) & (1.0, -1.0) |
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)
\end{bmatrix}
```

INFO = 0
\[ B = \begin{pmatrix} 7.0 & -1.0 \\ 12.0 & -1.0 \\ 13.0 & 3.0 \end{pmatrix} \begin{pmatrix} 8.0 & 6.0 \\ 13.0 & 11.0 \\ 10.0 & 16.0 \end{pmatrix} \begin{pmatrix} 6.0 & -8.0 \\ 11.0 & -13.0 \\ 16.0 & -10.0 \end{pmatrix} \]

Output:

\[ B = \begin{pmatrix} 3.00 & -3.33 \\ 0.66 & 1.66 \\ 0.83 & -1.50 \end{pmatrix} \begin{pmatrix} 6.33 & -0.33 \\ -1.00 & 2.33 \\ 2.33 & 1.00 \end{pmatrix} \begin{pmatrix} 0.33 & -6.33 \\ 2.33 & 1.00 \\ 1.66 & -2.33 \end{pmatrix} \]

\[ B = \begin{pmatrix} 3.00 & -3.33 \\ 0.66 & 1.66 \\ 0.83 & -1.50 \end{pmatrix} \begin{pmatrix} 6.33 & -0.33 \\ -1.00 & 2.33 \\ 2.33 & 1.00 \end{pmatrix} \begin{pmatrix} 0.33 & -6.33 \\ 2.33 & 1.00 \\ 1.66 & -2.33 \end{pmatrix} \]

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>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</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>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGBF ( agb ), lda, n, ml, mu, ipvt )</td>
</tr>
<tr>
<td>C and C++</td>
<td><code>sgbf (agb, lda, n, ml, mu, ipvt);</code></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 181 on page 693, 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 of order , containing the results of the factorization. See “Function” on page 694. Returned as: an by (at least) array, containing numbers of the data type indicated in Table 181 on page 693.

**ipvt**
is the integer vector of length , containing the pivot information necessary to construct matrix from the information contained in the output array . Returned as: a one-dimensional array of (at least) length , containing integers.

**Notes**
1. is not a permutation vector in the strict sense. It is used to record column interchanges in due to partial pivoting and to improve performance.
2. The entire array specified for must remain unchanged between calls to the factorization and solve subroutines.
3. This subroutine can be used for tridiagonal matrices (); however, the tridiagonal subroutines and 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 94.

**Function**
The general band matrix , stored in general-band storage mode, is factored using Gaussian elimination with partial pivoting to compute the factorization of , 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 contains in packed format, along with the multipliers necessary to construct, with the help of , a matrix , such that . This factorization can then be used by SGBS or DGBS, respectively, to solve the system of equations. See reference [“48” on page 1255].

**Error conditions**

**Resource Errors**
Unable to allocate internal work area.

**Computational Errors**
Matrix is singular.
- One or more columns of and the corresponding diagonal of contain all zeros (all columns of are checked). The last column, of , with a corresponding diagonal element is identified in the computational error message.
- The return code is set to 1.
- 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 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 65.

**Input-Argument Errors**
1. 
2. 

---

694 IBM Engineering and Scientific Subroutine Library for Linux on POWER  Version 6.1: ESSL Guide and Reference
3. \( ml \geq n \)
4. \( mu < 0 \)
5. \( mu \geq n \)
6. \( lda < 2ml+mu+16 \)

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:

\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
4.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 5.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 6.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 7.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 8.0 & 1.0 & 1.0 & 1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 9.0 & 1.0 & 1.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 10.0 & 11.0 & 12.0
\end{bmatrix}
\]

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 )
```

```
AGB =
| 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000 |
| 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000 |
| 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000 |
| 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000 |
| 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000 |
| 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000 |
| 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000 |
| 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000 |
| 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000 |
| 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000 |
```

Linear algebraic equations subroutines 695
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 182: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A, b, x$</td>
<td></td>
</tr>
<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>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGBS</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgsbs</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 182 on page 696, where $lda \geq 2ml+mu+16$. 

696 IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and Reference
**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 182 on page 696.

**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 182 on page 696.

**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 “Vector concepts" on page 71.
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 94.

**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 694. See reference [“48" on page 1255].

**Error conditions**

**Computational Errors**

**Note:** If the factorization performed by SGBF or DGBF failed due to a singular matrix argument, the results returned by this subroutine are unpredictable, and there may be a divide-by-zero program exception message.
Input-Argument Errors

1. $lda \leq 0$
2. $ml < 0$
3. $ml \geq n$
4. $mu < 0$
5. $mu \geq n$
6. $lda < 2ml+mu+16$

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 )
```

IPVT = (2, -65534, -131070, -196606, -262142, -327678, -327678, -327680, -327680)
BX = (4.0000, 5.0000, 9.0000, 10.0000, 11.0000, 12.0000, 12.0000, 12.0000, 33.0000)
AGB = (same as output $AGB$ in Example)

Output:

```
BX       =  (1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 0.9999, 1.0001)
```

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

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>
<thead>
<tr>
<th>Data Types</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

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SPBF</td>
</tr>
<tr>
<td>C and C++</td>
<td>spbf</td>
</tr>
</tbody>
</table>

**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 183 on page 698. See “Notes” on page 699.

- **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; $0 ≤ m < n$.

**On Return**

- **apb** is the transformed matrix $A$ of order $n$, containing the results of the factorization. See “Function” on page 699. Returned as: an `lda` by (at least) $n$ array, containing numbers of the data type indicated in Table 183 on page 698. For further details, see “Notes” on page 699.

### 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 101.

### 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 ERRESET 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 programs,” on page 125.

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 ERRESET 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 programs,” on page 125.

**Input-Argument Errors**

1. $lda \leq 0$
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:

```
  | 1.0  1.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0 |
  | 1.0  2.0  2.0  1.0  0.0  0.0  0.0  0.0  0.0 |
  | 1.0  2.0  3.0  2.0  1.0  0.0  0.0  0.0  0.0 |
  | 0.0  1.0  2.0  3.0  2.0  1.0  0.0  0.0  0.0 |
  | 0.0  0.0  1.0  2.0  3.0  2.0  1.0  0.0  0.0 |
  | 0.0  0.0  0.0  1.0  2.0  3.0  2.0  1.0  0.0 |
  | 0.0  0.0  0.0  0.0  1.0  2.0  3.0  2.0  1.0 |
  | 0.0  0.0  0.0  0.0  0.0  1.0  2.0  3.0  2.0 |
  | 0.0  0.0  0.0  0.0  0.0  0.0  1.0  2.0  3.0 |
```

and on output, matrix $A$ is:

```
  | 1.0  1.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0 |
  | 1.0  1.0  1.0  1.0  0.0  0.0  0.0  0.0  0.0 |
  | 1.0  1.0  1.0  1.0  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 |
```
where array location APB(2, 9) is set to 0.0.

Call Statement and Input:

```
CALL SPBF( APB , 3 , 9 , 2 )
```

Output:

```
APB = | 1.0 2.0 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 .   .   . |
     | 1.0 1.0 1.0 1.0 1.0 1.0 1.0 .    .    . |
```

**Example 2**

This example shows a Cholesky factorization of the same matrix used in Example 1.

Call Statement and Input:

```
CALL SPBCHF( APB , 3 , 9 , 2 )
```

APB = (same as input APB in Example 1)

Output:

```
APB = | 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 |
     | 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 0.0 |
     | 1.0 1.0 1.0 1.0 1.0 1.0 1.0 .    .    . |
```

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

**Purpose**

These subroutines solve the system \( \mathbf{A} \mathbf{x} = \mathbf{b} \) for \( \mathbf{x} \), where \( \mathbf{A} \) is a positive definite symmetric band matrix, and \( \mathbf{x} \) and \( \mathbf{b} \) are vectors. They use the results of the factorization of matrix \( \mathbf{A} \), 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 184: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathbf{A}, \mathbf{b}, \mathbf{x} )</td>
<td>SPBS and SPBCHS</td>
</tr>
</tbody>
</table>
**Table 184: Data Types. (continued)**

<table>
<thead>
<tr>
<th>Data Types</th>
<th></th>
</tr>
</thead>
<tbody>
<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 698.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SPBS</td>
</tr>
<tr>
<td>C and C++</td>
<td>spbs</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 184 on page 701. See “Notes” on page 702.

- `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 184 on page 701.

**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 184 on page 701.

**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 “Vector concepts” on page 71.
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 101.
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:

```
APB  LDA  N   M   BX
CALL SPBS( APB , 3 , 9 , 2 , BX )
```

$APB =$ (same as output $APB$ in Example 1)

$BX = (3.0, 6.0, 9.0, 9.0, 9.0, 9.0, 8.0, 6.0)$

Output:

```
BX       =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
```

This example shows how to solve the system $Ax = b$, where matrix $A$ is the same matrix factored in the Example 2 for SPBCHF and DPBCHF, using Cholesky factorization.

Call Statement and Input:

```
APB  LDA  N   M   BX
CALL SPBCHS( APB , 3 , 9 , 2 , BX )
```

$APB =$ (same as output $APB$ in Example 2)

$BX = (3.0, 6.0, 9.0, 9.0, 9.0, 9.0, 9.0, 8.0, 6.0)$

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>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>c, d, e, f</td>
<td></td>
</tr>
<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>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGTF</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgft</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 185 on page 704.
- $d$ 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 185 on page 704.
- $e$ 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 185 on page 704.
- $f$ 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 185 on page 704.
\( \mathbf{d} \) is the vector \( \mathbf{d} \), containing part of the factorization of matrix \( \mathbf{A} \) in an array, referred to as \( \mathbf{D} \). Returned as: a one-dimensional array of (at least) length \( n \), containing numbers of the data type indicated in Table 185 on page 704.

\( \mathbf{e} \) is the vector \( \mathbf{e} \), containing part of the factorization of the matrix \( \mathbf{A} \) in positions 1 through \( n \) in an array, referred to as \( \mathbf{E} \). Returned as: a one-dimensional array of (at least) length \( n \), containing numbers of the data type indicated in Table 185 on page 704.

\( \mathbf{f} \) is the vector \( \mathbf{f} \), containing part of the factorization of matrix \( \mathbf{A} \) in the first \( n \) positions in an array, referred to as \( \mathbf{F} \). Returned as: a one-dimensional array of (at least) length \( n \), containing numbers of the data type indicated in Table 185 on page 704.

\( \mathbf{ipvt} \) is the integer vector \( \mathbf{ipvt} \) 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 106.
2. \( \mathbf{ipvt} \) 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 \( \mathbf{A} \) is stored in nonstandard format.

Function
The standard Gaussian elimination with partial pivoting of tridiagonal matrix \( \mathbf{A} \) is computed. The factorization is returned by overwriting input arrays \( \mathbf{C} \), \( \mathbf{D} \), and \( \mathbf{E} \), and by writing into output array \( \mathbf{F} \), along with pivot information in vector \( \mathbf{ipvt} \). This factorization can then be used by SGTS or DGTS, respectively, to solve tridiagonal systems of linear equations. See references [“53” on page 1256], [“65” on page 1256], [“66” on page 1256], and [“111” on page 1259]. If \( n \) is 0, no computation is performed.

Error conditions
Computational Errors
Matrix \( \mathbf{A} \) 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 65.

Input-Argument Errors
\( n < 0 \)

Examples
Example
This example shows how to factor the following tridiagonal matrix \( \mathbf{A} \) of order 4:

\[
\begin{pmatrix}
2.0 & 2.0 & 0.0 & 0.0 \\
1.0 & 3.0 & 2.0 & 0.0 \\
0.0 & 1.0 & 3.0 & 2.0 \\
0.0 & 0.0 & 1.0 & 3.0
\end{pmatrix}
\]
Call Statement and Input:

```
| 0.0  0.0  1.0  3.0 |
```

Call Statement and Input:

```
CALL DGTF( 4 , C , D , E , F , IPVT )
```

C        =  ( . , 1.0, 1.0, 1.0)
D        =  (2.0, 3.0, 3.0, 3.0)
E        =  (2.0, 2.0, 2.0, . )

Output:

C        =  ( . , -0.5, -0.5, -0.5)
D        =  (-0.5, -0.5, -0.5, -0.5)
E        =  (2.0, 2.0, 2.0, . )
IPVT     =  (X'00', X'00', X'00', X'00')

Notes :
1. F is stored in an internal format and is passed unchanged to the solve subroutine.
2. A "." means you do not have to store a value in that position in the array. However, these storage positions are required and may be overwritten during the computation.

**SGTS and DGTS (General Tridiagonal Matrix Solve)**

**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 186: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>c, d, e, f, b, x</td>
<td>SGTS</td>
</tr>
<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>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGTS</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgt</td>
</tr>
</tbody>
</table>

**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 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 186 on page 706.
\( d \) 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 186 on page 706.

\( e \) 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 186 on page 706.

\( f \) 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 186 on page 706.

\( \text{ipvt} \) 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 186 on page 706. For details on specifying the length, see “Notes” on page 707.

**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 186 on page 706. For details about the length, see “Notes” on page 707.

**Notes**

1. For a description of how tridiagonal matrices are stored, see “General Tridiagonal Matrix” on page 106.
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 [“53” on page 1256], [“65” on page 1256], [“66” on page 1256], and [“111” on page 1259].

**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 )
```

\[
\begin{align*}
  C &= \text{(same as output } C \text{ in Example)} \\
  D &= \text{(same as output } D \text{ in Example)} \\
  E &= \text{(same as output } E \text{ in Example)} \\
  F &= \text{(same as output } F \text{ in Example)} \\
  \text{IPVT} &= \text{(same as output IPVT in Example)} \\
  BX &= (4.0, 6.0, 6.0, 4.0, .)
\end{align*}
\]

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.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
\textbf{Data Types} & \textbf{c, d, e, b, x} & \textbf{Subroutine} \\
\hline
Short-precision real & SGTNP \\
Long-precision real & DGTNP \\
Short-precision complex & CGTNP \\
Long-precision complex & ZGTNP \\
\hline
\end{tabular}
\end{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

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGTNP</td>
</tr>
</tbody>
</table>

708 IBM Engineering and Scientific Subroutine Library for Linux on POWER  Version 6.1: ESSL Guide and Reference
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 187 on page 708. 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 187 on page 708. 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 187 on page 708. 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 187 on page 708.

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 187 on page 708.

Notes

For a description of how tridiagonal matrices are stored, see “General Tridiagonal Matrix” on page 106.

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:

\[
\begin{pmatrix}
7.0 & 4.0 & 0.0 & 0.0 \\
1.0 & 8.0 & 5.0 & 0.0 \\
0.0 & 2.0 & 9.0 & 6.0 \\
0.0 & 0.0 & 3.0 & 10.0
\end{pmatrix}
\]

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:

\[
\text{CALL DGTNP( 4 , C , D , E , BX )}
\]

\[
\begin{align*}
C & = ( . , 1.0, 2.0, 3.0) \\
D & = (7.0, 8.0, 9.0, 10.0) \\
E & = (4.0, 5.0, 6.0, . ) \\
BX & = (11.0, 14.0, 17.0, 13.0)
\end{align*}
\]

Output:

\[
\begin{align*}
BX & = (1.0, 1.0, 1.0, 1.0)
\end{align*}
\]

Example 2
This example shows a factorization of the complex tridiagonal matrix \( A \), of order 4:

\[
\begin{pmatrix}
(7.0, 7.0) & (4.0, 4.0) & (0.0, 0.0) & (0.0, 0.0) \\
(1.0, 1.0) & (8.0, 8.0) & (5.0, 5.0) & (0.0, 0.0) \\
(0.0, 0.0) & (2.0, 2.0) & (9.0, 9.0) & (6.0, 6.0) \\
(0.0, 0.0) & (0.0, 0.0) & (3.0, 3.0) & (10.0, 10.0)
\end{pmatrix}
\]

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:

\[
\text{CALL ZGTNP( 4 , C , D , E , BX )}
\]

\[
\begin{align*}
C & = ( . , (1.0, 1.0), (2.0, 2.0), (3.0, 3.0))
\end{align*}
\]
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 188: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c, d, e$</td>
<td></td>
</tr>
<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

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGTNPF</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgtnpf</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 188 on page 711.
\( d \) 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 188 on page 711.

\( e \) 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 188 on page 711.

\( iopt \) 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 188 on page 711.

\( d \) 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 188 on page 711.

\( e \) 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 188 on page 711. 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 106.

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 [“93” on page 1258].

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. $i	ext{o}pt \neq 0$ or $1$

Examples

Example 1

This example shows a factorization of the tridiagonal matrix $A$, of order 4:

$$
\begin{bmatrix}
1.0 & 1.0 & 0.0 & 0.0 \\
1.0 & 2.0 & 1.0 & 0.0 \\
0.0 & 1.0 & 3.0 & 1.0 \\
0.0 & 0.0 & 1.0 & 1.0 \\
\end{bmatrix}
$$

Call Statement and Input:

```
N   C   D   E   IOPT
|   |   |   |   |
CALL DGTNPF( 4 , C , D , E , 0   )
```

```
C = ( . , 1.0, 1.0, 1.0)
D = (1.0, 2.0, 3.0, 1.0)
E = (1.0, 1.0, 1.0, . )
```

Output:

```
C = ( . , -1.0, -1.0, 1.0)
D = (-1.0, -1.0, -1.0, -1.0)
E = (1.0, 1.0, -1.0, . )
```

Example 2

This example shows a factorization of the tridiagonal matrix $A$, of order 4:

$$
\begin{bmatrix}
(7.0, 7.0) & (4.0, 4.0) & (0.0, 0.0) & (0.0, 0.0) \\
(1.0, 1.0) & (8.0, 8.0) & (5.0, 5.0) & (0.0, 0.0) \\
(0.0, 0.0) & (2.0, 2.0) & (9.0, 9.0) & (6.0, 6.0) \\
(0.0, 0.0) & (0.0, 0.0) & (3.0, 3.0) & (10.0, 10.0) \\
\end{bmatrix}
$$

Call Statement and Input:

```
N   C   D   E   IOPT
|   |   |   |   |
CALL ZGTNPF( 4 , C , D , E , 0   )
```

```
C = ( . , (1.0, 1.0), (2.0, 2.0), (3.0, 3.0))
D = ((7.0, 7.0), (8.0, 8.0), (9.0, 9.0), (10.0, 10.0))
E = ((4.0, 4.0), (5.0, 5.0), (6.0, 6.0), . )
```

Output:

```
C = ( . , (-0.142, 0.0), (-0.269, 0.0), (3.0, 3.0))
D = ((-0.0714, 0.0714), (-0.0673, 0.0673), (-0.0854, 0.0854),
( -0.05, 0.05))
E = ((4.0, 4.0), (5.0, 5.0), (-0.6, 0.0), . )
```

Notes:

1. A "." means you do not have to store a value in that position in the array. However, these storage positions are required and may be overwritten during the computation.
SGTNPS, DGTNPS, CGTNPS, and ZGTNPS (General Tridiagonal Matrix Solve with No Pivoting)

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 189: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c, d, e, b, x$</td>
<td>SGTNPS</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>DGTNPS</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>CGTNPS</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>ZGTNPS</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td></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

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGTNPS</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgtcps</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 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 189 on page 714.

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 189 on page 714.

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 189 on page 714.

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 189 on page 714.

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 189 on page 714.
Notes
For a description of how tridiagonal matrices are stored, see “General Tridiagonal Matrix” on page 106.

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 [“93” on page 1258].

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:

\[
\begin{align*}
\text{CALL DGTNPF(} & 4 , C , D , E , BX ) \\
C & = \text{(same as output C in Example 1)} \\
D & = \text{(same as output D in Example 1)} \\
E & = \text{(same as output E in Example 1)} \\
BX & = (2.0, 4.0, 5.0, 2.0)
\end{align*}
\]

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:

```
N   C   D   E   BX
|   |   |   |   |
CALL ZGTPNPS( 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 190: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$, $d$</td>
<td></td>
</tr>
<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**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SPTF</td>
</tr>
<tr>
<td>C and C++</td>
<td>sptf</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 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 190 on page 716.

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 190 on page 716.

$iopt$

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 190 on page 716.

\( d \)

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 190 on page 716. 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 109.

**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 [“93” on page 1258].

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:

<table>
<thead>
<tr>
<th>N</th>
<th>C</th>
<th>D</th>
<th>IOPT</th>
</tr>
</thead>
</table>
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>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c, d, b, x )</td>
<td></td>
</tr>
<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>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SPTS</td>
</tr>
<tr>
<td>C and C++</td>
<td>spts</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 191 on page 718.
- \( d \) 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 191 on page 718.
- \( bx \) 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 191 on page 718.

**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 191 on page 718.

**Notes**

For a description of how tridiagonal matrices are stored, see “Positive Definite or Negative Definite Symmetric Matrix” on page 84.

**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 [“93” on page 1258].

**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:

```plaintext
<table>
<thead>
<tr>
<th>N</th>
<th>C</th>
<th>D</th>
<th>BX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>.</td>
<td>-1.0</td>
<td>-1.0</td>
</tr>
</tbody>
</table>
CALL DPTS( 4 , C , D , BX )
```

\( C = ( . , -1.0, -1.0, -1.0) \)
\( D = (-1.0, -1.0, -1.0, -1.0) \)
\( BX = (2.0, 4.0, 5.0, 2.0) \)

Output:

\( BX = (1.0, 1.0, 1.0, 1.0) \)

**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; $iopt = 0, 1, \text{or } 2$.

$n$
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 110 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 110 for more information on storage techniques.

$lna$
is the length of the arrays specified for $a$, $ia$, and $ja$. 
Specified as: an integer; \( lna > 2nz \). If you do not specify a sufficient amount, it results in an error. See “Error conditions” on page 723.

The size of \( lna \) depends on the structure of the input matrix. The requirement that \( lna > 2nz \) does not guarantee a successful run of the program. If the input matrix is expected to have many fill-ins, \( lna \) should be set larger. Larger \( lna \) may result in a performance improvement.

For details on how \( lna \) relates to storage compressions, see “Performance and Accuracy Considerations” on page 477.

**iparm**

is an array of parameters, \( \text{IPARM}(i) \), where:

- \( \text{IPARM}(1) \) 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.

- \( \text{IPARM}(2) \) determines the number of minimal Markowitz counts that are examined to determine a pivot. (See reference [“123” on page 1259].)

- \( \text{IPARM}(3) \) has the following meaning, where:
  
  If \( \text{IPARM}(3) = 0 \), this subroutine checks the values in arrays \( \text{IA} \) and \( \text{JA} \).
  
  If \( \text{IPARM}(3) = 1 \), this subroutine assumes that the input values are correct in arrays \( \text{IA} \) and \( \text{JA} \).

- \( \text{IPARM}(4) \) 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 \( \text{U} \).

  These values are stored in \( \text{OPARM}(2) \) and \( \text{OPARM}(3) \), respectively.

- \( \text{IPARM}(5) \) is reserved.

Specified as: an array of (at least) length 5, containing integers, where the \( \text{iparm} \) values must be:

- \( \text{IPARM}(1) = 0 \) or \( 1 \)
- \( \text{IPARM}(2) \geq 1 \)
- \( \text{IPARM}(3) = 0 \) or \( 1 \)
- \( \text{IPARM}(4) = 0 \) or \( 1 \)

**rparm**

is an array of parameters, \( \text{RPARM}(i) \), where:

- \( \text{RPARM}(1) \) 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.

- \( \text{RPARM}(2) \) is the threshold pivot tolerance used to control the choice of pivots.

- \( \text{RPARM}(3) \) is reserved.

- \( \text{RPARM}(4) \) is reserved.

- \( \text{RPARM}(5) \) is reserved.

Specified as: a one-dimensional array of (at least) length 5, containing long-precision real numbers, where the \( \text{rparm} \) values must be:
RPARM(1) ≥ 0.0
0.0 ≤ RPARM(2) ≤ 1.0

For additional information about rparm, see “Performance and Accuracy Considerations” on page 477.

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 ≥ 10n+100.

For 64-bit integer arguments
naux ≥ 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 110.
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 49.
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 [“17” on page 1254], [“58” on page 1256], and [“114” on page 1259].

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 65.

   • 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 \( ina \) 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. \( ina \leq 2nz \)
5. \( IPARM(1) \neq 0 \) or 1
6. \( IPARM(2) \leq 0 \)
7. \( \text{IPARM}(3) \neq 0 \) or \( 1 \)
8. \( \text{IPARM}(4) \neq 0 \) or \( 1 \)
9. \( \text{RPARM}(1) < 0.0 \)
10. \( \text{RPARM}(2) < 0.0 \) or \( \text{RPARM}(2) > 1.0 \)
11. \( \text{ioph} = 1 \) and \( \text{ia}(i) \geq \text{ia}(i+1), \ i = 1, n \)
12. \( \text{ioph} = 2 \) and \( \text{ja}(i) \geq \text{ja}(i+1), \ i = 1, n \)
13. \( \text{ioph} = 0 \) or \( 1 \) and \( \text{ja}(i) < 1 \) or \( \text{ja}(i) > n, \ i = 1, nz \)
14. \( \text{ioph} = 0 \) or \( 1 \) and \( \text{ia}(i) < 1 \) or \( \text{ia}(i) > n, \ i = 1, nz \)
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. \( \text{naux} \) 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 factors 5 by 5 sparse matrix \( A \), which is stored by indices in arrays \( A, \text{IA}, \text{JA} \). 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):

\[
\text{CALL DGSF}( 0, 5, 13, \ A, \text{IA}, \text{JA}, 27, \ \text{IPARM}, \ \text{RPARM}, \ \text{OPARM}, \ \text{AUX}, 150 )
\]

\[
\begin{array}{cccccccccccc}
\text{A} & = & (2.0, & 1.0, & 1.0, & 3.0, & 4.0, & 1.0, & 5.0, & 2.0, & 2.0, & 1.0, & 1.0, \\
& & 4.0, & 3.0, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, \\
\text{IA} & = & (1, & 2, & 2, & 3, & 3, & 4, & 4, & 4, & 5, & 5, & 1, & 2, \\
& & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, \\
\text{JA} & = & (1, & 1, & 2, & 3, & 4, & 4, & 5, & 1, & 2, & 3, & 4, & 3, \\
& & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, \\
\text{IPARM} & = & (1, & 3, & 1, & 1) \\
\text{RPARM} & = & (1.0-12, & 0.100)
\end{array}
\]

Call Statement and Input (Storage-By-Rows):

\[
\text{CALL DGSF}( 1, 5, 13, \ A, \text{IA}, \text{JA}, 27, \ \text{IPARM}, \ \text{RPARM}, \ \text{OPARM}, \ \text{AUX}, 150 )
\]

\[
\begin{array}{cccccccccccc}
\text{A} & = & (2.0, & 4.0, & 1.0, & 1.0, & 3.0, & 3.0, & 4.0, & 2.0, & 2.0, & 1.0, & 5.0, \\
& & 1.0, & 1.0, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, \\
\text{IA} & = & (1, & 3, & 6, & 8, & 12, & 14, & \ldots, & \ldots, & \ldots, & \ldots, \\
& & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, \\
\text{JA} & = & (1, & 3, & 1, & 2, & 5, & 3, & 4, & 1, & 2, & 4, & 5, & 3, \\
& & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, & \ldots, \\
\text{IPARM} & = & (1, & 3, & 1, & 1) \\
\text{RPARM} & = & (1.0-12, & 0.100)
\end{array}
\]
Call Statement and Input (Storage-By-Columns):

| IOPT N NZ A IA JA LNA IPARM RPARM OPARM AUX NAUX |
|---|---|---|---|---|---|---|---|---|---|---|
| CALL DGSF( 2 , 5, 13, A, IA, JA, 27 , IPARM, RPARAM, OPARM, AUX, 150 ) |

A

\[
(2.0, 1.0, 2.0, 1.0, 2.0, 4.0, 3.0, 1.0, 4.0, 1.0, 1.0, \\
3.0, 5.0, . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . )
\]

IA

\[
(1, 2, 4, 2, 4, 1, 3, 5, 3, 4, 5, 2, 4, . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . )
\]

JA

\[
(1, 4, 6, 9, 12, 14, . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . , . . . )
\]

IPARM

\[
(1, 3, 0, 1)
\]

RPARAM

\[
(1.D-12, 0.1D0)
\]

Output:

A

\[
(0.5, . . , 0.3, 1.0, . . , 1.0, . . , 3.0, . . , . . , . . , 1.0, \\
1.0, . . , . . , . . , -1.7, -0.5, -1.0, -1.0, \\
4.0, -3.0, -4.0)
\]

IA

\[
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, . . , . . , . . , . . , . . , . . , . . , . . , . . , . . , . . , . . , . . , . . , . . )
\]

JA

\[
(1, 0, 5, 2, 0, 4, 0, 2, 0, 0, 0, 3, 4, . . , . . , . . , . . , . . , . . , . . , . . , . . , . . , . . , . . , . . , . . , . . )
\]

OPARM

\[
(1.000000, 0.333333, 3.000000)
\]

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.

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>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL DGSS (jopt, n, a, ia, ja, lna, bx, aux, naux)</td>
</tr>
<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 not sparse.
- If \(jopt = 11\), \(A^T x = b\) is solved, where the right-hand side is sparse.

Specified as: an integer; \(jopt = 0, 1, 10, \text{ or } 11\).
\( n \) 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 49.
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 [“17” on page 1254], [“58” on page 1256], and [“114” on page 1259].

Error conditions

Computational Errors
None

Input-Argument Errors

1. \( jopt \neq 0, 1, 10, \) or 11
2. \( n < 0 \)
3. \( lna \leq 0 \)
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:

\[
\begin{bmatrix}
2.0 & 0.0 & 4.0 & 0.0 & 0.0 \\
1.0 & 1.0 & 0.0 & 0.0 & 3.0 \\
0.0 & 0.0 & 3.0 & 4.0 & 0.0 \\
2.0 & 2.0 & 0.0 & 1.0 & 5.0 \\
0.0 & 0.0 & 1.0 & 1.0 & 0.0 \\
\end{bmatrix}
\]

Call Statement and Input:

```
CALL DGSS( 0  , 5 , A , IA , JA , 27  , BX , AUX , 150 )
```

\( A \) = (0.5, . , 0.3, 1.0, . , 1.0, . , 3.0, . , . , 1.0, 
1.0, . , . , . , . , . , . , -1.7, -0.5, -1.0, -1.0, 
4.0, -3.0, -4.0)

\( IA \) = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 
 . , . , 2, 1, 1, 3, 3, 5, 5)

\( JA \) = (1, 0, 5, 2, 0, 4, 0, 2, 0, 0, 0, 3, 4, . , . , 
 . , . , 4, 2, 4, 4, 1, 3, 1)

\( BX \) = (1.0, 1.0, 1.0, 1.0, 1.0)

Linear algebraic equations subroutines 727
Output:

\[
\begin{align*}
\mathbf{IA} & = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 1, 1, 3, 3, 5, 5) \\
\mathbf{BX} & = (-5.500000, 9.500000, 3.000000, -2.000000, -1.000000)
\end{align*}
\]

**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 \( \mathbf{A}^T \mathbf{x} = \mathbf{b} \), using the same matrix \( \mathbf{A} \) used in Example 1. The input is also the same as in Example 1, except for the \( \text{jopt} \) argument. The right-hand side is not sparse.

Call Statement and Input:

```
JOPT  N   A   IA   JA   LNA   BX   AUX   NAUX
|    |   |    |    |    |     |    |     |
CALL DGSS( 1  , 5 , A , IA , JA , 27  , BX , AUX , 150 )
BX       =  (1.0, 1.0, 1.0, 1.0, 1.0)
```

Output:

\[
\begin{align*}
\mathbf{IA} & = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 1, 1, 3, 3, 5, 5) \\
\mathbf{BX} & = (0.000000, -3.000000, -2.000000, 2.000000, 7.000000)
\end{align*}
\]

**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 \( \mathbf{A} \mathbf{x} = \mathbf{b} \), using the same matrix \( \mathbf{A} \) as in Examples 1 and 2. The input is also the same as in Examples 1 and 2, except for the \( \text{jopt} \) and \( \text{bx} \) arguments. The right-hand side is sparse.

Call Statement and Input:

```
JOPT  N   A   IA   JA   LNA   BX   AUX   NAUX
|    |   |    |    |    |     |    |     |
CALL DGSS( 10 , 5 , A , IA , JA , 27  , BX , AUX , 150 )
BX       =  (0.0, 0.0, 0.0, 1.0, 0.0)
```

Output:

\[
\begin{align*}
\mathbf{IA} & = (1, 4, 2, 5, 3, 5, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) \\
\mathbf{BX} & = (0.000000, 3.000000, -2.000000, 2.000000, 7.000000)
\end{align*}
\]

**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 \( \mathbf{A}^T \mathbf{x} = \mathbf{b} \), using the same matrix \( \mathbf{A} \) as in Examples 1, 2, and 3. The input is also the same as in Examples 1, 2, and 3, except for the \( \text{jopt} \) argument. The right-hand side is sparse.

Call Statement and Input:

```
JOPT  N   A   IA   JA   LNA   BX   AUX   NAUX
|    |   |    |    |    |     |    |     |
CALL DGSS( 11 , 5 , A , IA , JA , 27  , BX , AUX , 150 )
```

728 IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and Reference
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, 2, 1, 1, 3, 3, 5, 5)
BX = (0.000000, 0.000000, 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 $A^Tx = b$ 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>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL DGKFS (n, au, nu, idu, al, nl, idl, iparm, rparm, aux, naux, bx, ldxb, mbx)</td>
</tr>
<tr>
<td>C and C++</td>
<td>dgkfs (n, au, nu, idu, al, nl, idl, iparm, rparm, aux, naux, bx, ldxb, mbx);</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 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; \( \nu \geq 0 \) 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 one of its three forms) in array \( A\text{U} \).

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 \( \text{IPARM}(3) = 0 \), then \( AL \) contains the unfactored lower triangle of general sparse matrix \( A \).
- If you are doing a **factor only**, and if \( \text{IPARM}(3) > 0 \), then \( AL \) contains the partially factored lower triangle of general sparse matrix \( A \). The first \( \text{IPARM}(3) \) 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 \( \text{IPARM}(4) = 0 \), diagonal-out skyline storage mode is used for \( A \).
- If \( \text{IPARM}(4) = 1 \), 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; \( nl \geq 0 \) 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 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, \( \text{IPARM}(i) \), where:

- \( \text{IPARM}(1) \) indicates whether certain default values for \( iparm \) and \( rparm \) are used by this subroutine, where:
  
  If \( \text{IPARM}(1) = 0 \), the following default values are used. For restrictions, see “Notes” on page 735.

  \[
  \begin{align*}
  \text{IPARM}(2) &= 0 \\
  \text{IPARM}(3) &= 0 \\
  \text{IPARM}(4) &= 0 \\
  \text{IPARM}(5) &= 0 \\
  \text{IPARM}(10) &= 0 \\
  \text{IPARM}(11) &= -1 \\
  \text{IPARM}(12) &= -1 \\
  \text{IPARM}(13) &= -1 \\
  \text{IPARM}(14) &= -1 \\
  \text{IPARM}(15) &= 0 \\
  \text{RPARM}(10) &= 10^{-12}
  \end{align*}
  \]

  If \( \text{IPARM}(1) = 1 \), the default values are not used.

- \( \text{IPARM}(2) \) indicates the type of computation performed by this subroutine. The following table gives the \( \text{IPARM}(2) \) values for each variation:
<table>
<thead>
<tr>
<th>Type of Computation</th>
<th>$Ax = b$</th>
<th>$Ax = b$ and Determinant($A$)</th>
<th>$A^T x = b$</th>
<th>$A^T x = b$ 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 735.

- **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 735.
      
      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 $a_{kk}$ for $k = 1, n$ when doing a full factorization, and they are $k = IPARM(3) + 1, n$ when doing a partial factorization. The region in which a pivot element falls depends on the sign and magnitude of the pivot element. The regions are determined by RPARM(10). For a description of the regions and associated pivot values, see “Notes” on page 735. 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 737.

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:

- IPARM(1) = 0 or 1
- IPARM(2) = 0, 1, 2, 10, 11, 100, 102, or 110
- If IPARM(2) = 0, 2, 10, 100, 102, or 110, then IPARM(3) = 0
- If IPARM(2) = 1 or 11, then 0 ≤ IPARM(3) ≤ n
- IPARM(4), IPARM(5) = 0 or 1
- If IPARM(2) = 0, 1, 10, 11, 100, or 110, then:
  - IPARM(10) = 0 or 1
  - IPARM(11), IPARM(12) = -1, 0, or 1
  - IPARM(13) = -1 or 1
  - IPARM(14), IPARM(15) = -1, 0, or 1

**rparm**

is an array of parameters, RPARM(i), where:

- RPARM(1) through RPARM(9) are reserved.
- RPARM(10) has the following meaning, where:
  - If you are doing a **factor and solve** or a **factor only**, RPARM(10) is the tolerance value for small pivots. This sets the bounds for the pivot regions, where pivots are processed according to the options you specify for the five regions in IPARM(11) through IPARM(15), respectively. The suggested value is \(10^{-15} \leq \text{IPARM}(10) \leq 1\).
  - 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 |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.
- RPARM(16) through RPARM(25), see On Return.

Specified as: a one-dimensional array of (at least) length 25, containing long-precision real numbers, where if IPARM(2) = 0, 1, 10, 11, 100, or 110, then:

- RPARM(10) ≥ 0.0
- RPARM(11) through RPARM(15) ≠ 0.0

**aux**

has the following meaning:

- If aux = 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 \textit{naux}.

Specified as: an area of storage, containing 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, 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 \textbf{factor only}:

\textbf{For 32-bit integer arguments}

Use \textit{naux} \geq 5n.

\textbf{For 64-bit integer arguments}

Use \textit{naux} \geq 7n.

If you are doing a \textbf{factor and solve} or a \textbf{solve only}:

\textbf{For 32-bit integer arguments}

Use \textit{naux} \geq 5n + 4\textit{mbx}.

\textbf{For 64-bit integer arguments}

Use \textit{naux} \geq 7n + 4\textit{mbx}.

\textit{bx}

has the following meaning, where:

If you are doing a \textbf{factor and solve} or a \textbf{solve only}, \textit{bx} is the array, containing the \textit{mbx} right-hand side vectors \textit{b} of the system \textit{Ax} = \textit{b} or \textit{A}^T\textit{x} = \textit{b}. Each vector \textit{b} is length \textit{n} and is stored in the corresponding column of the array.

If you are doing a \textbf{factor only}, this argument is not used in the computation.

Specified as: an \textit{ldbx} by (at least) \textit{mbx} array, containing long-precision real numbers.

\textit{ldbx}

has the following meaning, where:

If you are doing a \textbf{factor and solve} or a \textbf{solve only}, \textit{ldbx} is the leading dimension of the array specified for \textit{bx}.

If you are doing a \textbf{factor only}, this argument is not used in the computation.

Specified as: an integer; \textit{ldbx} \geq \textit{n} and:

If \textit{mbx} \neq 0, then \textit{ldbx} > 0.

If \textit{mbx} = 0, then \textit{ldbx} \geq 0.

\textit{mbx}

has the following meaning, where:

If you are doing a \textbf{factor and solve} or a \textbf{solve only}, \textit{mbx} is the number of right-hand side vectors, \textit{b}, in the array specified for \textit{bx}.

If you are doing a \textbf{factor only}, this argument is not used in the computation.

Specified as: an integer; \textit{mbx} \geq 0.

\textbf{On Return}

\textit{au}

is the array, referred to as \textit{AU}, containing the upper triangular part of the \textbf{LU} factored form of general sparse matrix \textit{A}, where:

If IPARM\((5) = 0\), diagonal-out skyline storage mode is used for \textit{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 ≤ 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 Aᵀx = 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 on page 736. The pivot values in each region are:

Region 1: pivot < -RPARM(10)
Region 2: -RPARM(10) ≤ pivot < 0
Region 3: pivot = 0
Region 4: $0 < \text{pivot} \leq \text{RPARM}(10)$
Region 5: $\text{pivot} > \text{RPARM}(10)$

Figure 13: Five Pivot Regions

3. The IPARM(4) and IPARM(5) arguments allow you to specify the same or different skyline storage modes for your input and output arrays for matrix $A$. This allows you to change storage modes as needed. However, if you are concerned with performance, you should use diagonal-out skyline storage mode for both input and output, if possible, because there is less overhead.

For a description of how sparse matrices are stored in skyline storage mode, see “Profile-In Skyline Storage Mode” on page 119 and “Diagonal-Out Skyline Storage Mode” on page 117.

4. Following is an illustration of the portion of matrix $A$ factored in the partial factorization when IPARM(3) $> 0$. In this case, the subroutine assumes that rows and columns 1 through IPARM(3) are already factored and that rows and columns IPARM(3) +1 through $n$ are to be factored in this computation.

You use the partial factorization function when, for design or storage reasons, you must factor the matrix $A$ in stages. When doing a partial factorization, you must use the same skyline storage mode for all parts of the matrix as it is progressively factored.

5. Your various arrays must have no common elements; otherwise, results are unpredictable.

6. You have the option of having the minimum required value for $naux$ dynamically returned to your program. For details, see “Using Auxiliary Storage in ESSL” on page 49.

Function

This subroutine can factor, compute the determinant of, and solve general sparse matrix $A$, stored in skyline storage mode. For all computations, input matrix $A$ can be stored in either diagonal-out or profile-in skyline storage mode. Output matrix $A$ can also be stored in either of these modes and can be different from the mode used for input.

Matrix $A$ is factored into the following form using specified pivot processing:

$$A = LU$$
where:

$U$ is an upper triangular matrix.
$L$ is a lower triangular matrix.

The transformed matrix $A$, factored into its $LU$ form, is stored in packed format in arrays $AU$ and $AL$. The inverse of the diagonal of matrix $U$ is stored in the corresponding elements of array $AU$. The off-diagonal elements of the upper triangular matrix $U$ are stored in the corresponding off-diagonal elements of array $AU$. The off-diagonal elements of the lower triangular matrix $L$ are stored in the corresponding off-diagonal elements of array $AL$. (The diagonal elements stored in array $AL$ do not have meaningful values.)

The partial factorization of matrix $A$, which you can do when you specify the factor-only option, assumes that the first $IPARM(3)$ rows and columns are already factored in the input matrix. It factors the remaining $n-IPARM(3)$ rows and columns in matrix $A$. (See “Notes” on page 735 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 $A^Tx = b$, 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 [“12” on page 1253], [“20” on page 1254], [“33” on page 1254], [“58” on page 1256], and [“85” on page 1257]. If $n$ is 0, no computation is performed. If $mbx$ is 0, no solve is performed.

**Error conditions**

**Resource Errors**

- Error 2015 is unrecoverable, $naux = 0$, and unable to allocate work area.
- Unable to allocate internal work area.

**Computational Errors**

1. If a pivot occurs in region $i$ for $i = 1,5$ and $IPARM(10+i) = 1$, the pivot value is replaced with $RPARM(10+i)$, an attention message is issued, and processing continues.
2. Unacceptable pivot values occurred in the factorization of matrix $A$.
   - One or more diagonal values 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 65.

**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 $IPARM(4) = 0$ for $i = 1, n$
6. $IDU(i) > IDU(i-1)+i$ and $IPARM(4) = 1$ for $i = 2, n$
7. $nl < 0$
8. \( \text{IDL}(n+1) > n+1 \)
9. \( \text{IDL}(i+1) \leq \text{IDL}(i) \) for \( i = 1, n \)
10. \( \text{IDL}(i+1) > \text{IDL}(i)+i \) and \( \text{IPARM}(4) = 0 \) for \( i = 1, n \)
11. \( \text{IDL}(i) > \text{IDL}(i-1)+i \) and \( \text{IPARM}(4) = 1 \) for \( i = 2, n \)
12. \( \text{IPARM}(1) \neq 0 \) or 1
13. \( \text{IPARM}(2) \neq 0, 1, 2, 10, 11, 100, 102, \) 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. \( \text{IPARM}(2) = 0, 1, 10, 11, 100, \) or 110 and:
   \( \text{IPARM}(10) \neq 0 \) or 1
   \( \text{IPARM}(11), \text{IPARM}(12) \neq -1, 0, \) or 1
   \( \text{IPARM}(13) \neq -1 \) or 1
   \( \text{IPARM}(14), \text{IPARM}(15) \neq -1, 0, \) or 1
   \( \text{RPARM}(10) < 0.0 \)
   \( \text{RPARM}(10+i) = 0.0 \) and \( \text{IPARM}(10+i) = 1 \) for \( i = 1, 5 \)
19. \( \text{IPARM}(2) = 0, 2, 10, 100, 102, \) or 110 and:
   \( ldbx \leq 0 \) and \( mbx \neq 0 \) and \( n \neq 0 \)
   \( ldbx < 0 \) and \( mbx = 0 \)
   \( ldbx < n \) and \( mbx \neq 0 \)
   \( mbx < 0 \)
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:

| 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 2.0 2.0 4.0 0.0 |
| 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 4.0 10.0 |
| 0.0 0.0 0.0 2.0 4.0 6.0 8.0 4.0 10.0 |
| 2.0 4.0 6.0 6.0 8.0 6.0 10.0 8.0 16.0 |

Output matrix \( A \), shown here, is in \( LU \) factored form with \( U^{-1} \) on the diagonal, and is stored in diagonal-out skyline storage mode. Matrix \( B \) is:

| 0.5 2.0 2.0 0.0 0.0 0.0 0.0 0.0 0.0 |
| 1.0 0.5 2.0 2.0 2.0 0.0 0.0 0.0 2.0 |
| 1.0 1.0 0.5 2.0 2.0 0.0 0.0 0.0 2.0 |
| 1.0 1.0 1.0 0.5 2.0 2.0 0.0 0.0 2.0 |
| 0.0 0.0 0.0 1.0 0.5 2.0 2.0 2.0 2.0 |
| 0.0 0.0 0.0 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 |
| 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 0.5 |
Call Statement and Input:

```plaintext
CALL DGKFS( 9 , AU, 33, IDU, AL, 35, IDL, IPARM, RPARM, AUX,  57 , BX , 12 , 3 )
```

| AU       | =  (2.0, 4.0, 2.0, 6.0, 4.0, 2.0, 6.0, 4.0, 2.0, 4.0, 6.0, 4.0, 2.0, 6.0, 4.0, 2.0, 6.0, 4.0, 2.0, 6.0, 4.0, 2.0, 6.0) |
| 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, 6.0, 4.0, 2.0, 0.0, 6.0, 4.0, 2.0, 0.0, 6.0, 4.0, 2.0) |
| IDL      | =  (1, 2, 4, 7, 11, 13, 18, 22, 27, 36) |
| IPARM    | =  (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36) |

Output:

| AU       | =  (0.5, 0.5, 2.0, 0.5, 2.0, 2.0, 0.5, 2.0, 2.0, 0.5, 2.0, 2.0, 0.5, 2.0, 2.0, 0.5, 2.0, 2.0, 0.5, 2.0, 2.0, 0.5, 2.0, 2.0) |
| IDU      | =  (same as input) |
| AL       | =  (0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0) |
| IDL      | =  (same as input) |
| IPARM    | =  (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36) |

```

```plaintext
| BX       | =  (not relevant) |
|          |  |
|          |  |
|          |  |
|          |  |
```

```
| BX       | =  (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 |
|          | 48.00 96.00 144.00 |
|          | 34.00 68.00 102.00 |
|          | 38.00 76.00 114.00 |
|          | 66.00 132.00 198.00 |
|          | . . . |
|          | . . . |
|          | . . . |
```

Linear algebraic equations subroutines 739
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 \( U^{-1} \) 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:

```
CALL DGKFS( 9, AU, 33, IDU, AL, 35, IDL, IPARM, RPARM, AUX, 57, BX, 12, 3 )
```

<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>RPARM</th>
<th>AUX</th>
<th>NAUX</th>
<th>BX</th>
<th>LDBX</th>
<th>MBX</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>2.0</td>
<td>2.0</td>
<td>4.0</td>
<td>2.0</td>
<td>4.0</td>
<td>6.0</td>
<td>2.0</td>
<td>4.0</td>
<td>6.0</td>
<td>2.0</td>
<td>4.0</td>
<td>6.0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>6.0</td>
<td>4.0</td>
<td>2.0</td>
<td>4.0</td>
<td>6.0</td>
<td>2.0</td>
<td>4.0</td>
<td>4.0</td>
<td>8.0</td>
<td>8.0</td>
<td>8.0</td>
<td>8.0</td>
<td>8.0</td>
</tr>
<tr>
<td></td>
<td>4.0</td>
<td>6.0</td>
<td>8.0</td>
<td>2.0</td>
<td>4.0</td>
<td>6.0</td>
<td>4.0</td>
<td>6.0</td>
<td>4.0</td>
<td>10.0</td>
<td>8.0</td>
<td>10.0</td>
<td>16.0</td>
</tr>
</tbody>
</table>

Output:

\[
\begin{pmatrix}
10.00 & 20.00 & 30.00 \\
20.00 & 40.00 & 00.00 \\
28.00 & 56.00 & 04.00 \\
30.00 & 60.00 & 09.00 \\
40.00 & 80.00 & 120.00 \\
43.00 & 88.00 & 132.00 \\
48.00 & 56.00 & 04.00 \\
60.00 & 120.00 & 180.00 \\
\end{pmatrix}
\]

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:

\[
\begin{pmatrix}
-2.0 & -2.0 & -2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
-2.0 & -4.0 & -4.0 & -2.0 & -2.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
-2.0 & -4.0 & -6.0 & -4.0 & -4.0 & 0.0 & -2.0 & 0.0 & -4.0 \\
-2.0 & -4.0 & -6.0 & -6.0 & -6.0 & -2.0 & -4.0 & 0.0 & -6.0 \\
0.0 & 0.0 & 0.0 & -2.0 & -4.0 & -4.0 & -4.0 & -2.0 & -4.0 \\
0.0 & -2.0 & -4.0 & -6.0 & -8.0 & -8.0 & -8.0 & -4.0 & -10.0 \\
0.0 & 0.0 & 0.0 & -2.0 & -4.0 & -6.0 & -8.0 & -8.0 & -8.0 \\
\end{pmatrix}
\]
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:

$$
\begin{bmatrix}
-0.5 & -2.0 & -2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & -0.5 & -2.0 & -2.0 & 0.0 & 0.0 & 0.0 & 0.0 & -2.0 \\
1.0 & 1.0 & 1.0 & -0.5 & -2.0 & -2.0 & 0.0 & 0.0 & -2.0 \\
0.0 & 0.0 & 0.0 & 1.0 & -0.5 & -2.0 & -2.0 & -2.0 & -2.0 \\
0.0 & 1.0 & 1.0 & 1.0 & -0.5 & -2.0 & -2.0 & -2.0 & -2.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 1.0 & -0.5 & -2.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & -0.5 & -2.0 \\
\end{bmatrix}
$$

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>(RPARM)</th>
<th>(AUX)</th>
<th>(NAUX)</th>
<th>(BX)</th>
<th>(LDBX)</th>
<th>(MBX)</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>AU</td>
<td>33</td>
<td>IDU</td>
<td>AL</td>
<td>35</td>
<td>IDL</td>
<td>IPARM</td>
<td>RPARM</td>
<td>AUX</td>
<td>57</td>
<td>BX</td>
<td>12</td>
<td>3</td>
</tr>
</tbody>
</table>

\begin{align*}
AU &= (-2.0, -4.0, -2.0, -6.0, -4.0, -2.0, -8.0, -6.0, -4.0) \\
IDU &= (1, 2, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, 34) \\
AL &= (0.0, 0.0, -2.0, -6.0, -4.0, -2.0, -8.0, -6.0, -4.0) \\
IDL &= (1, 2, 4, 7, 11, 13, 17, 22, 27, 34) \\
IPARM &= (1, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) \\
BX &= (same as input BX in Example 1)
\end{align*}

Output:

\begin{align*}
AU &= (-0.5, -2.0, -2.0, -0.5, -2.0, -2.0, -0.5, -2.0, -2.0, -0.5, -2.0, -2.0) \\
IDU &= (same as input) \\
AL &= (0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 0.0) \\
IDL &= (same as input) \\
IPARM &= (1, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0) \\
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, -1.00, -2.00, -3.00, -1.00, -2.00, -3.00)
\end{align*}
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 general sparse matrix $A$ from Example 1 and compute the determinant of $A_1$. Input matrix $A_1$, shown here, is stored in diagonal-out skyline storage mode. Input matrix $A_1$ is:

\[
\begin{bmatrix}
2.0 & 2.0 & 2.0 & 0.0 & 0.0 & 0.0 \\
2.0 & 4.0 & 4.0 & 2.0 & 2.0 & 0.0 \\
2.0 & 4.0 & 6.0 & 4.0 & 4.0 & 0.0 \\
2.0 & 4.0 & 6.0 & 6.0 & 6.0 & 2.0 \\
0.0 & 0.0 & 0.0 & 2.0 & 4.0 & 4.0 \\
0.0 & 2.0 & 4.0 & 6.0 & 8.0 & 6.0
\end{bmatrix}
\]

Output matrix $A_1$, shown here, is in $LU$ factored form with $U^{-1}$ on the diagonal, and is stored in diagonal-out skyline storage mode. Output matrix $A_1$ is:

\[
\begin{bmatrix}
0.5 & 2.0 & 2.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 0.5 & 2.0 & 2.0 & 2.0 & 0.0 \\
1.0 & 1.0 & 0.5 & 2.0 & 2.0 & 0.0 \\
1.0 & 1.0 & 1.0 & 0.5 & 2.0 & 2.0 \\
0.0 & 0.0 & 0.0 & 1.0 & 0.5 & 2.0 \\
0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.5
\end{bmatrix}
\]

Call Statement and Input:

\[
\begin{bmatrix}
N & AU & NU & IDU & AL & NL & IDL & IPARM & RPARAM & AUX & NAUX & BX & LDBX & MBX \\
6 & AU & 33 & IDU & AL & 35 & IDL & IPARM & RPARAM & AUX & 45 & BX & LDBX & MBX
\end{bmatrix}
\]

\[
\text{CALL DGKFS( 6, AU, 33, IDU, AL, 35, IDL, IPARM, RPARAM, AUX, 45, BX, LDBX, MBX )}
\]

\[
\begin{align*}
AU &= \text{(same as input AU in Example 1)} \\
IDU &= (1, 2, 4, 7, 10, 14, 17) \\
AL &= \text{(same as input AL in Example 1)} \\
IDL &= (1, 2, 4, 7, 11, 13, 18) \\
IPARM &= (1, 11, 0, 0, 0, \ldots, 0, \ldots, 3, \ldots, 0, 0, 0, 0) \\
RPARAM &= (\ldots, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) \\
BX &= \text{(not relevant)} \\
LDBX &= \text{(not relevant)} \\
MBX &= \text{(not relevant)}
\end{align*}
\]

Output:

\[
\begin{align*}
AU &= (0.5, 0.5, 2.0, 0.5, 2.0, 2.0, 0.5, 2.0, 2.0, 0.5, 0.5, 2.0, 2.0, 2.0) \\
IDU &= \text{(same as input)} \\
AL &= (0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0) \\
IDL &= \text{(same as input)} \\
IPARM &= (1, 11, 0, 0, 0, \ldots, 3, \ldots, 0, 0, 0, 0) \\
RPARAM &= (\ldots, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) \\
BX &= \text{(same as input)} \\
LDBX &= \text{(same as input)} \\
MBX &= \text{(same as input)}
\end{align*}
\]

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 \( \mathbf{A}_2 \), shown here, is made up of the output factored matrix \( \mathbf{A}_1 \) plus the three remaining unfactored rows and columns of matrix \( \mathbf{A} \). Matrix \( \mathbf{A}_2 \) is:

\[
\begin{bmatrix}
0.5 & 2.0 & 2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 0.5 & 2.0 & 2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 2.0 \\
1.0 & 1.0 & 0.5 & 2.0 & 2.0 & 2.0 & 4.0 & 0.0 & 6.0 \\
0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 0.5 & 2.0 & 4.0 & 2.0 \\
0.0 & 0.0 & 0.0 & 2.0 & 4.0 & 6.0 & 8.0 & 6.0 & 8.0 \\
0.0 & 0.0 & 0.0 & 2.0 & 4.0 & 6.0 & 8.0 & 8.0 & 10.0 \\
2.0 & 4.0 & 6.0 & 6.0 & 8.0 & 6.0 & 10.0 & 8.0 & 16.0 \\
\end{bmatrix}
\]

Both parts of input matrix \( \mathbf{A}_2 \) are stored in diagonal-out skyline storage mode. Output matrix \( \mathbf{A}_2 \) is the same as output matrix \( \mathbf{A} \) in Example 1 and is stored in diagonal-out skyline storage mode.

**Call Statement and Input:**

\[
\begin{array}{cccccccccccccccc}
\hline
N & AU & NU & IDU & AL & NL & IDL & IPARM & RPARM & AUX & NAUX & BX & LDBX & MBX \\
\hline
\end{array}
\]

\[
\text{CALL DGKFS}( 9, \text{AU}, 33, \text{IDU}, \text{AL}, 35, \text{IDL}, \text{IPARM}, \text{RPARM}, \text{AUX}, 45, \text{BX}, \text{LDBX}, \text{MBX})
\]

\[
\begin{align*}
\text{AU} & = \text{(same as output AU in Example 4)} \\
\text{IDU} & = \text{(same as input IDU in Example 1)} \\
\text{AL} & = \text{(same as output AL in Example 4)} \\
\text{IDL} & = \text{(same as input IDL in Example 1)} \\
\text{IPARM} & = (1, 11, 6, 0, 0, \ldots, 0, \ldots, 0, 9, \ldots, 0, 0, 0, 0, 0, 0) \\
\text{RPARM} & = \text{(not relevant)} \\
\text{BX} & = \text{(not relevant)} \\
\text{LDBX} & = \text{(not relevant)} \\
\text{MBX} & = \text{(not relevant)}
\end{align*}
\]

**Output:**

\[
\begin{align*}
\text{AU} & = \text{(same as output AU in Example 1)} \\
\text{IDU} & = \text{(same as input IDU in Example 1)} \\
\text{AL} & = \text{(same as output AL in Example 1)} \\
\text{IDL} & = \text{(same as input IDL in Example 1)} \\
\text{IPARM} & = (1, 11, 6, 0, 0, \ldots, 0, 9, 0, 0, 0, 0, 0, 0, 0, \ldots, 0, 0, 0, 0, 0, 0, 0, 0) \\
\text{RPARM} & = \text{(not relevant)} \\
\text{BX} & = \text{(same as input)} \\
\text{LDBX} & = \text{(same as input)} \\
\text{MBX} & = \text{(same as input)}
\end{align*}
\]

**Example 6**

This example shows how to solve the system \( \mathbf{A}\mathbf{x} = \mathbf{b} \) with one right-hand side for a general sparse matrix \( \mathbf{A} \). Input matrix \( \mathbf{A} \), used here, is the same as factored output matrix \( \mathbf{A} \) from Example 1, stored in profile-in skyline storage mode. Here, output matrix \( \mathbf{A} \) is unchanged on output and is stored in profile-in skyline storage mode.

**Call Statement and Input:**

\[
\begin{array}{cccccccccccccccc}
\hline
N & AU & NU & IDU & AL & NL & IDL & IPARM & RPARM & AUX & NAUX & BX & LDBX & MBX \\
\hline
\end{array}
\]

\[
\text{CALL DGKFS}( 9, \text{AU}, 33, \text{IDU}, \text{AL}, 35, \text{IDL}, \text{IPARM}, \text{RPARM}, \text{AUX}, 49, \text{BX}, 9, \text{LDBX}, \text{MBX})
\]

\[
\begin{align*}
\text{AU} & = (0.5, 2.0, 0.5, 2.0, 2.0, 0.5, 2.0, 2.0, 0.5, 2.0, \\
& 2.0, 0.5, 2.0, 2.0, 0.5, 2.0, 2.0, 2.0, 0.5, 2.0, \\
& 2.0, 2.0, 0.5, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 0.5) \\
\text{IDU} & = (1, 3, 6, 9, 13, 16, 21, 25, 33, 34) \\
\text{AL} & = (0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0)
\end{align*}
\]
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>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL DSKFS (n, a, na, idiag, iparm, rparm, aux, naux, bx, ldbx, mbx)</td>
</tr>
<tr>
<td>C and C++</td>
<td>dskfs (n, a, na, idiag, iparm, rparm, aux, naux, bx, ldbx, mbx);</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 $\text{IPARM}(3) = 0$, then $A$ contains the unfactored upper triangle of symmetric sparse matrix $A$.
  - If you are doing a **factor only**, and if $\text{IPARM}(3) > 0$, then $A$ contains the partially factored upper triangle of symmetric sparse matrix $A$. The first $\text{IPARM}(3)$ 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 $\text{IPARM}(4) = 0$, diagonal-out skyline storage mode is used for $A$.
- If $\text{IPARM}(4) = 1$, 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 $na \geq (\text{IDIAG}(n+1)-1)$.

$idiag$

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, $\text{IPARM}(i)$, where:

- $\text{IPARM}(1)$ indicates whether certain default values for $iparm$ and $rparm$ are used by this subroutine, where:

  If $\text{IPARM}(1) = 0$, the following default values are used. For restrictions, see “Notes” on page 750.

    - $\text{IPARM}(2) = 0$
    - $\text{IPARM}(3) = 0$
    - $\text{IPARM}(4) = 0$
    - $\text{IPARM}(5) = 0$
    - $\text{IPARM}(10) = 0$
    - $\text{IPARM}(11) = -1$
    - $\text{IPARM}(12) = -1$
    - $\text{IPARM}(13) = -1$
    - $\text{IPARM}(14) = -1$
    - $\text{IPARM}(15) = 0$
    - $\text{RPARM}(10) = 10^{-12}$

  If $\text{IPARM}(1) = 1$, the default values are not used.

- $\text{IPARM}(2)$ indicates the type of computation performed by this subroutine. The following table gives the $\text{IPARM}(2)$ 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>

- $\text{IPARM}(3)$ indicates whether a full or partial factorization is performed on matrix $A$, where:

  If $\text{IPARM}(3) = 0$, 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 $\text{IPARM}(3) > 0$, and you are doing a factor only, then a partial factorization is performed on matrix $A$. Rows 1 through $\text{IPARM}(3)$ of columns 1 through $\text{IPARM}(3)$ in matrix $A$ must be in factored form from a preceding call to this subroutine. The factorization is performed on rows $\text{IPARM}(3)+1$ through $n$ and columns $\text{IPARM}(3)+1$ through $n$. For an illustration, see “Notes” on page 750.

- $\text{IPARM}(4)$ indicates the input storage mode used for matrix $A$. This determines the arrangement of data in arrays $A$ and IDIAG on input, where:

  If $\text{IPARM}(4) = 0$, 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 750.
      - 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 750. 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 752.
    - 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, 2, 10, 100, 102, or 110, then IPARM(3) = 0
- If IPARM(2) = 1, 11, 101, or 111, then 0 ≤ IPARM(3) ≤ 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, RPARM($10+i$) is not used.
  If you are doing a solve only, these arguments are not used.
• RPARM(16) through RPARM(25), see On Return.
Specified as: a one-dimensional array of (at least) length 25, containing long-precision real numbers, where if IPARM(2) = 0, 1, 10, 11, 100, 101, 110, or 111, then:
RPARM(10) ≥ 0.0
If IPARM(2) = 0, 1, 10, or 11, then RPARM(11) through RPARM(15) ≠ 0.0
If IPARM(2) = 100, 101, 110, or 111, then RPARM(11) through RPARM(15) > 0.0

aux has the following meaning:
If naux = 0 and error 2015 is unrecoverable, aux is ignored.
Otherwise, it is the storage work area used by this subroutine. Its size is specified by naux.
Specified as: an area of storage, containing long-precision real numbers.

naux is the size of the work area specified by aux—that is, the number of elements in aux.
Specified as: 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 $n_{aux} \geq n$.

For 64-bit integer arguments
You can use $n_{aux} \geq 2n$.

However, for optimal performance:

For 32-bit integer arguments
Use $n_{aux} \geq 3n$.

For 64-bit integer arguments
Use $n_{aux} \geq 4n$.

If you are doing a factor and solve or a solve only:

For 32-bit integer arguments
Use $n_{aux} \geq 3n + 4\cdot m_{bx}$.

For 64-bit integer arguments
Use $n_{aux} \geq 4n + 4\cdot m_{bx}$.

For further details on error handling and the special factor-only case, see “Notes” on page 750.

$b_{x}$
has the following meaning, where:

If you are doing a factor and solve or a solve only, $b_{x}$ is the array, containing the $m_{bx}$ 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 $l_{dbx}$ by (at least) $m_{bx}$ array, containing long-precision real numbers.

$l_{dbx}$
has the following meaning, where:

If you are doing a factor and solve or a solve only, $l_{dbx}$ is the leading dimension of the array specified for $b_{x}$.

If you are doing a factor only, this argument is not used in the computation.

Specified as: an integer; $l_{dbx} \geq n$ and:
If $m_{bx} \neq 0$, then $l_{dbx} > 0$.
If $m_{bx} = 0$, then $l_{dbx} \geq 0$.

$m_{bx}$
has the following meaning, where:

If you are doing a factor and solve or a solve only, $m_{bx}$ is the number of right-hand side vectors, $b$, in the array specified for $b_{x}$.

If you are doing a factor only, this argument is not used in the computation.

Specified as: an integer; $m_{bx} \geq 0$.

On Return

$a$
is the array, referred to as $A$, containing the upper triangular part of symmetric sparse matrix $A$ in $LDL^T$ or $RTR$ 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 $m_{bx} = 0$ and you are doing a solve only, then $a$ is unchanged on output.) Returned as: a one-dimensional array of (at least) length $n_{a}$, 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:
    - IPARM(16) = -1, your factorization did not complete successfully, resulting in computational error 2126.
    - 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:
    - IPARM(20+i) = -1, your factorization did not complete successfully, resulting in computational error 2126.
    - IPARM(20+i) ≥ 0, it is the number of pivots in region $i$ for the columns that were factored in matrix $A$, where:
      - If IPARM(3) = 0, columns 1 through $n$ were factored in the full factorization.
      - If IPARM(3) > 0, columns IPARM(3)+1 through $n$ were factored in the partial factorization.
  - If you are doing a solve only, these arguments are not used in the computation and are unchanged.

Returned as: a one-dimensional array of (at least) length 25, containing 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:
    - RPARM(16) = 0.0, your factorization did not complete successfully, resulting in computational error 2126.
    - $|RPARM(16)| > 0.0$, it is the ratio for row $k$, $a_{kk}/d_{kk}$ for Gaussian elimination and $a_{kk}/r_{kk}$ for Cholesky decomposition, having the maximum absolute value. Row $k$ is indicated in IPARM(16), and:
      - If IPARM(3) = 0, the ratio corresponds to one of the rows, 1 through $n$, in the full factorization.
      - If IPARM(3) > 0, the ratio corresponds to one of the rows, IPARM(3)+1 through $n$, in the partial factorization.
If you are doing a solve only, this argument is not used in the computation and is unchanged.

- **RPARM(17)** and **RPARM(18)** have the following meaning, where:

If you are **computing the determinant** of matrix \( A \), then **RPARM(17)** is the mantissa, \( \text{detbas} \), and **RPARM(18)** is the power of 10, \( \text{detpwr} \), used to express the value of the determinant: 

\[
\text{detbas} \times 10^{\text{detpwr}}, \quad \text{where } 1 \leq \text{detbas} < 10.
\]

- 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 \( ld bx \) 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 \( |pivot| > \text{RPARM}(10) \).

3. Many of the input and output parameters for **iparm** and **rparm** are defined for the five pivot regions handled by this subroutine. The limits of the regions are based on **RPARM(10)**, as shown in **Figure 14** on page 750. The pivot values in each region are:

   - **Region 1**: \( pivot < -\text{RPARM}(10) \)
   - **Region 2**: \( -\text{RPARM}(10) \leq pivot < 0 \)
   - **Region 3**: \( pivot = 0 \)
   - **Region 4**: \( 0 < pivot \leq \text{RPARM}(10) \)
   - **Region 5**: \( pivot > \text{RPARM}(10) \)

**Figure 14: Five Pivot Regions**

4. The **IPARM(4)** and **IPARM(5)** arguments allow you to specify the same or different skyline storage modes for your input and output arrays for matrix \( A \). This allows you to change storage modes as
needed. However, if you are concerned with performance, you should use diagonal-out skyline storage mode for both input and output, if possible, because there is less overhead.

For a description of how sparse matrices are stored in skyline storage mode, see "Profile-In Skyline Storage Mode" on page 119 and "Diagonal-Out Skyline Storage Mode" on page 117. 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>A</td>
<td>AU</td>
</tr>
<tr>
<td>na</td>
<td>nu</td>
</tr>
<tr>
<td>IDIAG</td>
<td>IDU</td>
</tr>
</tbody>
</table>

5. Following is an illustration of the portion of matrix $A$ factored in the partial factorization when $\text{IPARM}(3) > 0$. In this case, the subroutine assumes that rows and columns 1 through $\text{IPARM}(3)$ are already factored and that rows and columns $\text{IPARM}(3) + 1$ through $n$ are to be factored in this computation.

\[
\begin{bmatrix}
\begin{array}{cccc}
    \ldots & \ldots & a_{1j} & a_{1,j+1} \\
    \ldots & \ldots & \ldots & \ldots \\
    \ldots & \ldots & \ldots & \ldots \\
    \ldots & \ldots & \ldots & \ldots \\
    a_{1j} & \ldots & a_{ij} & \ldots \\
    \ldots & \ldots & \ldots & \ldots \\
    a_{1,j+1} & \ldots & \ldots & \ldots \\
    \ldots & \ldots & \ldots & \ldots \\
    \ldots & \ldots & \ldots & \ldots \\
    a_{1n} & \ldots & \ldots & a_{n,j} \\
\end{array}
\end{bmatrix}
\]

where $j = \text{IPARM}(3)$

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 49.

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:  
$$A = R^T R$$  
where $R$ is an upper triangular matrix  

The transformed matrix $A$, factored into its $R^T R$ form, is stored in packed format in array $A$, such that the inverse of the diagonal elements of the upper triangular matrix $R$ is stored in the corresponding elements of array $A$. The off-diagonal elements of matrix $R$ are stored in the corresponding off-diagonal elements of array $A$.  

The partial factorization of matrix $A$, which you can do when you specify the factor-only option, assumes that the first $IPARM(3)$ rows and columns are already factored in the input matrix. It factors the remaining $n-IPARM(3)$ rows and columns in matrix $A$. (See “Notes” on page 750 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 [“12” on page 1253], [“20” on page 1254], [“33” on page 1254], [“58” on page 1256], [“85” on page 1257]. If $n$ is 0, no computation is performed. If $mbx$ is 0, no solve is performed.  

**Error conditions**  

**Resource Errors**  
- Error 2015 is unrecoverable, $naux = 0$, and unable to allocate work area.  
- Unable to allocate internal work area.  

**Computational Errors**  
1. If a pivot occurs in region $i$ for $i = 1,5$ and $IPARM(10+i) = 1$, the pivot value is replaced with $RPARM(10+i)$, an attention message is issued, and processing continues.  
2. Unacceptable pivot values occurred in the factorization of matrix $A$.  
   - One or more diagonal elements of $D$ or $R$ contains unacceptable pivots and no valid fixup is applicable. The row number $i$ of the first unacceptable pivot element is identified in the computational error message.  
   - The return code is set to 2.  
   - $i$ can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use $ERRSET$ to change the number of allowable errors for error code 2126 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see “What Can You Do about ESSL Computational Errors?” on page 65.  

**Input-Argument Errors**  
1. $n < 0$  
2. $na < 0$
3. $\text{IDIAG}(n+1) > na+1$
4. $\text{IDIAG}(i+1) \leq \text{IDIAG}(i)$ 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 $\text{IPARM}(2) \neq 1, 11, 101, \text{or } 111$
12. $\text{IPARM}(4), \text{IPARM}(5) \neq 0$ or $1$
13. $\text{IPARM}(2) = 0, 1, 10, \text{or } 11$ and:
   
   $\text{IPARM}(10) \neq 0$ or $1$
   $\text{IPARM}(11), \text{IPARM}(12) \neq -1, 0, \text{or } 1$
   $\text{IPARM}(13) \neq -1$ or $1$
   $\text{IPARM}(14), \text{IPARM}(15) \neq -1, 0, \text{or } 1$
   $\text{RPARM}(10) < 0.0$
   $\text{RPARM}(10+i) = 0.0$ and $\text{IPARM}(10+i) = 1$ for $i = 1, 5$
14. $\text{IPARM}(2) = 100, 101, 110, \text{or } 111$ and:
   
   $\text{IPARM}(10) \neq 0$ or $1$
   $\text{IPARM}(11), \text{IPARM}(12), \text{IPARM}(13) \neq -1$ or $1$
   $\text{IPARM}(14), \text{IPARM}(15) \neq -1, 0, \text{or } 1$
   $\text{RPARM}(10) < 0.0$
   $\text{RPARM}(10+i) \leq 0.0$ and $\text{IPARM}(10+i) = 1$ for $i = 1, 5$
15. $\text{IPARM}(2) = 0, 2, 10, 100, 102, \text{or } 110$ and:
   
   $l_{dbx} \leq 0$ and $m_{bx} \neq 0$ and $n \neq 0$
   $l_{dbx} < 0$ and $m_{bx} = 0$
   $l_{dbx} < n$ and $m_{bx} \neq 0$
   $m_{bx} < 0$
16. Error 2015 is recoverable or $n_{aux} \neq 0$, and $n_{aux}$ 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 $\text{IPARM}$ and $\text{RPARM}$. 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 $D^{-1}$ on the diagonal, and is stored in diagonal-out skyline storage mode. Matrix $A$ is:

\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 2.0 & 2.0 & 2.0 & 1.0 & 1.0 & 0.0 & 1.0 & 0.0 \\
1.0 & 2.0 & 3.0 & 3.0 & 2.0 & 2.0 & 0.0 & 2.0 & 0.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 3.0 & 3.0 & 0.0 & 3.0 & 0.0 \\
0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 4.0 & 1.0 & 4.0 & 0.0 \\
0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 2.0 & 5.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & 3.0 & 3.0 & 2.0 \\
0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 3.0 & 7.0 & 3.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 0.0 \\
\end{bmatrix}
\]
Call Statement and Input:

\[
\begin{array}{cccccccc}
N & A & NA & IDIAG & IPARM & RPARM & AUX & NAUX & BX & LDBX & MBX \\
\end{array}
\]

\[
\text{CALL DSKFS( 9, A, 33, IDIAG, IPARM, RPARM, AUX, 39, BX, 12, 3 )}
\]

\[
\begin{array}{cccccccccc}
A & = & (1.0, 2.0, 1.0, 3.0, 2.0, 1.0, 4.0, 3.0, 2.0, 1.0, 4.0, \\
& & 3.0, 2.0, 1.0, 5.0, 4.0, 3.0, 2.0, 1.0, 3.0, 2.0, 1.0, \\
& & 7.0, 3.0, 5.0, 4.0, 3.0, 2.0, 1.0, 4.0, 3.0, 2.0, 1.0) \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
\text{IDIAG} & = & (1, 2, 4, 7, 11, 15, 20, 23, 30, 34) \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
\text{IPARM} & = & (0, . , . , . , . , . , . , . , . , . , . , . , . , . , \\
& & . , . , . , . , . , . , . , . , . , . , . , . , . , . , . , . , . ) \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\text{RPARM} & = & (not relevant) \\
\end{array}
\]

\[
\begin{array}{cccccccc}
4.00 & 8.00 & 12.00 \\
10.00 & 20.00 & 30.00 \\
15.00 & 30.00 & 45.00 \\
19.00 & 38.00 & 57.00 \\
19.00 & 38.00 & 57.00 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\text{BX} & = & 23.00 & 46.00 & 69.00 \\
& & 11.00 & 22.00 & 33.00 \\
& & 28.00 & 56.00 & 84.00 \\
& & 10.00 & 20.00 & 30.00 \\
& & . & . & . \\
& & . & . & . \\
& & . & . & . \\
\end{array}
\]

Output:

\[
\begin{array}{cccccccccc}
\text{A} & = & (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, \\
& & 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, \\
& & 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, \\
& & 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0 \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
\text{IDIAG} & = & \text{(same as input)} \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
\text{IPARM} & = & (0, . , . , . , . , . , . , . , . , . , . , . , . , . , \\
& & . , 8, . , . , . , . , . , . , . , . , . , . , . , . ) \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
\text{RPARM} & = & ( . , . , . , . , . , . , . , . , . , . , . , . , . , . , . , \\
& & . , 7.0, . , . , . , . , . , . , . , . , . ) \\
\end{array}
\]

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:

```
CALL DSKFS( 9, A, 33, IDIAG, IPARM, RPARM, AUX, 39 , BX , 12 , 3 )
```

$A = \begin{pmatrix}
1.0 & 1.0 & 2.0 & 1.0 & 2.0 & 3.0 & 1.0 & 2.0 & 3.0 & 4.0 & 1.0 \\
2.0 & 3.0 & 4.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 1.0 & 2.0 & 3.0 \\
1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 3.0 & 7.0 & 1.0 & 2.0 & 3.0 & 4.0
\end{pmatrix}$

$\text{IDIAG} = (1, 3, 6, 10, 14, 19, 22, 29, 33, 34)$

$\text{IPARM} = (1, 10, 0, 1, 0, \ldots, 0, 0, 0, 0, 0, 0)$

$\text{RPARM} = \text{(not relevant)}$

$\text{BX} = \begin{pmatrix}
4.00 & 8.00 & 12.00 \\
10.00 & 20.00 & 30.00 \\
15.00 & 30.00 & 45.00 \\
19.00 & 38.00 & 57.00 \\
19.00 & 38.00 & 57.00 \\
23.00 & 46.00 & 69.00 \\
11.00 & 22.00 & 33.00 \\
28.00 & 56.00 & 84.00 \\
10.00 & 20.00 & 30.00 \\
\ldots & \ldots & \ldots \\
\ldots & \ldots & \ldots \\
\ldots & \ldots & \ldots
\end{pmatrix}$

Output:

$A =$ (same as output $A$ in Example 1)

$\text{IDIAG} =$ (same as input $\text{IDIAG}$ in Example 1)

$\text{IPARM} = (1, 10, 0, 1, 0, \ldots, 0, 0, \ldots, 0, 0, \ldots, 8, \ldots, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$

$\text{RPARM} = \text{(same as output RPARM in Example 1)}$

$\text{BX} =$ (same as output $\text{BX}$ in Example 1)

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{pmatrix}
-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 & 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 & 0.0 & 0.0 & -1.0 & -2.0 & -3.0 \\
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
\end{pmatrix}
```

Output matrix $A$, shown here, is in $LDL^T$ factored form with $D^{-1}$ on the diagonal, and is stored in diagonal-out skyline storage mode. Matrix $A$ is:

```
\begin{pmatrix}
-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 & 0.0 & 1.0 & 0.0
\end{pmatrix}
```
### 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:

\[
\begin{bmatrix}
  1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 \\
  0.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 \\
  0.0 & 0.0 & 1.0 & 1.0 & 0.0 & 0.0 \\
  1.0 & 2.0 & 2.0 & 1.0 & 1.0 & 1.0 \\
  0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 \\
  0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 \\
\end{bmatrix}
\]

Call Statement and Input:

```latex
\begin{verbatim}
CALL DSKFS(9, A, 33, IDIAG, IPARM, RPARM, AUX, 39, BX, 12, 3)
\end{verbatim}
```

### Output:

\[
\begin{bmatrix}
  -1.00 & -2.00 & -3.00 \\
  -1.00 & -2.00 & -3.00 \\
  -1.00 & -2.00 & -3.00 \\
  -1.00 & -2.00 & -3.00 \\
  -1.00 & -2.00 & -3.00 \\
  -1.00 & -2.00 & -3.00 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
  \cdot & \cdot & \cdot \\
  \cdot & \cdot & \cdot \\
  \cdot & \cdot & \cdot \\
\end{bmatrix}
\]

Example 4
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, ..., 0, ..., 0, ..., 0, ..., 0, 0, 0, 0, 0, 6, ...
RX = (not relevant)
BX = (not relevant)
LDBX = (not relevant)
MBX = (not relevant)

Output:

$A$ = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 3.0, 2.0, 1.0,
7.0, 3.0, 5.0, 4.0, 3.0, 2.0, 1.0, 4.0, 3.0, 2.0, 1.0)
IDIAG = (same as input)
IPARM = (1, 11, 0, 0, 0, ..., 0, ..., 0, ..., 0, ..., 0, ..., 0, 0, 0, 0, 6, ...
RX = (not relevant)
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:

$$
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 1.0 & 0.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 2.0 & 0.0 & 0.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 0.0 & 3.0 & 0.0 & 0.0 \\
0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 4.0 & 0.0 & 0.0 \\
0.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 2.0 & 5.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & 3.0 & 3.0 & 2.0 \\
0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 3.0 & 7.0 & 3.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & 3.0 & 4.0 \\
\end{bmatrix}
$$

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:
N
A
NA
IDIAG
IPARM
RPARM
AUX
NAUX BX
LDBX
MBX
|
|
|
|
|
|
|
|
|
|
|
CALL DSKFS (9 , A , 33 , IDIAG , IPARM , RPARM , AUX , 27 , BX , LDBX , MBX )

A =(same as output A in Example 4)
IDIAG =(same as input IDIAG in Example 1)
IPARM = (1, 11, 6, 0, 0, . , . , . , . , 0, . , . , . , . , . ,
. , . , . , . , . , . , . , . , . , . )
RPARM =(not relevant)
BX =(not relevant)
LDBX =(not relevant)
MBX =(not relevant)
Output:
A =(same as output A in Example 1)
IDIAG =(same as output IDIAG in Example 1)
IPARM = (1, 11, 6, 0, 0, . , . , . , . , 0, . , . , . , . , . , 8,
. , . , . , . , 0, 0, 0, 0, 3)
RPARM = ( . , . , . , . , . , . , . , . , . , . , . , . , . , . ,
. , 7.0, 1.0, 0.0, . , . , . , . , . , . , . )
BX =(same as input)
LDBX =(same as input)
MBX =(same as input)
Example 6
This example shows how to solve the system 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,
IDIAG = (1, 3, 6, 10, 14, 19,
IPARM = (1, 2, 0, 1, 1, . , .
. , . , . , . , . , . , . , .
RPARM =(not relevant)
BX = (10.0, 38.0, 64.0, 87.0,

1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0, 1.0)
22, 29, 33, 34)
, . , . , . , . , . , . , . , . ,
, . , . )
103.0, 133.0, 80.0, 174.0, 80.0)

Output:
A =(same as input)
IDIAG =(same as input)
IPARM =(same as input)
APARM =(same as input)
BX = (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0)

758 IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and
Reference


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:

\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 1.0 \\
1.0 & 5.0 & 3.0 & 0.0 & 3.0 & 0.0 & 0.0 & 0.0 & 3.0 \\
1.0 & 3.0 & 11.0 & 3.0 & 5.0 & 3.0 & 3.0 & 0.0 & 5.0 \\
0.0 & 0.0 & 3.0 & 17.0 & 5.0 & 5.0 & 0.0 & 0.0 & 5.0 \\
0.0 & 3.0 & 5.0 & 5.0 & 29.0 & 7.0 & 7.0 & 0.0 & 9.0 \\
0.0 & 0.0 & 3.0 & 5.0 & 7.0 & 39.0 & 9.0 & 6.0 & 9.0 \\
0.0 & 0.0 & 3.0 & 5.0 & 7.0 & 9.0 & 53.0 & 8.0 & 11.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 6.0 & 8.0 & 66.0 \\
1.0 & 3.0 & 5.0 & 5.0 & 9.0 & 9.0 & 11.0 & 10.0 & 89.0
\end{bmatrix}
\]

Output matrix \( A \), shown here, is in \( R' R \) factored form with the inverse of the diagonal of \( R \) on the diagonal, and is stored in profile-in-skyline storage mode. Matrix \( A \) is:

\[
\begin{bmatrix}
1.0 & 1.0 & 1.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 1.0 \\
1.0 & 0.5 & 1.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 1.0 \\
1.0 & 0.333 & 1.0 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 1.0 \\
0.0 & 0.0 & 0.25 & 1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 1.0 \\
1.0 & 1.0 & 1.0 & .2 & 1.0 & 1.0 & 0.0 & 0.0 & 1.0 \\
0.0 & 0.0 & 1.0 & 1.0 & 1.0 & .167 & 1.0 & 1.0 & 1.0 \\
0.0 & 0.0 & 1.0 & 1.0 & 1.0 & .143 & 1.0 & 1.0 & 1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & .111
\end{bmatrix}
\]

Call Statement and Input:

\[
\begin{array}{cccccccccc}
N & A & NA & IDIAG & IPARM & RPARM & AUX & NAUX & BX & LDBX & MBX \\
9 & 34 & & & & & & & & &
\end{array}
\]

\[
\text{CALL DSKFS( 9, A, 34, IDIAG, IPARM, RPARM, AUX, 43, BX, 10, 4 )}
\]

\[
\begin{bmatrix}
5.0 & 10.0 & 15.0 & 20.0 \\
15.0 & 30.0 & 45.0 & 60.0 \\
34.0 & 68.0 & 102.0 & 136.0 \\
40.0 & 80.0 & 120.0 & 160.0
\end{bmatrix}
\]

\[
\begin{bmatrix}
66.0 & 132.0 & 198.0 & 264.0 \\
78.0 & 156.0 & 234.0 & 312.0 \\
96.0 & 192.0 & 288.0 & 384.0 \\
90.0 & 180.0 & 270.0 & 360.0 \\
142.0 & 284.0 & 426.0 & 568.0
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
1.0 & 1.0 & .5 & 1.0 & 1.0 & .333 & 1.0 & .25 & 1.0 \\
1.0 & 0.2 & 1.0 & 1.0 & 1.0 & .167 & 1.0 & .125 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & .111 \\
\end{bmatrix}
\]

Linear algebraic equations subroutines 759
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>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL DSRIS (stor, init, n, ar, ja, ia, b, x, iparm, rparm, aux1, naux1, aux2, naux2)</td>
</tr>
<tr>
<td>C and C++</td>
<td>dsris (stor, init, n, ar, ja, ia, b, x, iparm, rparm, aux1, naux1, aux2, 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 $init = 'I'$, the preconditioning matrix is computed, the internal representation of the sparse matrix is generated, and the iteration procedure is performed. The coefficient matrix and preconditioner in internal format are saved in $aux1$. 
If \( \text{init} = 'S' \), 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 \( \text{init} = 'I' \). You use this option to solve the same matrix for different right-hand sides, \( \mathbf{b} \), 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 \( \text{init} = 'S' \).

Specified as: a single character. It must be 'I' or 'S'.

\( n \)

is the order of the linear system \( \mathbf{A} \mathbf{x} = \mathbf{b} \) and the number of rows and columns in sparse matrix \( \mathbf{A} \).

Specified as: an integer; \( n \geq 0 \).

\( \mathbf{a}r \)

is the sparse matrix \( \mathbf{A} \) of order \( n \), stored by rows in an array, referred to as \( \mathbf{AR} \). The \( \text{stor} \) argument indicates the storage variation used for storing matrix \( \mathbf{A} \).

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 \( \text{IA}(n+1) \).

\( \mathbf{j}a \)

is the array, referred to as \( \mathbf{JA} \), containing the column numbers of each nonzero element in sparse matrix \( \mathbf{A} \).

Specified as: a one-dimensional array, containing integers; \( 1 \leq (\mathbf{JA} \text{ elements}) \leq n \). The number of elements in this array can be determined by subtracting 1 from the value in \( \text{IA}(n+1) \).

\( \mathbf{i}a \)

is the row pointer array, referred to as \( \mathbf{IA} \), containing the starting positions of each row of matrix \( \mathbf{A} \) in array \( \mathbf{AR} \) and one position past the end of array \( \mathbf{AR} \). Specified as: a one-dimensional array of (at least) length \( n+1 \), containing integers; \( \text{IA}(i+1) \geq \text{IA}(i) \) for \( i = 1, n+1 \).

\( \mathbf{b} \)

is the vector \( \mathbf{b} \) 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.

\( \mathbf{x} \)

is the vector \( \mathbf{x} \) 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.

\( \text{iparm} \)

is an array of parameters, \( \text{IPARM}(i) \), where:

- \( \text{IPARM}(1) \) controls the number of iterations.
  - If \( \text{IPARM}(1) > 0 \), \( \text{IPARM}(1) \) is the maximum number of iterations allowed.
  - If \( \text{IPARM}(1) = 0 \), the following default values are used:
    - \( \text{IPARM}(1) = 300 \)
    - \( \text{IPARM}(2) = 4 \)
    - \( \text{IPARM}(4) = 4 \)
    - \( \text{IPARM}(5) = 1 \)
    - \( \text{RPARM}(1) = 10^{-6} \)
    - \( \text{RPARM}(2) = 1 \)
- \( \text{IPARM}(2) \) is the flag used to select the iterative procedure used in this subroutine.
  - If \( \text{IPARM}(2) = 1 \), the conjugate gradient (CG) method is used. Note that this algorithm should only be used with positive definite symmetric matrices.
  - If \( \text{IPARM}(2) = 2 \), the conjugate gradient squared (CGS) method is used.
  - If \( \text{IPARM}(2) = 3 \), the generalized minimum residual (GMRES) method, restarted after \( k \) steps, is used.
If $\text{IPARM}(2) = 4$, the more smoothly converging variant of the CGS method (Bi-CGSTAB) is used.

If $\text{IPARM}(2) = 5$, the transpose-free quasi-minimal residual method (TFQMR) is used.

- $\text{IPARM}(3)$ has the following meaning, where:
  
  If $\text{IPARM}(2) \neq 3$, then $\text{IPARM}(3)$ is not used.
  
  If $\text{IPARM}(2) = 3$, then $\text{IPARM}(3) = k$, 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.

- $\text{IPARM}(4)$ is the flag that determines the type of preconditioning.
  
  If $\text{IPARM}(4) = 1$, the system is not preconditioned.
  
  If $\text{IPARM}(4) = 2$, the system is preconditioned by a diagonal matrix.
  
  If $\text{IPARM}(4) = 3$, the system is preconditioned by SSOR splitting with the diagonal given by the absolute values sum of the input matrix.
  
  If $\text{IPARM}(4) = 4$, the system is preconditioned by an incomplete LU factorization.
  
  If $\text{IPARM}(4) = 5$, 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 $\text{IPARM}(4) = 1$ or 2.

- $\text{IPARM}(5)$ 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:
  
  - $\epsilon$ is the desired relative accuracy and is stored in $\text{RPARM}(1)$.
  
  - $x_j$ is the solution found at the $j$-th iteration.
  
  - $r_j$ and $r_0$ are the preconditioned residuals obtained at iterations $j$ and 0, respectively. (The residual at iteration $j$ is given by $b - Ax_j$.)

  If $\text{IPARM}(5) = 1$, the iterative method is stopped when:
  
  $\|r_j\|_2 / \|x_j\|_2 < \epsilon$

  **Note:** $\text{IPARM}(5) = 1$ 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 $\text{IPARM}(5)$ value, you, by default, use the above stopping criterion.

  If $\text{IPARM}(5) = 2$, the iterative method is stopped when:
  
  $\|r_j\|_2 / \|r_0\|_2 < \epsilon$

  If $\text{IPARM}(5) = 3$, the iterative method is stopped when:
  
  $\|x_j - x_{j-1}\|_2 / \|x_j\|_2 < \epsilon$

  **Note:** Stopping criterion 3 performs poorly with the TFQMR method; therefore, if you specify TFQMR ($\text{IPARM}(2) = 5$), you should not specify stopping criterion 3.

- $\text{IPARM}(6)$, see On Return.

Specified as: an array of (at least) length 6, containing integers, where:

- $\text{IPARM}(1) \geq 0$
- $\text{IPARM}(2) = 1, 2, 3, 4, \text{ or } 5$
- If $\text{IPARM}(2) = 3$, then $\text{IPARM}(3) > 0$
- $\text{IPARM}(4) = 1, 2, 3, 4, \text{ or } 5$
- $\text{IPARM}(5) = 1, 2, \text{ or } 3$ (Other values default to stopping criterion 1.)

$rparm$ is an array of parameters, $\text{RPARM}(\cdot)$, where:

$\text{RPARM}(1)$ has the following meaning, where:
• if RPARM(1) > 0, then RPARM(1) is the relative accuracy $\varepsilon$ 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 765.

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 = 'I', the working storage is computed. It can contain any values.
If init = 'S', the working storage is used in solving the linear system. It contains the coefficient matrix and preconditioner in internal format, computed in an earlier call to this subroutine.

Specified as: an area of storage, containing naux1 long-precision real numbers.

naux1
is the number of doublewords in the working storage specified in aux1.

Specified as: an integer, where:
In these formulas nw has the following value:
If stor = 'G', then nw = IA(n+1)-1+n.
If stor = 'U' or 'L', then nw = 2(IA(n+1)-1).

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 ≥ 3nw + n + 20.

For 64-bit integer arguments
If IPARM(4) = 1, use naux1 ≥ 2nw + n + 40.
If IPARM(4) = 2, use naux1 ≥ 2nw + 2n + 40.
If IPARM(4) = 3, 4, or 5, use naux1 ≥ 4nw + 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 \( \text{naux2} = 0 \) 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 \( \text{IPARM}(2) = 1 \), use \( \text{naux2} \geq 4n \).
- If \( \text{IPARM}(2) = 2 \), use \( \text{naux2} \geq 7n \).
- If \( \text{IPARM}(2) = 3 \), use \( \text{naux2} \geq (k+2)n+k(k+4)+1 \), where \( k = \text{IPARM}(3) \).
- If \( \text{IPARM}(2) = 4 \), use \( \text{naux2} \geq 7n \).
- If \( \text{IPARM}(2) = 5 \), use \( \text{naux2} \geq 9n \).

On Return

- \( \text{ar} \) is the sparse matrix \( A \) of order \( n \), stored by rows in an array, referred to as \( \text{AR} \). The \( \text{stor} \) argument indicates the storage variation used for storing matrix \( A \). The order of the elements in each row of \( A \) in \( \text{AR} \) 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 \( \text{IA}(n+1) \).

- \( \text{ja} \) is the array, referred to as \( \text{JA} \), containing the column numbers of each nonzero element in sparse matrix \( A \). These elements correspond to the arrangement of the contents of \( \text{AR} \) on output.
  
  Returned as: a one-dimensional array, containing integers; \( 1 \leq (\text{JA} \text{ elements}) \leq n \). The number of elements in this array can be determined by subtracting 1 from the value in \( \text{IA}(n+1) \).

- \( \text{x} \) 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.

- \( \text{iparm} \) is an array of parameters, \( \text{IPARM}(i) \), where:
  
  - \( \text{IPARM}(1) \) through \( \text{IPARM}(5) \) are unchanged.
  - \( \text{IPARM}(6) \) contains the number of iterations performed by this subroutine.
  
  Returned as: a one-dimensional array of length 6, containing integers.

- \( \text{rparm} \) is an array of parameters, \( \text{RPARM}(i) \), where:
  
  - \( \text{RPARM}(1) \) is unchanged.
  - \( \text{RPARM}(2) \) contains the estimate of the error of the solution. If the process converged, \( \text{RPARM}(2) \leq \varepsilon \).
  - \( \text{RPARM}(3) \) is unchanged.
  
  Returned as: a one-dimensional array of length 3, containing long-precision real numbers.

- \( \text{aux1} \) 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 \( \text{naux1} \) long-precision real numbers.

Notes

1. If you want to solve the same sparse linear system of equations multiple times using a different algorithm with the same preconditioner and using a different right-hand side each time, you get the best performance by using the following technique. Call DSRIS the first time with \( \text{init} = 'I' \). This solves the system, and then stores the coefficient matrix and preconditioner in internal format in \( \text{aux1} \). On the subsequent invocations of DSRIS with different right-hand sides, specify \( \text{init} = 'S' \). 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 $\varepsilon$ (in RPARM(1)) must be specified reasonably ($10^{-4}$ to $10^{-8}$). 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.

6. For a description of how sparse matrices are stored by rows, see “Storage-by-Rows” on page 115.

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 49.

**Function**

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, 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 [“46” on page 1255], [“69” on page 1257], [“101” on page 1258], [“107” on page 1258], [“110” on page 1259], and [“116” on page 1259].

When you call this subroutine to solve a system for the first time, you specify init = 'I'. After that, you can solve the same system any number of times by calling this subroutine each time with init = 'S'. These subsequent calls use the coefficient matrix and preconditioner, stored in internal format in aux1. You
optimize performance by doing this, because certain portions of the computation have already been performed.

**Error conditions**

**Resource Errors**
Error 2015 is unrecoverable, \( \text{naux2} = 0 \), 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 65.

- For error 2110, if \( \text{RPARM}(1) > 0 \), 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 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. \( \text{IPARM}(2) = 1, 2, 3, 4, \text{or} 5 \)
8. \( \text{IPARM}(3) \leq 0 \text{ and} \text{IPARM}(2) = 3 \)
9. \( \text{IPARM}(4) = 1, 2, 3, 4, \text{or} 5 \)
10. \( \text{RPARM}(1) < 0 \)
11. \( \text{RPARM}(3) \leq 0 \text{ and} \text{IPARM}(4) = 3 \)
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 RPARM(1). The algorithm is allowed to perform 20 iterations. The process converges after 9 iterations. Matrix $A$ is:

\[
\begin{bmatrix}
2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 0.0 & 0.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 1.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & -1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & -1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & -1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 \\
\end{bmatrix}
\]

Call Statement and Input:

```
STOR INIT N AR JA IA B X IPARM RPARM AUX1 NAUX1 AUX2
|     |    |   |    |   |   |   |   |     |       |      |       |      |       |
CALL DSRIS( 'G' , 'I' , 9 , AR , JA , IA , B , X , IPARM , RPARM , AUX1 , 98 , AUX2 , 63 )
```

AR = (2.0, 2.0, -1.0, 1.0, 2.0, 1.0, 2.0, -1.0, 1.0, 2.0, -1.0, 1.0, 2.0, -1.0, 1.0, 2.0, -1.0, 1.0, 2.0)
JA = (1, 2, 3, 2, 3, 1, 4, 5, 4, 5, 6, 5, 6, 7, 6, 7, 8, 7, 8, 9, 8, 9)
IA = (1, 2, 4, 6, 9, 12, 15, 18, 21, 23)
B = (2.0, 1.0, 3.0, 2.0, 2.0, 2.0, 2.0, 2.0, 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, 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) = 4
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, 1.0, 1.0)
IPARM(6) = 9
RPARM(2) = 0.290-16
```

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:

```
STOR INIT N AR JA IA B X IPARM RPARM AUX1 NAUX1 AUX2
|     |    |   |    |   |   |   |   |     |       |      |       |      |       |
CALL DSRIS( 'G' , 'S' , 9 , AR , JA , IA , B , X , IPARM , RPARM , AUX1 , 98 , AUX2 , 63 )
```

AR = (same as input AR in Example 1)
JA = (same as input JA in Example 1)
IA = (same as input IA in Example 1)
B = (same as input B in Example 1)
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 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:

$$
\begin{pmatrix}
2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
-1.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & -1.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & -1.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & -1.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & -1.0 & 0.0 & 2.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -1.0 & 0.0 & 2.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -1.0 & 0.0 & 2.0 \\
\end{pmatrix}
$$

Example 2

This example finds the solution of the linear system $Ax = b$ for the same sparse matrix $A$ used in Example 1. 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 768
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**

```
| NAUX2 | STOR INIT N AR JA IA B X IPARM RPARM AUX1 NAUX1 AUX2 |
|-------|-----------------|------|-----|-----|------|------|------|------|------|------|
|       | CALL DSRIS( 'U', 'I', 9 , AR , JA , IA, B , X , IPARM , RPARM , AUX1 , 219 , AUX2 , 109 ) |
```

```
AR   = (2.0, -1.0, 2.0, -1.0, 2.0, -1.0, 2.0, -1.0, 2.0, -1.0, 2.0, -1.0, 2.0, -1.0, 2.0, 2.0)
JA   = (1, 3, 2, 4, 3, 5, 4, 6, 5, 7, 6, 8, 7, 9, 8, 9)
IA   = (1, 3, 5, 7, 9, 11, 13, 15, 16, 17)
B    = (1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0)
X    = (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
IPARM(1) = 20
IPARM(2) = 3
IPARM(3) = 5
IPARM(4) = 3
IPARM(5) = 1
RPARM(1) = 1.0-7
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) = 0.330-7
```

---

**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 115, 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 1212

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fortran</strong></td>
<td>CALL DSMCG (m, nz, ac, ka, lda, b, x, iparm, rparm, aux1, naux1, aux2, naux2)</td>
</tr>
<tr>
<td><strong>C and C++</strong></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 $A$.  

---

**Linear algebraic equations subroutines 769**
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 $1 \leq$ (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 \( \text{RPARM}(2) \) 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 \( \text{aux2} \).

- **IPARM(4)**, see On Return.

Specified as: an array of (at least) length 4, containing integers, where:

\[
\begin{align*}
\text{IPARM}(1) & \geq 0 \\
\text{IPARM}(2) & = 0, 1, \text{ or } 2 \\
\text{IPARM}(3) & = 0, 10, \text{ or } -10
\end{align*}
\]

\( \text{rparm} \) is an array of parameters, \( \text{RPARM}(i) \), where \( \varepsilon \) is stored in \( \text{RPARM}(1) \), and \( \lambda \) is stored in \( \text{RPARM}(2) \).

\( \text{RPARM}(1) > 0 \), is the relative accuracy \( \varepsilon \) used in the stopping criterion.

\( \text{RPARM}(2) > 0 \), is the estimate of the smallest eigenvalue, \( \lambda \), of the iteration matrix. It is only used when \( \text{IPARM}(2) = 2 \).

\( \text{RPARM}(3) \), see On Return.

Specified as: a one-dimensional array of (at least) length 3, containing long-precision real numbers.

\( \text{aux1} \) has the following meaning:

If \( \text{naux1} = 0 \) and error 2015 is unrecoverable, \( \text{aux1} \) 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 \( \text{naux1} \).

Specified as: an area of storage, containing long-precision real numbers.

\( \text{naux1} \) is the size of the work area specified by \( \text{aux1} \)—that is, the number of elements in \( \text{aux1} \).

Specified as: an integer, where:

If \( \text{naux1} = 0 \) 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, \( \text{naux1} \) must have at least the following value, where:

If \( \text{IPARM}(2) = 0 \) or 2, use \( \text{naux1} \geq 3m \).

If \( \text{IPARM}(2) = 1 \) and \( \text{IPARM}(1) \neq 0 \), use \( \text{naux1} \geq 3m+2(\text{IPARM}(1)) \).

If \( \text{IPARM}(2) = 1 \) and \( \text{IPARM}(1) = 0 \), use \( \text{naux1} \geq 3m+600 \).

\( \text{aux2} \) is a storage work area used by this subroutine. If \( \text{IPARM}(3) = -10 \), \( \text{aux2} \) must contain the incomplete Cholesky factorization of matrix \( A \), computed in an earlier call to DSMCG. The size of \( \text{aux2} \) is specified by \( \text{naux2} \).

Specified as: an area of storage, containing long-precision real numbers.

\( \text{naux2} \) is the size of the work area specified by \( \text{aux2} \)—that is, the number of elements in \( \text{aux2} \).

Specified as: an integer. When \( \text{IPARM}(3) = 10 \) or -10, \( \text{naux2} \) must have at least the following value:
For 32-bit integer arguments
\[ naux2 \geq m(nz-1)1.5+2(m+6). \]

For 64-bit integer arguments
\[ naux2 \geq m(nz-1)2.0+3(m+6). \]

On Return

\( x \)

is the vector \( x \) of length \( m \), containing the solution of the system \( Ax = b \). Returned as: a one-dimensional array of (at least) length \( m \), containing long-precision real numbers.

\( iparm \)

is an array of parameters, IPARM(\( i \)), where:

- IPARM(1) is unchanged.
- IPARM(2) is unchanged.
- IPARM(3) is unchanged.
- IPARM(4) contains the number of iterations performed by this subroutine.

Returned as: a one-dimensional array of length 4, containing integers.

\( rparm \)

is an array of parameters, RPARAM(\( i \)), where:

- RPARAM(1) is unchanged.
- RPARAM(2) is unchanged if IPARM(2) = 0 or 2. If IPARM(2) = 1, RPARAM(2) contains \( \lambda \), an estimate of the smallest eigenvalue of the iteration matrix.
- RPARAM(3) contains the estimate of the error of the solution. If the process converged, RPARAM(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” on page 772 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:

\[ \| b - Ax \|_2 / \lambda \| x \|_2 < \varepsilon \]
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 110.

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 49.

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 [“75” on page 1257] and [“82” on page 1257]. [“46” on page 1255].

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 1212).

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 65.

- 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) ≠ 0, 1, or 2
8. IPARM(3) ≠ 0, 10, or -10
9. RPARM(1) < 0
10. RPARM(2) < 0
11. Error 2015 is recoverable or \(naux1\neq0\), 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:

\[
\begin{bmatrix}
2.0 & 0.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & -1.0 & 2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
-1.0 & 0.0 & 0.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & -1.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & -1.0 & 2.0 & -1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 2.0 & -1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 2.0 & -1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -1.0 & 2.0 & 0.0 \\
\end{bmatrix}
\]

Note: For input matrix KA, ( . ) indicates any value between 1 and 9.

Call Statement and Input:

```call
call dsmcg( 9, 3, ac, ka, 9, b, x, iparm, rparm, aux1, 27, aux2, 0 )
```

IPARM(1) = 20
IPARM(2) = 0
IPARM(3) = 0
RPARM(1) = 1.D-7

\[
\begin{bmatrix}
2.0 & -1.0 & 0.0 \\
2.0 & -1.0 & 0.0 \\
-1.0 & 2.0 & 0.0 \\
-1.0 & 2.0 & 0.0 \\
\end{bmatrix}
\]
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 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:

```call
CALL DSMCG( 9 , 3 , AC, KA, 9 , B , X, IPARM, RPARM, AUX1, 67  , AUX2, 74  )
```

**Output:**

- **IPARM(4) = 1**
- **RPARM(2) = 1**
- **RPARM(3) = 0.100D-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:

```call
CALL DSMCG( 9 , 3 , AC, KA, 9 , B , X, IPARM, RPARM, AUX1, 67  , AUX2, 74  )
```

**Output:**

- **IPARM(4) = 1**
- **RPARM(2) = 1**
- **RPARM(3) = 0.100D-15**
DSDCG (Sparse Positive Definite or Negative Definite Symmetric Matrix Iterative Solve Using Compressed-Diagonal Storage Mode)

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>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL DSDCG ($iopt$, $m$, $nd$, $ad$, $lda$, $la$, $b$, $x$, $iparm$, $rparm$, aux1, aux2, aux3, aux4)</td>
</tr>
<tr>
<td>C and C++</td>
<td>dsdcg ($iopt$, $m$, $nd$, $ad$, $lda$, $la$, $b$, $x$, $iparm$, $rparm$, aux1, aux2, aux3, aux4);</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 $AD$.

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 $AD$, the number of columns in the array $AD$, 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 $AD$, containing the values of the nonzero elements of the sparse matrix stored in compressed-diagonal storage mode. If $iopt = 1$, the main diagonal and one of each pair of identical diagonals is stored in this array.

Specified as: an $lda$ by (at least) $nd$ array, containing long-precision real numbers.

$lda$
is the leading dimension of the array specified for $ad$.

Specified as: 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 111.
Specified as: a one-dimensional array of (at least) length \( nd \), containing integers, where \( 1 \leq m \leq (\text{elements of } LA) \leq m-1 \).

\( b \) 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.

\( \text{iparm} \)

is an array of parameters, \( IPARM(i) \), where:

- \( IPARM(1) \) controls the number of iterations.
  
  \( \text{If } IPARM(1) > 0, \text{ IPARM}(1) \text{ is the maximum number of iterations allowed.} \)
  
  \( \text{If } IPARM(1) = 0, \text{ the following default values are used:} \)
  
  \( IPARM(1) = 300 \)
  \( IPARM(2) = 1 \)
  \( IPARM(3) = 0 \)
  \( RPARM(1) = 10^{-6} \)

- \( IPARM(2) \) is the flag used to select the stopping criterion.
  
  \( \text{If } IPARM(2) = 0, \text{ the conjugate gradient iterative procedure is stopped when:} \)
  
  \[ \| r \|_2 / \| x \|_2 < \varepsilon \]
  
  where \( r = b - Ax \) is the residual and \( \varepsilon \) is the desired relative accuracy. \( \varepsilon \) is stored in \( RPARM(1) \).
  
  \( \text{If } IPARM(2) = 1, \text{ the conjugate gradient iterative procedure is stopped when:} \)
  
  \[ \| r \|_2 / \| x \|_2 < \varepsilon \]
  
  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 \( RPARM(2) \).
  
  \( \text{If } IPARM(2) = 2, \text{ the conjugate gradient iterative procedure is stopped when:} \)
  
  \[ \| r \|_2 / \| x \|_2 < \varepsilon \]
  
  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.
  
  \( \text{If } IPARM(3) = 0, \text{ the system is not preconditioned.} \)
  
  \( \text{If } IPARM(3) = 10, \text{ the system is preconditioned by an incomplete Cholesky factorization.} \)
  
  \( \text{If } IPARM(3) = -10, \text{ the system is preconditioned by an incomplete Cholesky factorization, where the factorization matrix was computed in an earlier call to this subroutine and is stored in aux2.} \)

- \( IPARM(4) \), see \text{On Return}. 

Specified as: an array of (at least) length 4, containing integers, where:

\( IPARM(1) = 0 \)
\( IPARM(2) = 0, 1, \text{ or } 2 \)
\( IPARM(3) = 0, 10, \text{ or } -10 \)
**rparm**
is an array of parameters, **RPARM(i)**, where $\varepsilon$ is stored in **RPARM(1)**, and $\lambda$ is stored in **RPARM(2)**.

**RPARM(1)** > 0, is the relative accuracy $\varepsilon$ used in the stopping criterion.

**RPARM(2)** > 0, is the estimate of the smallest eigenvalue, $\lambda$, 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 $\geq 3m$.
- If IPARM(2) = 1 and IPARM(1) $\neq 0$, use naux1 $\geq 3m + 2(IPARM(1))$.
- If IPARM(2) = 1 and IPARM(1) = 0, use naux1 $\geq 3m + 600$.

**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**
  naux2 $\geq m(3nd + 2) + 8$

- **For 64-bit integer arguments**
  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) ≤ $\varepsilon$.

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” on page 779 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 110.
6. On output, array AD and vector $b$ are not bitwise identical to what they were on input, because the matrix $A$ and the right-hand side are scaled before starting the iterative process and are unscaled before returning control to the user. In addition, arrays AD and LA may be rearranged on output, but still contain a mathematically equivalent mapping of the elements in matrix $A$.
7. You have the option of having the minimum required value for naux dynamically returned to your program. For details, see “Using Auxiliary Storage in ESSL” on page 49.

### 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 \( AD \) and \( LA \).
- **x** is a vector of length \( m \).
- **b** is a vector of length \( m \).

The system is solved using the two-term conjugate gradient method, with or without preconditioning by an incomplete Cholesky factorization. In both cases, the matrix is scaled by the square root of the diagonal.

See references [“75” on page 1257] and [“82” on page 1257]. [“46” on page 1255].

### 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 65.

- For error 2110, return code 1 indicates that the subroutine exceeded \( IPARM(1) \) iterations without converging. Vector \( x \) contains the approximate solution computed at the last iteration.
- For error 2111, return code 2 indicates that \( aux2 \) contains an incorrect factorization. The subroutine has been called with \( IPARM(3) = -10 \), and \( aux2 \) contains an incomplete factorization of the input matrix \( A \) that was computed by a previous call to the subroutine when \( IPARM(3) = 10 \). This error indicates that \( aux2 \) has been modified since the last call to the subroutine, or that the input matrix is not the same as the one that was factored. If the default action has been overridden, the subroutine can be called again with the same parameters, with the exception of \( IPARM(3) \) = 0 or 10.
- For error 2109, return code 3 indicates that the inner product \( (y, Ay) \) is negative in the iterative procedure after iteration \( i \). This should not occur, because the input matrix is assumed to be positive or negative definite. Vector \( x \) contains the results of the last iteration. The value \( i \) is identified in the computational error message.
- For error 2108, return code 4 indicates that the matrix is not positive definite. AC is partially modified and does not represent the same matrix as on entry.

#### Input-Argument Errors

1. \( iopt \neq 0 \) 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. \( IPARM(2) \neq 0, 1, \) or 2
10. \( IPARM(3) \neq 0, 10, \) or -10
11. \( RPARM(1) < 0 \)
12. $\text{RPARM}(2) < 0$

13. 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.

14. $\text{naux2}$ 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 $\mathbf{Ax} = \mathbf{b}$ for 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 two-term conjugate gradient method. In this example, $\text{IOPT} = 0$. Matrix $\mathbf{A}$ is:

$$
\begin{pmatrix}
2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
-1.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & -1.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & -1.0 & 0.0 & 2.0 & 0.0 & -1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 2.0 & 0.0 & 2.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -1.0 & 0.0 & 2.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -1.0 & 0.0 & 2.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -1.0 & 0.0 & 2.0 & 0.0 \\
\end{pmatrix}
$$

Call Statement and Input:

```plaintext
CALL DSDCG( 0 , 9 , 3 , AD , 9 , LA , B , X , IPARM , RPARM , AUX1 , 283 , AUX2 , 0 )
```

- $\text{IPARM}(1) = 20$
- $\text{IPARM}(2) = 0$
- $\text{IPARM}(3) = 0$
- $\text{RPARM}(1) = 1.0D-7$

**AD**

- $\begin{pmatrix}
2.0 & 0.0 & -1.0 \\
2.0 & 0.0 & -1.0 \\
2.0 & -1.0 & -1.0 \\
2.0 & -1.0 & -1.0 \\
2.0 & -1.0 & -1.0 \\
2.0 & -1.0 & -1.0 \\
2.0 & -1.0 & 0.0 \\
2.0 & -1.0 & 0.0 \\
\end{pmatrix}$

**LA**

- $\begin{pmatrix}
0.0 , -2.0 , 2 \\
1.0 , 1.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 1.0 , 1.0 \\
\end{pmatrix}$

**X**

- $\begin{pmatrix}
0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 \\
\end{pmatrix}$

Output:

- $\begin{pmatrix}
1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 \\
\end{pmatrix}$
- $\text{IPARM}(4) = 5$
- $\text{RPARM}(2) = 0$
- $\text{RPARM}(3) = 0.46D-16$

**Example 2**

This example finds the solution of the linear system $\mathbf{Ax} = \mathbf{b}$ 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 two-term conjugate gradient method. In this example, $\text{IOPT} = 1$, indicating that the matrix is symmetric, and only the main diagonal and one of each pair of identical diagonals are stored in array $\mathbf{AD}$. 
Call Statement and Input:

<table>
<thead>
<tr>
<th>IOPT</th>
<th>M</th>
<th>ND</th>
<th>AD</th>
<th>LDA</th>
<th>LA</th>
<th>B</th>
<th>X</th>
<th>IPARM</th>
<th>RPARM</th>
<th>AUX1</th>
<th>NAUX1</th>
<th>AUX2</th>
<th>NAUX2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>9</td>
<td>2</td>
<td>AD</td>
<td>9</td>
<td>LA</td>
<td>B</td>
<td>X</td>
<td>IPARM</td>
<td>RPARM</td>
<td>AUX1</td>
<td>283</td>
<td>AUX2</td>
</tr>
<tr>
<td>CALL DSGC( 1 , 9 , 2 , AD , 9 , LA , B , X , IPARM , RPARM , AUX1 , 283 , AUX2 , 80 )</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>

IPARM(1) = 20
IPARM(2) = 0
IPARM(3) = 10
RPARM(1) = 1.D-7

<table>
<thead>
<tr>
<th>2.0</th>
<th>0.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2.0</td>
<td>-1.0</td>
</tr>
<tr>
<td>2.0</td>
<td>-1.0</td>
</tr>
</tbody>
</table>

AD =

<table>
<thead>
<tr>
<th>2.0</th>
<th>0.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2.0</td>
<td>-1.0</td>
</tr>
<tr>
<td>2.0</td>
<td>-1.0</td>
</tr>
<tr>
<td>2.0</td>
<td>-1.0</td>
</tr>
<tr>
<td>2.0</td>
<td>-1.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0.0</th>
<th>0.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

LA = (0, -2)
B = (1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0)
X = (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)

Output:

| X     | 1.0   | 1.0   | 1.0   | 1.0   | 1.0   | 1.0   | 1.0   | 1.0   |
|       | (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0) |
| IPARM(4) | 1     |
| RPARM(2)  | 0     |
| RPARM(3)  | 0.89D-16 |

DSMGCG (General Sparse Matrix Iterative Solve Using Compressed-Matrix Storage Mode)

**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 115, 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 1212.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL DSMGCG (m, nz, ac, ka, lda, b, x, iparm, rparm, aux1, naux1, aux2, naux2)</td>
</tr>
<tr>
<td>C and C++</td>
<td>dsmgcg (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 $1 \leq$ (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:
  $IPARM(1) = 300$
  $IPARM(2) = 0$
  $IPARM(3) = 10$
  $RPARM(1) = 10^{-6}$
- $IPARM(2)$ is the flag used to select the iterative procedure used in this subroutine.
  If $IPARM(2) = 0$, the conjugate gradient squared method is used.
  If $IPARM(2) = k$, the generalized minimum residual method, restarted after $k$ steps, is used. Note
  that the size of the work area $aux1$ becomes larger as $k$ increases. A value for $k$ in the range of 5 to
  10 is suitable for most problems.
- $IPARM(3)$ is the flag that determines whether the system is to be preconditioned by an incomplete
  LU factorization with no fill-in.
  If $IPARM(3) = 0$, the system is not preconditioned.
  If $IPARM(3) = 10$, the system is preconditioned by an incomplete LU factorization.
  If $IPARM(3) = -10$, the system is preconditioned by an incomplete LU factorization, where the
  factorization matrix was computed in an earlier call to this subroutine and is stored in $aux2$. 
• IPARM(4), see On Return.

Specified as: an array of (at least) length 4, containing integers, where:

IPARM(1) ≥ 0
IPARM(2) ≥ 0
IPARM(3) = 0, 10, or -10

rparm

is an array of parameters, RPARM(i), where:

RPARM(1) > 0, is the relative accuracy $\varepsilon$ used in the stopping criterion. The iterative procedure is stopped when:

$$\frac{\|b-Ax\|_2}{\|x\|_2} < \varepsilon$$

RPARM(2) is reserved.

RPARM(3), see On Return.

Specified as: a one-dimensional array of (at least) length 3, containing long-precision real numbers.

aux1

has the following meaning:

If naux1 = 0 and error 2015 is unrecoverable, aux1 is ignored.

Otherwise, it is a storage work area used by this subroutine, which is available for use by the calling program between calls to this subroutine. Its size is specified by naux1.

Specified as: an area of storage, containing long-precision real numbers.

naux1

is the size of the work area specified by aux1—that is, the number of elements in aux1.

Specified as: 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 ≥ 7m.

If IPARM(2) > 0, use naux1 ≥ (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) ≤ 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” on page 785 for additional information on aux2. Returned as: an area of storage, containing long-precision real numbers.

Notes
1. When IPARM(3) = -10, this subroutine uses the incomplete LU factorization in aux2, computed in an earlier call to this subroutine. When IPARM(3) = 10, this subroutine computes the incomplete LU factorization and stores it in aux2.
2. If you solve the same sparse linear system of equations several times with different right-hand sides using the preconditioned algorithm, specify IPARM(2) = 10 on the first invocation. The incomplete factorization is stored in aux2. You may save computing time on subsequent calls by setting IPARM(3) equal to -10. In this way, the algorithm reutilizes the incomplete factorization that was computed the first time. Therefore, you should not modify the contents of aux2 between calls.
3. Matrix A must have no common elements with vectors x and b; otherwise, results are unpredictable.
4. In the iterative solvers for sparse matrices, the relative accuracy ε (RPARM(1)) must be specified “reasonably” (10^{-4} to 10^{-8}). The algorithm computes a sequence of approximate solution vectors \( \mathbf{x} \) that converge to the solution. The iterative procedure is stopped when the norm of the residual is sufficiently small—that is, when:
\[
\| \mathbf{b} - \mathbf{A} \mathbf{x} \|_2 / \| \mathbf{x} \|_2 < \varepsilon
\]
As a result, if you specify a larger ε, the algorithm takes fewer iterations to converge to a solution. If you specify a smaller ε, the algorithm requires more iterations and computer time, but converges to a more precise solution. If the value you specify is unreasonably small, the algorithm may fail to converge within the number of iterations it is allowed to perform.
5. For a description of how sparse matrices are stored in compressed-matrix storage mode, see “Compressed-Matrix Storage Mode” on page 110.
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 49.

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 [“107” on page 1258] and [“109” on page 1258]; [“46” on page 1255].

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 1212).

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 65.

- For error 2110, return code 1 indicates that the subroutine exceeded \( IPARM(1) \) iterations without converging. Vector \( x \) contains the approximate solution computed at the last iteration.
- For error 2111, return code 2 indicates that \( aux2 \) contains an incorrect factorization. The subroutine has been called with \( IPARM(3) = -10 \), and \( aux2 \) contains an incomplete factorization of the input matrix \( A \) that was computed by a previous call to the subroutine when \( IPARM(3) = 10 \). This error indicates that \( aux2 \) has been modified since the last call to the subroutine, or that the input matrix is not the same as the one that was factored. If the default action has been overridden, the subroutine can be called again with the same parameters, with the exception of \( IPARM(3) = 0 \) or 10.
- For error 2112, return code 3 indicates that the incomplete LU factorization of \( A \) could not be completed, because one pivot was 0.
- For error 2116, return code 4 indicates that the matrix is singular, because all elements in one row of the matrix contain 0. Array \( AC \) is partially modified and does not represent the same matrix as on entry.

Input-Argument Errors

1. \( m < 0 \)
2. \( lda < 1 \)
3. \( lda < m \)
4. \( nz < 0 \)
5. \( nz = 0 \) and \( m > 0 \)
6. \( IPARM(1) < 0 \)
7. \( IPARM(2) < 0 \)
8. \( IPARM(3) \neq 0, 10, \) 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 squared method. Matrix $A$ is:

\[
\begin{bmatrix}
2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 1.0 & 2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
1.0 & 0.0 & 0.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 1.0 & 2.0 & -1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & -1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & -1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 2.0 \\
\end{bmatrix}
\]

Note: For input matrix KA, ( . ) indicates any value between 1 and 9.

Call Statement and Input:

```
M   NZ   AC   KA   LDA   B   X   IPARM   RPARM   AUX1   NAUX1   AUX2   NAUX2
|    |    |    |    |    |   |     |       |      |       |      |       |
CALL DSMGCG( 9 ,  3 , AC , KA ,  9 , B , X , IPARM , RPARM , AUX1 ,  63 , AUX2 ,   0 )
```

IPARM(1) = 20
IPARM(2) = 0
IPARM(3) = 0
RPARM(1) = 1.0-7

AC =

\[
\begin{bmatrix}
2.0 & 0.0 & 0.0 \\
2.0 & -1.0 & 0.0 \\
1.0 & 2.0 & 0.0 \\
1.0 & 2.0 & -1.0 \\
1.0 & 2.0 & -1.0 \\
1.0 & 2.0 & -1.0 \\
1.0 & 2.0 & 0.0 \\
\end{bmatrix}
\]

KA =

\[
\begin{bmatrix}
1 & . & . \\
2 & 3 & . \\
2 & 3 & . \\
1 & 4 & 5 \\
4 & 5 & 6 \\
5 & 6 & 7 \\
6 & 7 & 8 \\
7 & 8 & 9 \\
8 & 9 & . \\
\end{bmatrix}
\]

B = (2.0, 1.0, 3.0, 2.0, 2.0, 2.0, 2.0, 2.0, 3.0)

X = (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)

Output:

X = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)

IPARM(4) = 9
RPARM(3) = 0.150D-19
Example 2

This example finds the solution of the linear system $Ax = b$ 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 DSMGCG( 9 ,  3 , AC , KA ,  9  , B , X , IPARM , RPARM , AUX1 ,  109  , AUX2 ,  46 )
```

- IPARM(1) = 20
- IPARM(2) = 5
- IPARM(3) = 10
- IPARM(4) = 2
- RPARM(1) = 1.D-7
- RPARM(3) = 0.290D-15
- AC = (same as input AC in Example 1)
- KA = (same as input KA in Example 1)
- $B = (2.0, 1.0, 3.0, 2.0, 2.0, 2.0, 2.0, 2.0, 3.0)$
- $X = (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)$

Output:

```
X        =  (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(4) =  2
RPARM(3) =  0.290D-15
```

**DSDGCG (General Sparse Matrix Iterative Solve Using Compressed-Diagonal Storage Mode)**

**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>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL DSDGCG (m, nd, ad, lda, la, b, x, iparm, rparm, aux1, naux1, aux2, naux2)</td>
</tr>
<tr>
<td>C and C++</td>
<td>dsdgcg (m, nd, ad, lda, la, 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; $m \geq 0$.
- $nd$ is the number of nonzero diagonals stored in the columns of array AD, the number of columns in array AD, and the number of elements in array LA.
Specified as: 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 \( AD \), containing the values of the nonzero elements of the sparse matrix, stored in compressed-matrix storage mode.
Specified as: an \( lda \) by (at least) \( nd \) array, containing long-precision real numbers.

\( lda \)

is the leading dimension of the arrays specified for \( ad \).
Specified as: 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 stored, see “Compressed-Diagonal Storage Mode” on page 111.
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, \( \text{IPARM}(i) \), where:
• \( \text{IPARM}(1) \) controls the number of iterations.
  If \( \text{IPARM}(1) > 0 \), \( \text{IPARM}(1) \) is the maximum number of iterations allowed.
  If \( \text{IPARM}(1) = 0 \), the following default values are used:
    \[ \text{IPARM}(1) = 300 \]
    \[ \text{IPARM}(2) = 0 \]
    \[ \text{IPARM}(3) = 10 \]
    \[ \text{RPARM}(1) = 10^{-6} \]
  • \( \text{IPARM}(2) \) is the flag used to select the iterative procedure used in this subroutine.
    If \( \text{IPARM}(2) = 0 \), the conjugate gradient squared method is used.
    If \( \text{IPARM}(2) = k \), the generalized minimum residual method, restarted after \( k \) steps, is used. Note that the size of the work area \( \text{aux1} \) becomes larger as \( k \) increases. A value for \( k \) in the range of 5 to 10 is suitable for most problems.
  • \( \text{IPARM}(3) \) is the flag that determines whether the system is to be preconditioned by an incomplete LU factorization with no fill-in.
    If \( \text{IPARM}(3) = 0 \), the system is not preconditioned.
    If \( \text{IPARM}(3) = 10 \), the system is preconditioned by an incomplete LU factorization.
    If \( \text{IPARM}(3) = -10 \), 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 \( \text{aux2} \).
  • \( \text{IPARM}(4) \), see On Return.
Specified as: an array of (at least) length 4, containing integers, where:
IPARM(1) ≥ 0
IPARM(2) ≥ 0
IPARM(3) = 0, 10, or -10

\textit{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:

\[
\frac{\| b - Ax \|_2}{\| x \|_2} < \epsilon
\]

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.

\textit{aux1} has the following meaning:

If \( \text{aux1} = 0 \) and error 2015 is unrecoverable, \textit{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{aux1}.

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 \( \text{naux1} = 0 \) 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, \( \text{naux1} > 0 \) and must have at least the following value, where:

If IPARM(2) = 0, use \( \text{naux1} \geq 7m \). If IPARM(2) > 0, use \( \text{naux1} \geq (k+2)m+k(k+4)+1 \), where \( k = \text{IPARM}(2) \).

\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:

For 32-bit integer arguments

\[ \text{naux2} \geq 3 + 2m + 1.5nd(m) \]

For 64-bit integer arguments

\[ \text{naux2} \geq 12 + 3m + 2nd(m) \]

\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.

\textit{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) ≤ 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” on page 791 for additional information on aux2. Returned as: an area of storage, containing long-precision real numbers.

Notes
1. When IPARM(3) = -10, this subroutine uses the incomplete LU factorization in aux2, computed in an earlier call to this subroutine. When IPARM(3) = 10, this subroutine computes the incomplete LU factorization and stores it in aux2.
2. If you solve the same sparse linear system of equations several times with different right-hand sides, using the preconditioned algorithm, specify IPARM(3) = 10 on the first invocation. The incomplete factorization is stored in aux2. You may save computing time on subsequent calls by setting IPARM(3) = -10. In this way, the algorithm reutilizes the incomplete factorization that was computed the first time. Therefore, you should not modify the contents of aux2 between calls.
3. Matrix A must have no common elements with vectors x and b; otherwise, results are unpredictable.
4. In the iterative solvers for sparse matrices, the relative accuracy ε (RPARM(1)) must be specified “reasonably” (10^-4 to 10^-8). The algorithm computes a sequence of approximate solution vectors x that converge to the solution. The iterative procedure is stopped when the norm of the residual is sufficiently small—that is, when:
   \[ \| b-Ax \|_2 / \| x \|_2 < \varepsilon \]
   As a result, if you specify a larger ε, the algorithm takes fewer iterations to converge to a solution. If you specify a smaller ε, the algorithm requires more iterations and computer time, but converges to a more precise solution. If the value you specify is unreasonably small, the algorithm may fail to converge within the number of iterations it is allowed to perform.
5. For a description of how sparse matrices are stored in compressed-diagonal storage mode, see “Compressed-Diagonal Storage Mode” on page 111.
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 49.
**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 [“107” on page 1258] and [“109” on page 1258]. [“46” on page 1255].

**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 65.

- For error 2110, return code 1 indicates that the subroutine exceeded \( IPARM(1) \) iterations without converging. Vector \( x \) contains the approximate solution computed at the last iteration.
- For error 2111, return code 2 indicates that \( aux2 \) contains an incorrect factorization. The subroutine has been called with \( IPARM(3) = -10 \), and \( aux2 \) contains an incomplete factorization of the input matrix \( A \) that was computed by a previous call to the subroutine when \( IPARM(3) = 10 \). This error indicates that \( aux2 \) has been modified since the last call to the subroutine, or that the input matrix is not the same as the one that was factored. If the default action has been overridden, the subroutine can be called again with the same parameters, with the exception of \( IPARM(3) = 0 \) or 10.
- For error 2112, return code 3 indicates that the incomplete LU factorization of \( A \) could not be completed, because one pivot was 0.
- For error 2116, return code 4 indicates that the matrix is singular, because all elements in one row of the matrix contain 0. Array \( AC \) is partially modified and does not represent the same matrix as on entry.

**Input-Argument Errors**

1. \( m < 0 \)
2. \( lda < 1 \)
3. \( lda < m \)
4. \( nd < 0 \)
5. \( nd = 0 \) and \( m > 0 \)
6. \( IPARM(1) < 0 \)
7. \( IPARM(2) < 0 \)
8. \( IPARM(3) \neq 0, 10, \) or \(-10 \)
9. \( RPARM(1) < 1.D0 \)
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. \( naux2 \) 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-diagonal storage mode in arrays $AD$ and $LA$. The system is solved using the conjugate gradient squared method. Matrix $A$ is:

```
| 2.0  0.0  -1.0   0.0   0.0   0.0   0.0   0.0   0.0 |
| 0.0  2.0   0.0  -1.0   0.0   0.0   0.0   0.0   0.0 |
| 0.0  0.0   2.0   0.0  -1.0   0.0   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   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   0.0   0.0   0.0 |
```

Call Statement and Input:

```
CALL DSDGCG( 9 ,  3 , AD ,  9  , LA , B , X , IPARM , RPARM , AUX1 ,  63   , AUX2 ,   0 )
```

```
| 2.0  -1.0  0.0 |
| 2.0  -1.0  0.0 |
| 2.0  -1.0  0.0 |
| 2.0  -1.0  0.0 |

AD =

```
LA       = (0, 2, -4)
B        = (1, 1, 1, 2, 2, 2, 3, 3)
X        = (0, 0, 0, 0, 0, 0, 0, 0, 0)
```

Output:

```
X        = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
IPARM(4) = 8
RPARM(3) = 0.308D-17
```

Example 2

This example finds the solution of the linear system $Ax = b$ 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 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 DSDGCG( 9 ,  3 , AD ,  9  , LA , B , X , IPARM , RPARM , AUX1 , 109  , AUX2 , 46 )
```

```
IPARM(1) = 20
IPARM(2) = 5
IPARM(3) = 10
RPARM(1) = 1.0-7
```
AD = (same as input AD in Example 1)
LA = (same as input LA in Example 1)
B = (1, 1, 1, 1, 2, 2, 2, 3, 3)
X = (0, 0, 0, 0, 0, 0, 0, 0, 0)

Output:

\[
X = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
\]

\[
\text{IPARM}(4) = 6
\]

\[
\text{RPARM}(3) = 0.2500\times10^{-15}
\]

---

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 192: Data Types.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A, U, vt, work</strong></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>

\(^\Delta\)LAPACK
Syntax

Table 193: Syntax for supported computing languages.

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fortran</strong></td>
<td>CALL SGESVD</td>
</tr>
<tr>
<td></td>
<td>CALL CGESVD</td>
</tr>
<tr>
<td></td>
<td>CALL SGESDD</td>
</tr>
<tr>
<td></td>
<td>CALL CGESDD</td>
</tr>
<tr>
<td><strong>C and C++</strong></td>
<td>sgesvd</td>
</tr>
<tr>
<td></td>
<td>cgesvd</td>
</tr>
<tr>
<td></td>
<td>sgesdd</td>
</tr>
<tr>
<td></td>
<td>cgesdd</td>
</tr>
<tr>
<td><strong>LAPACK</strong></td>
<td>info = LAPACK_sgesvd</td>
</tr>
<tr>
<td></td>
<td>info = LAPACK_cgesvd</td>
</tr>
<tr>
<td></td>
<td>info = LAPACK_sgesdd</td>
</tr>
<tr>
<td></td>
<td>info = LAPACK_cgesdd</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. The value of the `matrix_layout` argument 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`† (for SGESVD/DGESVD) or `V`† (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 $V_i$ (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 $m \geq n$, 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 192 on page 794.

$lda$
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; $ldu \geq 0$ and:
- If $jobu = 'A'$ or 'S', $ldu \geq \max(1, m)$.
- If $jobu = 'A'$ or 'S', $ldu \geq \max(1, m)$.
- If $jobu = 'O'$ 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 $jobvt = 'A'$, $ldvt \geq \max(1, n)$.
- If $jobvt = 'S$', $ldvt \geq \max(1, \min(m, n))$.
- 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 192 on page 794.

**work**

is the 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 192 on page 794.

**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 \( lwork \neq -1 \), 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 \( jobz = 'N' \), \( lwork \geq 3\min(m, n) + \max(m, n), 7\min(m, n) \)

If \( jobz = 'O' \):

\[ lwork \geq 3\min(m, n) + \max(m, n), 5\min(m, n)\min(m, n) + 4\min(m, n) \]

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) $5 \times \min(m, n)$.

For CGESDD and ZGESDD

*rwork* is a work area of size (at least) $\max(1, lrwork)$.

If $jobz = 'N'$, $lrwork \geq 5 \times \min(m, n)$.

If $\max(m, n) > \min(m, n)$, $lwork \geq 5 \times \min(m, n) \times \min(m, n) + 5 \times \min(m, n)$.

Otherwise, $lrwork \geq \max(5 \times \min(m, n) \times \min(m, n) + 5 \times \min(m, n), 2 \times \max(m, n) \times \min(m, n) + 2 \times \min(m, n) \times \min(m, n) + \min(m, n))$.

**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 $\max(m, n) \geq 17.0 \times \min(m, n)/9.0$. 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 192 on page 794.

On Return

*a* is overwritten as follows:

If $jobu = 'O'$ or $jobz = 'O'$ and $m \geq n$, $u$ 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 $jobvt = 'O'$ 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 $jobu \neq 'O'$ and $jobvt \neq 'O'$ and $jobz \neq 'O'$, 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 192 on page 794.

*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 192 on page 794.

*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 $m \geq n$, $u$ is not referenced.

Returned as: an ldru 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 192 on page 794.

*vt*

For SGESVD, DGESVD, SGESDD, and DGESDD:

- If $jobvt = 'A'$ or $jobz = 'A'$ or $jobz = 'O'$ and $m \geq n$, $vt$ 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 jobvt = 'A' or jobz = 'A' or jobz = 'O' and m ≥ n, vt contains the matrix \(V^H\) of order n.
- If jobvt = 'S' or jobz = 'S', vt contains the first \(\min(m, n)\) rows of \(V^H\). 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 \(V^H\) instead of \(V\)

Returned as: an \(ldvt\) by (at least) \(n\) array, containing numbers of the data type indicated in Table 192 on page 794.

superb
is a work area used by this subroutine.

Returned as: an area of storage. If info > 0, superb contains the unconverged superdiagonal elements of the upper bidiagonal matrix \(B\) whose diagonal is in \(s\).

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.

If \(lwork = 0\), work is not referenced.

Returned as: an area of storage where:

If \(lwork \geq 1\) or \(lwork = -1\), then \(work_1\) is set to the optimal \(lwork\) value.

For SGESVD or DGESVD, if \(lwork \geq 1\) and \(info > 0\), \(work_2: \min(m, n)\) 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 \(V^H\).

rwork
is a work area used by this subroutine.

Returned as: an area of storage where:

For CGESVD and ZGESVD, if \(info > 0\), \(rwork_1: \min(m, n)\) 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, LAPACKE_sgesvd, LAPACKE_dgesvd, LAPACKE_cgesvd, and LAPACKE_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, $info \geq 0$.

**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, CGESVD, and ZGESD,

$$ A = U \Sigma V^T, \quad U^T = U^{-1} \quad \text{and} \quad V^T = V^{-1} $$

For CGESDD, ZGESDD, CGESDD, and ZGESDD,

$$ A = U \Sigma V^H, \quad U^H = U^{-1} \quad \text{and} \quad V^H = V^{-1} $$

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 [“75” on page 1257, “106” on page 1258].

**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. $jobu \neq 'A$', 'S', 'O', or 'N'
2. \( \text{jobvt} \neq 'A', 'S', 'O', \text{or} 'N' \)
3. \( \text{jobu} = 'O' \) and \( \text{jobvt} = 'O' \)
4. \( \text{jobz} \neq 'A', 'S', 'O', \text{or} 'N' \)
5. \( m < 0 \)
6. \( n < 0 \)
7. \( \text{lda} \leq 0 \)
8. \( m > \text{lda} \)
9. \( \text{ldu} \leq 0 \)
10. \( m > \text{ldu} \) and \( \text{(jobu} = 'A' \text{ or jobz} = 'S') \)
11. \( m > \text{ldu} \) and \( \text{(jobz} = 'A' \text{ or jobz} = 'S') \)
12. \( m > \text{ldu} \) and \( \text{jobz} = 'O' \) and \( m < n \)
13. \( \text{ldvt} \leq 0 \)
14. \( n > \text{ldvt} \) and \( \text{jobvt} = 'A' \)
15. \( n > \text{ldvt} \) and \( \text{jobz} = 'A' \)
16. \( n > \text{ldvt} \) and \( \text{jobz} = 'O' \) and \( m \geq n \)
17. \( \min(m, n) > \text{ldvt} \) and \( \text{jobvt} = 'S' \)
18. \( \min(m, n) > \text{ldvt} \) and \( \text{jobz} = 'S' \)
19. \( \text{lwork} \neq 0 \) and \( \text{lwork} \neq -1 \) and \( \text{lwork} < \) 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 \( \text{lwork} = 0 \), the subroutine dynamically allocates WORK.
5. \( \text{iwork} \) is an integer work array of size 64.

Call Statement and Input:

```fortran
CALL DGESVD( 'N' , 'N' , 4 , 4 , A , 4 , S , U , 1 , VT , 1 , WORK , 0 , INFO )
–or–
CALL DGESDD( 'N' , 4 , 4 , A , 4 , S , U , 1 , VT , 1 , WORK , 0 , IWORK , INFO )
```

\( A = \begin{bmatrix} 1.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 3.0 & 1.0 \end{bmatrix} \)
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 $VT$.

Note:
1. Because $lwork = 0$, the subroutine dynamically allocates WORK.
2. $iwork$ is an integer work array of size 64.

Call Statement and Input:

```
CALL DGESVD( 'A' , 'A' , M , N , A , LDA , S , U , LDU , VT , LDVT , WORK , LWORK , INFO )
```

—or–

```
CALL DGESDD( 'A' , M , N , A , LDA , S , U , LDU , VT , LDVT , WORK , LWORK , IWORK , INFO )
```

Output:

Array $A$ is overwritten.

$$
\begin{bmatrix}
11.344814 \\
0.515729 \\
0.170915 \\
-0.327985 & -0.736976 & -0.591009 \\
-0.591009 & -0.327985 & 0.736976 \\
-0.736976 & 0.591009 & -0.327985 \\
-0.327985 & -0.591009 & -0.736976 \\
0.736976 & 0.327985 & -0.591009 \\
-0.591009 & 0.736976 & -0.327985 \\
\end{bmatrix}
$$
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 )
```

-OR-

```
CALL DGESDD( 'S' , 2 , 4 , A , 2 , S , U , 2 , VT , 2 , WORK , 0 , IWORK , INFO )
```

A    = |  1.0    2.0    3.0    4.0  |
      |  5.0    6.0    7.0    8.0  |

Output:

• For DGESVD:

  A    = | -0.352062 -0.443626 -0.535190 -0.626754 |
         |  0.758981  0.321242 -0.116498 -0.554238 |

• For DGESDD, $A$ has been overwritten on output.

S    = |  14.227407 |
      |  1.257330 |

U    = | -0.376168 -0.926551 |
      | -0.926551  0.376168 |

• For DGESVD, $VT$ is not referenced.
• For DGESDD, $VT$ is:

  $VT$ = | -0.352062 -0.443626 -0.535190 -0.626754 |
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 , 4 , S , U , 1 , VT , 1 , WORK , 0 , RWORK , INFO )
```

**Output:**

Array $A$ is overwritten.

$$
\begin{bmatrix}
4.389511 \\
3.276236 \\
2.346361 \\
1.221907 \\
\end{bmatrix}
$$

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', M, N, A, LDA, S, U, LDU, VT, LDVT, WORK, LWORK, RWORK, INFO )
```

—or–

```
  CALL ZGESDD( 'A', M, N, A, LDA, S, U, LDU, VT, LDVT, WORK, LWORK, IWORK, INFO )
```

```
A = | (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) |
```

Output:

Array A is overwritten.

```
S = | 11.370686 |
    | 2.386257 |
    | 1.006620 |
```

```
U = | (-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) |
```

```
VT = | (-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) |
```

```
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 *lwork* = 0, the subroutine dynamically allocates WORK.
3. *rwork* is a work array of size 60.
CALL ZGESVD( 'S' , 'O' , 4 , 2 ,  A , 4 , S , U , 4 , VT ,  1 , WORK , 0 , RWORK , INFO )

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) |

Output:

A = | (-0.642481, 0.000000) (-0.754181, 0.135753) |
    | (-0.766302, 0.000000) ( 0.632319,-0.113817) |
    | . . |
    | . . |

S = | 14.394066 |
    | 0.900474 |
    | (-0.149426,-0.063497) ( 0.553415,-0.598204) |
    | (-0.352918, 0.014671) ( 0.382229,-0.196618) |
    | (-0.537547,-0.101222) (-0.041751,-0.092615) |
    | (-0.741039,-0.023054) (-0.212937, 0.308971) |

INFO = 0

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 VH 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.
4. rwork is a work array of size 60.

Call Statement and Input:

CALL ZGESDD( 'O' , 2 , 3 ,  A , 2 , S , U , 2 , VT ,  1 , WORK , 0 , RWORK , IWORK , INFO )

A = | (1.0, 1.0) (2.0, 0.0) (3.0, 0.0) |
    | (0.0, 0.0) (5.0,-1.0) (6.0, 0.0) |

Output:

A = | (-0.0488,-0.0488) (-0.6265, 0.0670) (-0.7722,-0.0447) |
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 194: Data Types.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A, \tau, \text{work} )</td>
<td></td>
</tr>
<tr>
<td>Short-precision real</td>
<td>SGEQRF*</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGEQRF*</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGEQRF*</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGEQRF*</td>
</tr>
<tr>
<td>LAPACK</td>
<td></td>
</tr>
</tbody>
</table>

### Syntax

#### Table 195: Syntax for supported computing languages.

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGEQRF</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgeqrf</td>
</tr>
<tr>
<td>LAPACK</td>
<td>info = LAPACKE_sgeqrf</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. The value of the matrix_layout argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

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.

a
is the m by n general matrix A whose QR factorization is to be computed.
Specified as: an lda by (at least) n array, containing numbers of the data type indicated in Table 194 on page 807.

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 194 on page 807.

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 work1. 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 194 on page 807.

\( \tau \)
is the vector \( \tau \), 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 194 on page 807.

\( \text{work} \)
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 \( \text{work}_1 \) is set to the optimal \( lwork \) value and contains numbers of the data type indicated in Table 194 on page 807. Except for \( \text{work}_1 \), the contents of \( \text{work} \) are overwritten on return.

\( \text{info} \)
indicates that a successful computation occurred.

Returned as:

- For \text{SGEQRF, DGEQRF, CGEQRF, and ZGEQRF}, returned as: an integer; \( \text{info} \geq 0 \).
- For \text{LAPACKE_sgeqrf, LAPACKE_dgeqrf, LAPACKE_cgeqrf, and LAPACKE_zgeqrf}, returned as an integer function value; \( \text{info} \geq 0 \).

**Notes and Coding Rules**

1. In your C program, argument \( \text{info} \) 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:

\[ Q = H_1 \cdot H_2 \ldots \cdot H_k \]

where:

\( k = \min(m, n) \)

For each \( i \):

- For \text{SGEQRF and DGEQRF}, \( H_i = I - \tau vv^T \)
- For \text{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 \( v_{1:i-1} = 0 \), \( v_i = 1 \).

\( v_{i+1:m} \) is stored on return in \( A_{i+1:m, i} \)

\( I \) is the identity matrix

Linear algebraic equations subroutines 809
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 [“63” on page 1256, “9” on page 1253, “78” on page 1257, “61” on page 1256, “62” on page 1256].

**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 $lwork < \text{max}(1, n)$

**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 allocate the work area used by this subroutine.

**Call Statements and Input:**

```
CALL DGEQRF ( 6, 2, A, 6, TAU, WORK, 0, INFO)
```

General matrix $A$ of size $6 \times 2$:

```
A =  
   [ 0.000000  2.000000 ]
   [ 2.000000 -1.000000 ]
   [ 2.000000 -1.000000 ]
   [ 0.000000  1.500000 ]
   [ 2.000000 -1.000000 ]
   [ 2.000000 -1.000000 ]
```

**Output:**

General matrix $A$ of size $6 \times 2$.

```
A =  
   [ -4.000000  2.000000 ]
   [ 0.500000  2.500000 ]
   [ 0.500000  2.500000 ]
   [ 0.500000  2.500000 ]
   [ 0.500000  2.500000 ]
   [ 0.500000  2.500000 ]
```
Vector $\tau$ of length 2:

\[
\begin{bmatrix}
1.000000 & 1.400000
\end{bmatrix}
\]

INFO = 0

Example 2

This example shows the QR factorization of a general matrix $A$ of size 4x5.

**Note:** Because $lwork = 0$, DGEQRF dynamically allocates the work area used by this subroutine.

Call Statements and Input:

```
CALL DGEQRF ( 4 , 5 , A , 4 , TAU , WORK , 0 , INFO)
```

General matrix $A$ of size $4 \times 5$:

\[
\begin{bmatrix}
0.500000 & 0.500000 & 1.207107 & 0.000000 & 1.707107 \\
0.500000 & -1.500000 & -0.500000 & 2.414214 & 0.707107 \\
0.500000 & 0.500000 & 0.207107 & 0.000000 & 0.292893 \\
0.500000 & -1.500000 & -0.500000 & -0.414214 & -0.707107 \\
\end{bmatrix}
\]

Output:

General matrix $A$ of size $4 \times 5$:

\[
\begin{bmatrix}
-1.000000 & 1.000000 & -0.207107 & -1.000000 & -1.000000 \\
0.333333 & 2.000000 & 1.207107 & -1.000000 & 1.000000 \\
0.333333 & -0.200000 & 0.707107 & 0.000000 & 1.000000 \\
0.333333 & 0.400000 & 0.071068 & -2.000000 & -1.000000 \\
\end{bmatrix}
\]

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 $6 \times 2$:

\[
\begin{bmatrix}
(-1.800000, -0.900000) & (1.100000, -0.800000) \\
(-1.600000, 1.000000) & (1.700000, 1.400000) \\
(-1.000000, 0.300000) & (1.200000, 0.300000) \\
(1.100000, -0.100000) & (0.700000, -1.900000) \\
(0.500000, 0.700000) & (-0.200000, -1.500000) \\
(-1.500000, -0.700000) & (1.800000, -0.600000)
\end{bmatrix}
\]

Output:
General matrix \(A\) of size 6 × 2:

\[
A = \begin{bmatrix}
(3.660601, 0.000000) & (-1.731956, -0.524504) \\
(0.255874, -0.225302) & (-3.865905, 0.000000) \\
(0.187102, 0.024101) & (0.135165, -0.000348) \\
(-0.193177, 0.050152) & (0.057186, -0.451900) \\
(-0.109713, -0.110108) & (-0.065903, -0.220144) \\
(0.288000, 0.080724) & (0.110147, -0.200172)
\end{bmatrix}
\]

Vector \(\tau\) of length 2:

\[
\tau = \begin{bmatrix}
(1.491723, 0.245861) \\
(1.268358, 0.545419)
\end{bmatrix}
\]

INFO = 0

**Example 4**

This example shows the QR factorization of a general matrix \(A\) of size 3 × 4.

**Note:** Because \(lwork = 0\), ZGEQRF dynamically allocates the work area used by this subroutine.

**Call Statements and Input:**

\[
\begin{array}{cccccccc}
M & N & A & LDA & TAU & WORK & LWORK & INFO \\
3 & 4 & A & 3 & TAU & WORK & 0 & INFO \\
\end{array}
\]

CALL ZGEQRF (3, 4, A, 3, TAU, WORK, 0, INFO)

**Output:**

General matrix \(A\) of size 3 × 4:

\[
A = \begin{bmatrix}
(-1.60, 0.10) & (0.30, 1.70) & (0.30, 0.20) & (-0.50, -1.80) \\
(-1.20, 0.00) & (-0.90, -0.50) & (1.50, 0.80) & (1.50, -1.10) \\
(-0.10, 1.30) & (-1.10, 0.50) & (0.40, -1.30) & (1.60, 0.70)
\end{bmatrix}
\]

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 \(A^H X = B\)
  - If \(\text{transa} = 'C'\) and \(m < n\): find the least squares solution of an overdetermined system; that is, solve the least squares problem: minimize \(\|B - A^H X\|\)

---

**Table 196: Data Types.**

<table>
<thead>
<tr>
<th>A, B, work</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SGELS$^*$</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGELS$^*$</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>CGELS$^*$</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>ZGELS$^*$</td>
</tr>
<tr>
<td>$^*$LAPACK</td>
<td></td>
</tr>
</tbody>
</table>

**Syntax**

**Table 197: Syntax for supported computing languages.**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGELS</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgels</td>
</tr>
<tr>
<td>LAPACK</td>
<td>info = LAPACKE_sgels</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. The value of the \(\text{matrix_layout}\) argument must be \(\text{LAPACK\_ROW\_MAJOR}\) or \(\text{LAPACK\_COL\_MAJOR}\).

**transa**

indicate the form of matrix \(A\) to use in the computation, where:

If \(\text{transa} = 'N'\), matrix \(A\) is used.
If \(\text{transa} = 'T'\), matrix \(A^T\) is used.
If $\text{transa} = 'C'$, 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; $m \geq 0$.

$n$

is the number of columns in matrix $A$ used in the computation.
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 $m$ 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 196 on page 813.

$lda$

is the leading dimension of the array specified for $a$.
Specified as: an integer; $lda > 0$ and $lda \geq m$.

$b$

is the matrix $B$ of right-hand side vectors.
If $\text{transa} = 'N'$, matrix $B$ has $m$ rows and $\text{nrhs}$ columns.
For DGELS and SGELS, if $\text{transa} = 'T'$, matrix $B$ has $n$ rows and $\text{nrhs}$ columns.
For CGELS and ZGELS, if $\text{transa} = 'C'$, matrix $B$ has $n$ rows and $\text{nrhs}$ columns.
Specified as: the $ldb$ by (at least) $\text{nrhs}$ array, containing numbers of the data type indicated in Table 196 on page 813.

$ldb$

is the leading dimension of the array specified for $b$.
Specified as: an integer; $ldb > 0$ and $ldb \geq \max(m,n)$.

$\text{work}$

has the following meaning:
If $lwork = 0$, $\text{work}$ is ignored.
If $lwork \neq 0$, $\text{work}$ 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 196 on page 813.

$lwork$

is the number of elements in array $\text{work}$.
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 \( work_1 \). No computation is performed, and the subroutine returns after error checking is complete.

• Otherwise, it must be:

\[
\text{\( lwork \geq \max(1, mn + \max(mn, nrhs)) \)}
\]

where \( mn = \min(m, n) \).

\[\text{info}\]

See On Return.

**On Return**

\[\text{a}\]

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 196 on page 813.

\[\text{b}\]

is the updated general matrix \( B \), containing the results of the computation. \( B \) is overwritten by the solution vectors, stored columnwise:

- If \( \text{transa} = 'N' \) 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 \( \text{transa} = 'N' \) and \( m < n \), rows 1 to \( n \) of \( B \) contain the minimum norm solution vectors.
- For SGELS and DGELS:
  - If \( \text{transa} = 'T' \) and \( m \geq n \), rows 1 to \( m \) of \( B \) contain the minimum norm solution vectors.
  - If \( \text{transa} = 'T' \) 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 \( \text{transa} = 'C' \) and \( m \geq n \), rows 1 to \( m \) of \( B \) contain the minimum norm solution vectors.
  - If \( \text{transa} = 'C' \) 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 196 on page 813.

\[\text{work}\]

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 196 on page 813.

Except for \( work_1 \), the contents of \( work \) are overwritten on return.

\[\text{info}\]

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 $m \geq n$: 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 $m \geq n$: find the minimum norm solution of an underdetermined system; that is, the problem is $A^TX = B$
  – If transa = 'T' and $m < n$: find the least squares solution of an overdetermined system; that is, solve the least squares problem: minimize $\| B - A^T X \|$
• For CGELS and ZGELS:
  – If transa = 'C' and $m \geq n$: find the minimum norm solution of an underdetermined system; that is, the problem is $A^HX = B$
  – If transa = 'C' and $m < n$: find the least squares solution of an overdetermined system; that is, solve the least squares problem: minimize $\| B - A^H X \|$

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 ["75" on page 1257].

Error conditions
Resource Errors
  lwork = 0 and unable to allocate work space.

Computational Errors
None.

Input-Argument Errors
1. For SGELS and DGELS, transa ≠ 'N' or 'T'
For CGELS and ZGELS, \texttt{transa} = '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. \( lwork \neq 0, lwork \neq -1, \text{ and } lwork < \max(1, mn + \max(mn, nrhs)) \) 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 \( \| \mathbf{B} - \mathbf{AX} \| \). Matrix \( \mathbf{A} \) is size 6 × 2 and matrix \( \mathbf{B} \) is size 6 × 3.

**Note:** Because \( lwork = 0 \), DGELS dynamically allocates the work area used by this subroutine.

**Call Statements and Input:**

```
TRANSA   M   N   NRHS   A   LDA  B  LDB  WORK  LWORK  INFO
|     |   |     |    |    |   |   |     |     |      |
CALL DGELS ('N',  6 , 2 ,   3 ,  A ,  6 , B,  6,  WORK,   0,   INFO )
```

General matrix \( \mathbf{A} \) of size 6 × 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 \( \mathbf{B} \) of size 6 × 3:

\[
\begin{bmatrix}
1.000000000 & 4.000000000 & 1.000000000 \\
1.000000000 & 1.000000000 & 2.000000000 \\
1.000000000 & 1.000000000 & 1.000000000 \\
1.000000000 & 3.000000000 & 2.000000000 \\
1.000000000 & 1.000000000 & 1.000000000 \\
1.000000000 & -1.000000000 & 1.000000000 \\
\end{bmatrix}
\]

**Output:**

General matrix \( \mathbf{A} \) is overwritten.

Solution matrix \( \mathbf{X} \) overwrites \( \mathbf{B} \):

\[
\begin{bmatrix}
.780000000 & 1.000000000 & 1.025000000 \\
.560000000 & 2.000000000 & .800000000 \\
.042857143 & -1.285714286 & -.250000000 \\
.185714286 & .428571429 & 1.250000000 \\
.042857143 & .714285714 & -.250000000 \\
.042857143 & -1.285714286 & -.250000000 \\
\end{bmatrix}
\]

\( \text{INFO} = 0 \)
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:

```plaintext
CALL DGELS ('T', 6, 2, 1, A, 6, B, 6, WORK, 0, INFO)
```

General matrix $A$ of size $6 \times 2$:

$$
A = \begin{bmatrix}
0.00000000 & 2.00000000 \\
2.00000000 & -1.00000000 \\
2.00000000 & -1.00000000 \\
0.00000000 & 1.50000000 \\
2.00000000 & -1.00000000 \\
2.00000000 & -1.00000000
\end{bmatrix}
$$

General matrix $B$ of size $2 \times 1$:

$$
B = \begin{bmatrix}
1.00000000 \\
1.00000000 \\
. . \\
. . \\
. .
\end{bmatrix}
$$

**Output:**

General matrix $A$ is overwritten.

Solution matrix $X$ overwrites $B$:

$$
B = \begin{bmatrix}
.48000000 \\
.12500000 \\
.36000000 \\
.12500000 \\
.12500000
\end{bmatrix}
$$

INFO = 0

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:

```plaintext
CALL DGELS ('N', 3, 4, 4, A, 3, B, 4, WORK, 0, INFO)
```

General matrix $A$ of size $3 \times 4$:

$$
A = \begin{bmatrix}
.50000000 & .50000000 & .50000000 & .50000000 \\
.50000000 & -1.50000000 & .50000000 & -1.50000000 \\
1.00000000 & 1.00000000 & .00000000 & 1.00000000
\end{bmatrix}
$$
General matrix $B$ of size $3 \times 4$:

\[
\begin{bmatrix}
1.000000000 & 1.000000000 & 1.000000000 & 0.000000000 \\
1.000000000 & -1.000000000 & 2.500000000 & 1.000000000 \\
1.000000000 & 1.000000000 & 3.000000000 & 0.000000000 \\
\end{bmatrix}
\]

Output:

General matrix $A$ is overwritten.

Solution matrix $X$ overwrites $B$:

\[
\begin{bmatrix}
1.000000000 & 0.000000000 & 3.500000000 & 0.500000000 \\
0.000000000 & 0.500000000 & -0.250000000 & -0.250000000 \\
1.000000000 & 1.000000000 & -1.000000000 & 0.000000000 \\
\end{bmatrix}
\]

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 $\|B - A^TX\|$. 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:

```fortran
CALL DGELS ( 'T' , 3 , 4 , 4 , A , 3 , B , 4 , WORK , 0 , INFO )
```

General matrix $A$ of size $3 \times 4$:

\[
\begin{bmatrix}
0.500000000 & 0.500000000 & 0.500000000 & 0.500000000 \\
0.500000000 & -1.500000000 & 0.500000000 & -1.500000000 \\
1.207106781 & -1.000000000 & 0.207106781 & -1.000000000 \\
\end{bmatrix}
\]

General matrix $B$ of size $4 \times 4$:

\[
\begin{bmatrix}
1.000000000 & 1.000000000 & 1.000000000 & 0.000000000 \\
1.000000000 & -1.000000000 & 2.000000000 & 2.414213562 \\
1.000000000 & 1.000000000 & 3.000000000 & 0.000000000 \\
1.000000000 & -1.000000000 & 4.000000000 & -0.414213562 \\
\end{bmatrix}
\]

Output:

General matrix $A$ is overwritten.

Solution matrix $X$ overwrites $B$:

\[
\begin{bmatrix}
2.000000000 & 1.000000000 & 6.1232344 & 0.500000000 \\
0.000000000 & 1.000000000 & 0.707106781 & -0.500000000 \\
0.000000000 & 0.000000000 & -2.000000000 & 0.000000000 \\
0.000000000 & 0.000000000 & 1.414213562 & -2.000000000 \\
\end{bmatrix}
\]

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 \( lwork = 0 \), 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 \( B \):

\[
B = \begin{bmatrix}
(-0.16, 0.15) & (-0.08, 0.18) & (0.16, -0.31) \\
(0.11, 0.02) & (0.21, -0.50) & (-0.38, 0.65) \\
(-0.13, -0.32) & (0.16, 0.12) & (-0.27, -0.28) \\
(0.37, -0.05) & (0.04, 0.06) & (-0.19, 0.33)
\end{bmatrix}
\]

\( \text{INFO} = 0 \)

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:

```fortran
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 ZGEOQR.

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 $B$:

$$
B = \begin{pmatrix}
(-1.135350, 0.520298) \\
(0.944864, 0.624509) \\
(1.062824, -0.899701) \\
(2.570856, 1.687827) \\
(2.556854, 2.835820) \\
(-3.982815, -0.231572) \\
\end{pmatrix}
$$

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:

```
TRANSA M N NRHS A LDA B LDB WORK LWORK INFO
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 $B$:

$$
B = \begin{pmatrix}
(2.0, -1.0) \\
(1.0, -1.0) \\
(3.0, 0.0) \\
\end{pmatrix}
$$

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:

```
TRANSA M N NRHS A LDA B LDB WORK LWORK INFO
CALL ZGELS ( 'C' , 2 , 6 , 1 , A , 2 , B , 6 , WORK , 0 , INFO )
```

General matrix $A$ of size $2 \times 6$:

$$
A = \begin{pmatrix}
(2.0, 0.0) & (6.0, 0.0) & (10.0, 0.0) & (14.0, 0.0) & (18.0, 0.0) & (22.0, 0.0) \\
(4.0, 0.0) & (8.0, 0.0) & (12.0, 0.0) & (16.0, 0.0) & (20.0, 0.0) & (24.0, 0.0) \\
\end{pmatrix}
$$
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 198: 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

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fortran</strong></td>
<td>CALL SGELSD</td>
</tr>
<tr>
<td></td>
<td>CALL CGELSD</td>
</tr>
<tr>
<td><strong>C and C++</strong></td>
<td>sge</td>
</tr>
<tr>
<td></td>
<td>cge</td>
</tr>
<tr>
<td><strong>LAPACKE</strong></td>
<td>info = LAPACKE_sgelsd</td>
</tr>
<tr>
<td></td>
<td>info = LAPACKE_cge</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. The value of the `matrix_layout` argument 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 \(A\).

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 198 on page 822.

**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 198 on page 822.

**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 $\varepsilon$ is used, where $\varepsilon$ is the machine precision.

Specified as: a number of data type indicated in Table 198 on page 822.

$work$

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 198 on page 822.

$lwork$

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 $work_1$. No computation is performed, and the subroutine returns after error checking is complete.
• Otherwise:

  For $SGELSD$ and $DGELSD$

  $lwork \geq 12r + 2(r)(smlsiz) + 8(r)(nlvl) + (r)(nrhs) + (smlsiz+1)^2$, where:

  - $r = \min(m,n)$
  - $smlsiz = 25$
  - $nlvl = \max(0, \text{int}(\log_2(r/(smlsiz+1))))+1$

  For $CGELSD$ and $ZGELSD$

  $lwork \geq \max(1, m+n+r, 2r + (r)(nrhs))$, 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:

$lrwork \geq 10r + 2(r)(smlsiz) + 8(r)(nlvl) + 2(smlsiz)(nrhs) + \max((smlsiz+1)^2, n(1 + nrhs) + 2nrhs)$, where:

• $r = \min(m, n)$
• $smlsiz = 25$
• $nlvl = \max(0, \text{int}(\log_2(r/(smlsiz+1)))) + 1)$

Specified as: an area of storage containing numbers of data type indicated in Table 198 on page 822.

$iwork$

is a work area of size max(1, $liwork$), where:

$liwork \geq 3(r)(nlvl) + 11r$ where:

• $r = \min(m, n)$
• $smlsiz = 25$
  
• $nlvl = \max(0, \text{int}(\log_2(r/(smlsiz+1))) + 1)$

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 198 on page 822.

*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 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 198 on page 822.

*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 198 on page 822.

**rank**

is the effective rank of $A$; that is the number of singular values that are greater than $rcond(s_1)$.

Returned as: an integer.

**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 198 on page 822.

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 $info = 0$, $rwork_1$ 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 $info = 0$, $iwork_1$ is set to the minimum size of $iwork$.

**info**

has the following meaning:

If $info = 0$, the subroutine completed successfully.

If $0 < info \leq \max(m, n)$, info specifies how many superdiagonals of an intermediate bidiagonal form did not converge to zero.
If \( \text{info} = \max(m, n) + 1 \), a singular value failed to converge.

SGELSD, DGELSD, CGELSD, and ZGELSD

Returned as:
- For SGELSD, DGELSD, CGELSD, and ZGELSD, returned as: an integer; \( \text{info} \geq 0 \).
- For LAPACKE_sgelsd, LAPACKE_dgelsd, LAPACKE_cgelsd, and LAPACKE_zgelsd, returned as an integer function value; \( \text{info} \geq 0 \).

Notes and Coding rules:
1. In your C program, arguments \( \text{rank} \) and \( \text{info} \) must be passed by reference.
2. \( a, b, s, \text{work}, \text{rwork} \) and \( \text{iwork} \) 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 \( m \geq n \): 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: \( AX = B \)

See reference [“35” on page 1255], [“75” on page 1257].

Error conditions

Resource Errors
\( lwork = 0 \) 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. \( ldb < \max(m, n) \)
8. \( lwork \neq 0 \) and \( lwork \neq -1 \) and \( lwork < \) the minimum required value
9. \( rwork \neq 0 \) and \( rwork \neq -1 \) and \( rwork < \) the minimum required value
10. \( iwork \neq 0 \) and \( iwork \neq -1 \) and \( iwork < \) 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 `lwork`=0, DGELSD dynamically allocates the work area used by this subroutine.
- `iwork` is an integer work array of size 22.

Call Statements and Input:

```
CALL DGELSD ( 6 , 2 , 3 , A , 6 , B , 6 , S , RCOND , RANK , WORK , 0 , IWORK , INFO )
```

A = (same as input A in Example 1 for DGELS)

B = (same as input B in Example 1 for DGELS)

```
RCOND  = .7450580D-08
```

Output:

General matrix A is overwritten.

Solution matrix X overwrites B:

```
| 0.7800000000  1.0000000000  1.0250000000 |
| 0.5600000000  2.0000000000  0.8000000000 |
```

```
B    =  | 0.042857143  -1.285714286 -0.250000000 |
| 0.185714286  0.428571429  1.250000000 |
| 0.042857143  0.714285714 -0.250000000 |
| 0.042857143 -1.285714286 -0.250000000 |
```

```
S    =  | 4.650367627 |
| 2.150367627 |
```

```
RANK  =  2
```

```
INFO  =  0
```

Example 2

This example finds the minimum norm solution of an underdetermined real general system \( AX = B \). Matrix A is size 3 × 4. On input, matrix B is size 3 × 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 , S , RCOND , RANK , WORK , 0 , IWORK , INFO )
```
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:

```plaintext
M   N   NRHS   A   LDA  B   LDB  S   RCOND   RANK   WORK LWORK RWORK
IWORK INFO

|   |     |    |    |   |    |   |     |       |      |    |     |
|   | 6   | 2   | 1   | A   6 | B   6 | S   RCOND | RANK | WORK | 0   | RWORK |
|   |     |     |     |     |   |   |   |       |      |     |     |

CALL ZGELSD (6, 2, 1, A, 6, B, 6, 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)

**Output:**

General matrix \( A \) is overwritten.
Solution matrix $X$ overwrites $B$:

\[
B = \begin{bmatrix}
(-1.135350, 0.520298) \\
(0.944064, 0.624509) \\
1.062824, -0.899701 \\
2.570856, 1.687826 \\
2.556854, 2.835826 \\
(-3.982815, -0.231572)
\end{bmatrix}
\]

\[
S = \begin{bmatrix}
4.781121271 \\
2.959878261
\end{bmatrix}
\]

RANK = 2

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:

\[
\text{CALL ZGELSD ( 3, 4, 3, A, 3, B, 4, S, RCOND, RANK, WORK, 0, RWORK, IWORK, INFO )}
\]

A = (same as input A in Example 5 for ZGELS)
B = (same as input B in Example 5 for ZGELS)

\[
RCOND = -1
\]

Output:

General matrix $A$ is overwritten.

Solution matrix $X$ overwrites $B$:

\[
B = \begin{bmatrix}
(-0.16, 0.15) \\
(-0.08, 0.18) \\
0.16, -0.31 \\
0.11, 0.02 \\
0.21, -0.50 \\
(-0.38, 0.65) \\
(-0.13, -0.32) \\
0.16, 0.12 \\
-0.27, -0.28 \\
0.37, -0.05 \\
0.04, 0.06 \\
-0.19, 0.33
\end{bmatrix}
\]

\[
S = \begin{bmatrix}
9.895527537 \\
4.876518979
\end{bmatrix}
\]
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 $AX \approx B$, follow the call to these subroutines with a call to SGESVS or DGESVS, respectively.

Table 200: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A, B, s, aux$</td>
<td>SGESVF</td>
</tr>
<tr>
<td>Short-precision real</td>
<td></td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGESVF</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGESVF</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgesvf</td>
</tr>
</tbody>
</table>

On Entry

$iopt$

indicates the type of computation to be performed, where:

- If $iopt = 0$ or $10$, singular values are computed.
- If $iopt = 1$ or $11$, singular values and $V$ are computed.
- If $iopt = 2$ or $12$, singular values, $V$, and $U^TB$ are computed.

Specified as: an integer; $iopt = 0, 1, 2, 10, 11, or 12$.

- If $iopt < 10$, singular values are unordered.
- If $iopt \geq 10$, 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 200 on page 830.

$lda$

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 $iopt = 0$, 1, 10, or 11, this argument is not used in the computation.

If $iopt = 2$ or 12, it is the $m$ by $nb$ matrix $B$.

Specified as: an $ldb$ by (at least) $nb$ array, containing numbers of the data type indicated in Table 200 on page 830.

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 200 on page 830.

$naux$

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 \(i\text{opt} = 0,\) or \(10,\) \(A\) is overwritten; that is, the original input is not preserved.

If \(i\text{opt} = 1, 2, 11,\) or \(12,\) \(A\) contains the real orthogonal matrix \(V,\) of order \(n,\) in its first \(n\) rows and \(n\) columns. If \(i\text{opt} = 11\) 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 \(\text{lda} \) by (at least) \(n\) array, containing numbers of the data type indicated in Table 200 on page 830.

\(b\)

has the following meaning, where:

If \(i\text{opt} = 0, 1, 10,\) or \(11,\) \(B\) is not used in the computation.

If \(i\text{opt} = 2\) or \(12,\) \(B\) is overwritten by the \(n\) by \(nb\) matrix \(U^TB.\)

If \(i\text{opt} = 12,\) the rows of \(U^TB\) 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 \(\text{ldb} \) by (at least) \(nb\) array, containing numbers of the data type indicated in Table 200 on page 830.

\(s\)

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 200 on page 830; \(s_i \geq 0,\) where:

If \(i\text{opt} < 10,\) the singular values are unordered in \(s.\)

If \(i\text{opt} \geq 10,\) 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 \(U^TB\) 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 \(i\text{opt} = 0, 1, 10,\) 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 49.

Function

The singular value decomposition of a real general matrix is computed as follows:

\[A = UΣV^T\]

where:

\[U^TU = V^TV = VV^T = I\]

\(A\) 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 $B$.

$\Sigma$ is an $n$ by $n$ real diagonal matrix. The diagonal elements of $\Sigma$ 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 [“120” on page 1259].)
   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 [“21” on page 1254].)
   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 [“21” on page 1254], [“71” on page 1257], [“94” on page 1258], and pages 134 to 151 in reference [“120” on page 1259]. These algorithms have a tendency to generate underflows that may hurt overall performance. The system default is to mask underflow, which improves the performance of these subroutines.

**Error conditions**

**Resource Errors**

Error 2015 is unrecoverable, $naux = 0$, and unable to allocate work area.

**Computational Errors**

Singular value ($i$) failed to converge after ($x$) iterations.

- The singular values ($s_j, j = n, n-1, ..., i+1$) are correct. If $i opt < 10$, they are unordered. Otherwise, they are ordered.
- $a$ has been modified.
- If $i opt = 2$ or 12, then $b$ has been modified.
- The return code is set to 1.
• *i* and *x* can be determined at run time by use of the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2107 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. See “What Can You Do about ESSL Computational Errors?” on page 65.

**Input-Argument Errors**

1. *iopt* ≠ 0, 1, 2, 10, 11, or 12
2. *lda* ≤ 0
3. *max(m, n) > lda*
4. *ldb* ≤ 0 and *iopt* = 2, 12
5. *max(m, n) > ldb* and *iopt* = 2, 12
6. *nb* ≤ 0 and *iopt* = 2, 12
7. *m* < 0
8. *n* < 0
9. Error 2015 is recoverable or *naux* ≠ 0, and *naux* is too small—that is, less than the minimum required value. Return code 2 is returned if error 2015 is recoverable.

**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:**

```
CALL DGESVF( 0  , A , 4 , DUMMY , 1 , 0 , S , 4 , 3 , AUX , 7  )
```

```
<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
</tr>
<tr>
<td>4.0</td>
</tr>
<tr>
<td>7.0</td>
</tr>
<tr>
<td>10.0</td>
</tr>
</tbody>
</table>
```

**Output:**

```
S        =   (25.462, 1.291, 0.000)
```

**Example 2**

This example computes the singular values, *s*, of a real long-precision general matrix *A* and the matrix *V*, where:

• *M* is equal to *N*.
• *NAUX* is greater than or equal to *2N+max(M, N) = 9*.
• *LDB* has been set to 1 to avoid a Fortran error message.
• *DUMMY* is a placeholder for argument *b*, which is not used in the computation.
• The singular values are returned in *S*.
• The matrix $V$ is returned in $A$.

Call Statement and Input:

```
IOPT  A  LDA  B  LDB  NB  S  M  N  AUX  NAUX
CALL DGESVF( 1 , A , 3 , DUMMY , 1 , 0 , S , 3 , 3 , AUX , 9 )
```

$$A = \begin{bmatrix}
2.0 & 1.0 & 1.0 \\
4.0 & 1.0 & 0.0 \\
-2.0 & 2.0 & 1.0 \\
\end{bmatrix}$$

Output:

$$A = \begin{bmatrix}
-0.994 & 0.105 & -0.041 \\
-0.112 & -0.870 & 0.480 \\
-0.015 & -0.482 & -0.876 \\
\end{bmatrix}$$

$$S = (4.922, 2.724, 0.597)$$

**Example 3**

This example computes the singular values, $s$, and computes matrices $V$ and $U^TB$ in preparation for solving the underdetermined system $AX = B$, where:

• $M$ is less than $N$.
• $N_{AUX}$ is greater than or equal to $2N + \max(M, N, NB) = 9$.
• The singular values are returned in $S$.
• The matrix $V$ is returned in $A$.
• The matrix $U^TB$ is returned in $B$.

Call Statement and Input:

```
IOPT  A  LDA  B  LDB  NB  S  M  N  AUX  NAUX
CALL DGESVF( 2 , A , 3 , B , 3 , 1 , S , 2 , 3 , AUX , 9 )
```

$$A = \begin{bmatrix}
1.0 & 2.0 & 2.0 \\
2.0 & 4.0 & 5.0 \\
\cdot & \cdot & \cdot \\
\end{bmatrix}$$

$$B = \begin{bmatrix}
1.0 \\
4.0 \\
\cdot \\
\end{bmatrix}$$

Output:

$$A = \begin{bmatrix}
-0.304 & -0.894 & 0.328 \\
-0.608 & 0.447 & 0.656 \\
-0.733 & 0.000 & -0.680 \\
\end{bmatrix}$$

$$B = \begin{bmatrix}
-4.061 \\
\cdot \\
\cdot \\
\end{bmatrix}$$

$$S = (7.342, 0.000, 0.305)$$
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 \( N \).
- \( N \text{AUX} \) 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  )
```

\[
A = \begin{pmatrix}
1.0 & 4.0 \\
2.0 & 5.0 \\
3.0 & 6.0
\end{pmatrix}
\]

\[
B = \begin{pmatrix}
7.0 & 10.0 \\
8.0 & 11.0 \\
9.0 & 12.0
\end{pmatrix}
\]

Output:

\[
A = \begin{pmatrix}
0.922 & -0.386 \\
-0.386 & -0.922 \\
. & .
\end{pmatrix}
\]

\[
B = \begin{pmatrix}
-1.310 & -2.321 \\
-13.867 & -18.963 \\
. & .
\end{pmatrix}
\]

\[
X = (0.773, 9.508)
\]

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 \( N \).
- \( N \text{AUX} \) 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  )
```

\[
A = \begin{pmatrix}
1.0 & 4.0 \\
2.0 & 5.0
\end{pmatrix}
\]
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 201: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V, UB, X, s, \tau$</td>
<td></td>
</tr>
<tr>
<td>Short-precision real</td>
<td>SGESVS</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGESVS</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGESVS</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgesvs</td>
</tr>
</tbody>
</table>

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 201 on page 837.

$ldv$

is the leading dimension of the array specified for $v$.

Specified as: an integer; $ldv > 0$ and $ldv \geq n$. 

Linear algebraic equations subroutines 837
**ub**

is an \( n \) by \( nb \) matrix, containing \( U^T B \). It is produced by a preceding call to SGESVF or DGESVF, where it corresponds to output argument \( b \). On output, \( U^T B \) is overwritten; that is, the original input is not preserved.

Specified as: an \( ldub \) by (at least) \( nb \) array, containing numbers of the data type indicated in Table 201 on page 837.

**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 \( U^T B \).

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 201 on page 837; \( s_i \geq 0 \).

**x**

See On Return.

**ldx**

is the leading dimension of the array specified for \( x \).

Specified as: an integer; \( ldx > 0 \) and \( ldx \geq n \).

**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 201 on page 837; \( \tau \geq 0 \). For more information on the values for \( \tau \), see “Notes” on page 838.

**On Return**

**x**

is an \( n \) by \( nb \) matrix, containing the minimal norm linear least solutions of \( AX \approx B \). 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 201 on page 837.

**Notes**

1. \( V, X, s, \) and \( U^T B \) 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:

\[
\tau \geq \max(|\Delta_j|)
\]
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:

$$\tau \geq \varepsilon \left(\sqrt{mn}\right) \max(s_j) \quad \text{for} \quad j = (1, \ldots, n)$$

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 [“21” on page 1254], [“71” on page 1257], [“94” on page 1258], and pages 134 to 151 in reference [“120” on page 1259].

Function

The minimal norm linear least squares solution of $AX \cong B$, 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 $AX \cong B$, is given by the following formula:

$$X = V \Sigma^+ U^T B$$

where:

- $\Sigma^+$ is the diagonal matrix with elements $\sigma^+_j$, where:
  $$\sigma^+_j = 1.0 / \sigma_j \quad \text{if} \quad \sigma_j \geq \tau \text{ and } \sigma_j \neq 0$$
  $$\sigma^+_j = 0 \quad \text{for all other cases}$$

If $m$ or $n$ is equal to 0, no computation is performed. See references [“21” on page 1254], [“71” on page 1257], [“94” on page 1258], and pages 134 to 151 in reference [“120” on page 1259]. 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. $n < 0$
Examples

Example 1

This example finds the linear least squares solution for the underdetermined system \( AX = B \), using the singular value decomposition computed by DGESV. Matrix \( A \) is:

\[
\begin{pmatrix}
1.0 & 2.0 & 2.0 \\
2.0 & 4.0 & 5.0
\end{pmatrix}
\]

and matrix \( B \) is:

\[
\begin{pmatrix}
1.0 \\
4.0
\end{pmatrix}
\]

On output, matrix \( U^T B \) is overwritten.

Note: This example corresponds to DGESVF Example 3.

Call Statement and Input:

\[
\begin{align*}
V & \quad \text{LDV} & \quad UB & \quad \text{LDUB} & \quad NB & \quad S & \quad X & \quad \text{LDX} & \quad M & \quad N & \quad TAU \\
\text{CALL DGESVS(} & \quad V, 3, \quad UB, 3, 1, \quad S, \quad X, 3, 2, 3, \quad TAU \text{)}
\end{align*}
\]

\[
\begin{align*}
V & = \begin{pmatrix}
-0.384 & -0.894 & 0.328 \\
-0.608 & 0.447 & 0.656 \\
-0.733 & 0.000 & -0.680
\end{pmatrix} \\
UB & = \begin{pmatrix}
-4.061 \\
0.000 \\
-0.714
\end{pmatrix} \\
S & = (7.342, 0.000, 0.305) \\
TAU & = 0.3993 \times 10^{-14}
\end{align*}
\]

Output:

\[
\begin{align*}
X & = \begin{pmatrix}
-0.690 \\
-1.280 \\
2.000
\end{pmatrix}
\end{align*}
\]

Example 2

This example finds the linear least squares solution for the overdetermined system \( AX = B \), using the singular value decomposition computed by DGESV. Matrix \( A \) is:

\[
\begin{pmatrix}
1.0 & 4.0 \\
2.0 & 5.0 \\
3.0 & 6.0
\end{pmatrix}
\]

and where \( B \) is:

\[
\begin{pmatrix}
7.0 & 10.0 \\
8.0 & 11.0
\end{pmatrix}
\]
On output, matrix $U^T B$ is overwritten.

**Note:** This example corresponds to DGESV Example 4.

Call Statement:

```
CALL DGESVS(V, 3, UB, 3, 2, S, X, 2, 3, 2, TAU)
```

**Input:**

```
V  = | 0.922  -0.386 |
    | -0.386  -0.922 |
    | .        .     |

UB = | -1.310  -2.321 |
     | -13.867  -18.963 |
     | .        .     |

S    = (0.773, 9.508)
TAU  = 0.5171D-14
```

**Output:**

```
X    = | -1.000  -2.000 |
      | 2.000   3.000 |
```

---

**SGELLS and DGELLS (Linear Least Squares Solution for a General Matrix with Column Pivoting)**

**Purpose**

These subroutines compute the minimal norm linear least squares solution of $AX = B$, using a QR decomposition with column pivoting.

**Table 202: Data Types.**

<table>
<thead>
<tr>
<th><strong>Data Types</strong></th>
<th><strong>Subroutine</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>$A, B, X, rn, \tau, aux$</td>
<td>SGELLS</td>
</tr>
<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><strong>Language</strong></th>
<th><strong>Syntax</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SGELLS</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgells</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 202 on page 841.

**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 202 on page 841.

**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 202 on page 841; \( \tau \geq 0 \). For more information on how to select a value for \( \tau \), see “Notes” on page 843.

**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 \( rn \).
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 202 on page 841. 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 ≥ 3n + max(n, nb)

For 64-bit integer arguments

naux ≥ 4n + max(n, nb) + 3

For DGELLS

For DGELLS, it must have the following values:

For 32-bit integer arguments

naux ≥ ceil(2.5n) + max(n, nb)

For 64-bit integer arguments

naux ≥ 3n + max(n, nb)

On Return

x

is the solution matrix X, with n rows and nb columns, where:

If k ≠ 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 202 on page 841.

rn

is the vector rn of length nb, where:

If iopt = 0 or k = 0, rn is not used in the computation.

If iopt = 1, rn 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 202 on page 841.

k

is the number of columns of matrix A used in the solution. Returned as: an integer; k = (the number of diagonal elements of matrix R exceeding τ in magnitude).

Notes

1. In your C program, argument k must be passed by reference.

2. If ldx ≥ max(m, n), matrix X and matrix B can be the same; otherwise, matrix X and matrix B can have no common elements, or the results are unpredictable.
3. The following items must have no common elements; otherwise, results are unpredictable:
   - Matrices $A$ and $X$, vector $rn$, and the data area specified for $aux$
   - Matrices $A$ and $B$, vector $rn$, and the data area specified for $aux$.

4. If the relative uncertainty in the matrix $B$ is $\rho$, then:
   \[ \tau \geq \rho \|A\|_F \]
   See references [“54” on page 1256], [“75” on page 1257], and [“94” on page 1258] for additional guidance on determining suitable values for $\tau$.

5. When you specify $iopt = 0$, you must also specify a dummy argument for $rn$. For more details, see Example 1.

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 49.

Function

The minimal norm linear least squares solution of $AX \approx B$ 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 $AX \approx B$. $A$ 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 [“54” on page 1256], [“75” on page 1257], and [“94” on page 1258]. 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 ≠ 0 or 1
2. lda ≤ 0
3. m > lda
4. ldb ≤ 0
5. m > ldb
6. ldx ≤ 0
7. n > ldx
8. m < 0
9. n < 0
10. nb ≤ 0
11. τ < 0
12. 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 solves the underdetermined system \( AX= B \). On output, \( A \) and \( B \) are overwritten. DUMMY is used as a placeholder for argument \( rn \), which is not used in the computation.

Call Statement and Input:

\[
\begin{array}{cccccccccccc}
\text{IOPT} & A & LDA & B & LDB & X & LDX & RN & TAU & M & N & NB & K & AUX & NAUX \\
\hline
0 & A & 2 & B & 2 & X & 3 & DUMMY & TAU & 2 & 3 & 1 & K & AUX & 11 \\
\end{array}
\]

\[
A = \begin{bmatrix}
1.0 & 2.0 & 2.0 \\
2.0 & 4.0 & 5.0 \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
1.0 \\
4.0 \\
\end{bmatrix}
\]

\[
TAU = 0.0
\]

Output:

\[
X = \begin{bmatrix}
-0.600 \\
-1.200 \\
2.000 \\
\end{bmatrix}
\]

\[
K = 2
\]

Example 2
This example solves the overdetermined system \( AX= B \). On output, \( A \) and \( B \) are overwritten. DUMMY is used as a placeholder for argument \( rn \), which is not used in the computation.
Call Statement and Input:

```plaintext
CALL DGELLS( 0 , A , 3 , B , 3 , X , 2 , DUMMY , TAU , 3 , 2 , 2 , K , AUX , 7 )
```

<table>
<thead>
<tr>
<th>A</th>
<th>1.0</th>
<th>4.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.0</td>
<td>5.0</td>
</tr>
<tr>
<td></td>
<td>3.0</td>
<td>6.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>7.0</th>
<th>10.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8.0</td>
<td>11.0</td>
</tr>
<tr>
<td></td>
<td>9.0</td>
<td>12.0</td>
</tr>
</tbody>
</table>

| TAU | 0.0  |

Output:

<table>
<thead>
<tr>
<th>X</th>
<th>-1.000</th>
<th>-2.000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.000</td>
<td>3.000</td>
</tr>
</tbody>
</table>

K = 2

**Example 3**

This example solves the overdetermined system $AX = B$ and computes the Euclidean Norms of the residual vectors. On output, $A$ and $B$ are overwritten.

Call Statement and Input:

```plaintext
CALL DGELLS( 1 , A , 3 , B , 3 , X , 2 , RN , TAU , 3 , 2 , 2 , K , AUX , 7 )
```

<table>
<thead>
<tr>
<th>A</th>
<th>1.1</th>
<th>-4.3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.0</td>
<td>-5.0</td>
</tr>
<tr>
<td></td>
<td>3.0</td>
<td>-6.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>-7.0</th>
<th>10.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-8.0</td>
<td>11.0</td>
</tr>
<tr>
<td></td>
<td>-9.0</td>
<td>12.0</td>
</tr>
</tbody>
</table>

| TAU | 0.0  |

Output:

<table>
<thead>
<tr>
<th>X</th>
<th>0.543</th>
<th>-1.360</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.785</td>
<td>-2.699</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RN</th>
<th>0.196</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.275</td>
</tr>
</tbody>
</table>

K = 2
Chapter 11. Eigensystem analysis subroutines

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 [“9” on page 1253].

<table>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGEEVΔ</td>
<td>DGEEVΔ</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 850</td>
</tr>
<tr>
<td>CGEEVΔ</td>
<td>ZGEEVΔ</td>
<td></td>
</tr>
<tr>
<td>SGEEVXΔ</td>
<td>DGEEVXΔ</td>
<td></td>
</tr>
<tr>
<td>CGEEVXΔ</td>
<td>ZGEEVXΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgeevΔ</td>
<td>LAPACKE_dgeevΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgeevΔ</td>
<td>LAPACKE_zgeevΔ</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgeevxΔ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgeevxΔ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Short-Precision Subroutine</td>
<td>Long-Precision Subroutine</td>
<td>Descriptive Name and Location</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>SSYEV&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>DSYEV&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SSYEV, DSYEV, CHEEV, ZHEEV, SSPEVX, DSPEVX, CHPEVX, ZHPEVX, SSYEVX, DSYEVX, CHEEVX, ZHEEVX, SSYEVR, DSYEVR, CHEEVR, and ZHEEVR (Eigenvalues, and Optionally, the Eigenvectors of a Real Symmetric or Complex Hermitian Matrix)” on page 867</td>
</tr>
<tr>
<td>CHEEV&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>ZHEEV&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SSPEVX&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>DSPEVX&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CHPEVX&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>ZHPEVX&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SSYEVR&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>DSYEVR&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CHEEVR&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>ZHEEVR&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssyev&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>LAPACKE_dsyev&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cheev&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>LAPACKE_zheev&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sspevx&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>LAPACKE_dspevx&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chpevx&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssyevx&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cheevx&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssyevr&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cheevr&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSPEVD&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>DSPEVD&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“SSPEVD, DSPEVD, CHPEVD, ZHPEVD, SSYEV, DSYEV, CHEEV, and ZHEEVR (Eigenvalues and, Optionally the Eigenvectors, of a Real Symmetric or Complex Hermitian Matrix Using a Divide-and-Conquer Algorithm)” on page 893</td>
</tr>
<tr>
<td>CHPEVD&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>ZHPEVD&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SSYEVD&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>DSYEVD&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CHEEVD&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>ZHEEVD&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sspevd&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chpevd&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssyevd&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cheevd&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SGGEV&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>DGGEV&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>“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 905</td>
</tr>
<tr>
<td>CGGEV&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>ZGGEV&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>SGGEVX&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>DGGEVX&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>CGGEVX&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td>ZGGEVX&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sggev&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cggev&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sggevx&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cggevx&lt;sup&gt;Δ&lt;/sup&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

848 IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and Reference
### Performance and Accuracy Considerations

1. The short precision subroutines provide increased accuracy by accumulating intermediate results in long precision when the 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 61.

### 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 204: Data Types.**

<table>
<thead>
<tr>
<th>$A$, $vl$, $vr$, $work$, $wr$, $wi$, $w$</th>
<th>$scale$, $abnrm$, $rconde$, $rcondv$, $rwork$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SGEEV$^\Delta$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SGEEVX$^\Delta$</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DGEEV$^\Delta$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DGEEVX$^\Delta$</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>CGEEV$^\Delta$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CGEEVX$^\Delta$</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>ZGEEV$^\Delta$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZGEEVX$^\Delta$</td>
</tr>
</tbody>
</table>

$^\Delta$ LAPACK
### Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fortran</strong></td>
<td>CALL SGEEV</td>
</tr>
<tr>
<td></td>
<td>CALL CGEEV</td>
</tr>
<tr>
<td></td>
<td>CALL SGEEVX</td>
</tr>
<tr>
<td></td>
<td>CALL CGEEVX</td>
</tr>
<tr>
<td><strong>C and C++</strong></td>
<td>sgeev</td>
</tr>
<tr>
<td></td>
<td>cgeev</td>
</tr>
<tr>
<td></td>
<td>sgeevx</td>
</tr>
<tr>
<td></td>
<td>cgeevx</td>
</tr>
<tr>
<td><strong>LAPACK</strong></td>
<td>info = LAPACKE_sgeev</td>
</tr>
<tr>
<td></td>
<td>info = LAPACKE_cgeev</td>
</tr>
<tr>
<td></td>
<td>info = LAPACKE_sgeevx</td>
</tr>
<tr>
<td></td>
<td>info = LAPACKE_cgeevx</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. The value of the `matrix_layout` argument 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'.

$jobv_l$ indicates whether or not to compute the left eigenvectors of $A$, where $jobv_l$ 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', $jobv_l$ must = 'V'.

Specified as: a single character. It must be 'N' or 'V'.

$jobv_r$ indicates whether or not to compute the right eigenvectors of $A$, where $jobv_r$ 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', $jobv_r$ 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 $jobv_l$ and $jobv_r$ 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 204 on page 850.

$lda$ 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 \( jobvl = 'V' \), \( ldvl \geq n \).

**\( ldvr \)**
is the leading dimension of the array specified for \( vr \).
Specified as: an integer; \( ldvr > 0 \); if \( jobvr = 'V' \), \( ldvr \geq n \).

**\( work \)**
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 204 on page 850.

**\( lwork \)**
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 \( jobvr = 'N' \), \( lwork \geq \max(1, 3n) \).
  - If \( jobvl = 'V' \) or \( jobvr = 'V' \), \( lwork \geq \max(1, 4n) \).
- For CGEEV and ZGEEV, \( lwork \geq \max(1, 2n) \).
- For SGEEVX and DGEEVX:
  - If \( sense = 'N' \) or \( E' \):
    - If \( jobvl = 'N' \) and \( jobvr = 'N' \), \( lwork \geq \max(1, 2n) \).
    - If \( jobvl = 'V' \) or \( jobvr = 'V' \), \( lwork \geq \max(1, 3n) \).
  - If \( sense = 'V' \) or \( B' \), \( lwork \geq \max(1, n(n + 6)) \).
- For CGEEVX and ZGEEVX:
  - If \( sense = 'N' \) or \( E' \), \( lwork \geq \max(1, 2n) \).
  - If \( sense = 'V' \) or \( B' \), \( lwork \geq \max(1, n^2 + 6n^2) \).

**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 204 on page 850.
**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 204 on page 850.

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 204 on page 850.

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 204 on page 850.

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 204 on page 850.

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 204 on page 850.

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 204 on page 850.
ilo
has the following meaning:
If \( \text{balanc} = 'N', \) \( \text{ilo} = 1. \)
Otherwise, the value of \( \text{ilo} \) is determined when \( A \) is balanced.
The balanced \( a_{ij} = 0 \) if \( i > j \) and \( j = 1, \ldots, (\text{ilo}-1) \) or \( i = (\text{ihi}+1), \ldots, n. \)
Returned as: an integer; \( 1 \leq \text{ilo} \leq n. \)

ihi
has the following meaning:
If \( \text{balanc} = 'N', \) \( \text{ihi} = n. \)
Otherwise, the value of \( \text{ihi} \) is determined when \( A \) is balanced.
The balanced \( a_{ij} = 0 \) if \( i > j \) and \( j = 1, \ldots, (\text{ilo}-1) \) or \( i = (\text{ihi}+1), \ldots, n. \)
Returned as: an integer; \( 1 \leq \text{ihi} \leq n. \)

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:
\[
\begin{align*}
\text{scale}_j &= p_j, \text{ for } j = 1, \ldots, (\text{ilo}-1) \\
\text{scale}_j &= d_j, \text{ for } j = \text{ilo}, \ldots, \text{ihi} \\
\text{scale}_j &= p_j, \text{ for } j = (\text{ihi}+1), \ldots, n
\end{align*}
\]
Returned as: a one-dimensional array of (at least) length \( n \) containing numbers of the data type indicated in Table 204 on page 850.

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 204 on page 850; \( \text{abnrm} \geq 0. \)

rconde
contains the computed reciprocal condition numbers of the eigenvalues, where \( \text{rconde}_j \) 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 204 on page 850.

rcondv
contains the computed reciprocal condition numbers of the eigenvectors, where \( \text{rcondv}_j \) 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 204 on page 850.

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 \( \text{work}_1 \) is set to the optimal \( lwork \) value and contains numbers of the data type indicated in Table 204 on page 850.
Except for \( \text{work}_1, \) the contents of \( \text{work} \) 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 204 on page 850.

**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 [“15” on page 1254], [“16” on page 1254], [“67” on page 1257], [“68” on page 1257], and [“76” on page 1257].

**Error conditions**

**Resource Errors**

$lwork = 0$, 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. \(balanc \neq \text{‘N’, ‘S’, ‘P’, or ‘B’}\)

2. \(jobvl \neq \text{‘N’, or ‘V’}\)

3. \(jobvr \neq \text{‘N’, or ‘V’}\)

4. \(sense \neq \text{‘N’, ‘E’, ‘V’, or ‘B’}\)

5. \((\text{sense} = \text{‘E’ or sense} = \text{‘B’}) \text{ and } (\text{jobvl} \neq \text{‘V’ or jobvr} \neq \text{‘V’})\)

6. \(n < 0\)

7. \(lda \leq 0\)

8. \(n > lda\)

9. \(ldvl \leq 0\)

10. \(ldvr \leq 0\)

11. \(jobvl \neq \text{‘V’ and ldvl} < n\)

12. \(jobvr \neq \text{‘V’ and ldvr} < n\)

13. For SGEEV and DGEEV, if \(lwork \neq 0\):

   • If \(jobvl = \text{‘N’ and jobvr} = \text{‘N’\}, lwork < \text{max(1, 3n)}\).
   • If \(jobvl = \text{‘V’ or jobvr} = \text{‘V’\}, lwork < 4n\).

   For CGEEV and ZGEEV, if \(lwork \neq 0\) and \(lwork < \text{max(1,2n)}\).

14. For SGEEVX and DGEEVX:

   • If \(sense = \text{‘N’ or ‘E’}\):
     – If \(jobvl = \text{‘N’ and jobvr} = \text{‘N’\}, lwork < \text{max(1, 2n)}\).
     – If \(jobvl = \text{‘V’ or jobvr} = \text{‘V’\}, lwork < 3n\).
   • If \(sense = \text{‘V’ or ‘B’\}, lwork < n(n + 6)\).

   For CGEEVX and ZGEEVX:

   • If \(sense = \text{‘N’ or ‘E’\}, lwork < \text{max(1, 2n)}\).
   • If \(sense = \text{‘V’ or ‘B’\}, lwork < n^2 + 2n\).

15. \(lwork \neq 0\) and \(lwork \neq -1\) and \(lwork < \text{the minimum required value}\).

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 [“76” on page 1257].
2. Because lwork = 0, the subroutine dynamically allocates WORK.
3. On output, A has been overwritten.

Call Statement and Input:

```fortran
CALL DGEEV( 'N', 'N', 4, A, 4, WR, WI, DUMMY1, 1, DUMMY2, 1, WORK, 0, INFO )
```

-or-

```fortran
CALL DGEEVX( 'N', 'N', 'N', 'N', 4, A, 4, WR, WI, DUMMY1, 1, DUMMY2, 1, ILO, IHI,
            SCALE, ABNRM, RCONDE, RCONDV, WORK, LWORK, IWORK, INFO )
```

A =

| -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 |

Output:

WR =

| 1.000000 |
| 2.000000 |
| 3.000000 |
| 4.000000 |

WI =

| 0.000000 |
| 0.000000 |
| 0.000000 |
| 0.000000 |

SCALE =

| 1.000000 |
| 1.000000 |
| 1.000000 |
| 1.000000 |

ILO = 1
IHI = 4
ABNRM = 11.0
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 [“76” on page 1257].
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:

```
JOBVL JOBVR N A LDA WR WI LDVL VR LDVR WORK LWORK INFO
CALL DGEEV( 'V', 'V', 4, A, 4, WR, WI, VL, 1, VR, 1, WORK, 0, INFO )
```

- or -

```
BALANC JOBVL JOBVR SENSE N A LDA WR WI LDVL VR LDVR ILO IHI
CALL DGEEVX( 'B', 'V', 'V', 'B', 4, A, 4, WR, WI, VL, 4, VR, 4, 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:

- 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:

$A = \begin{bmatrix}
1.000000 & -6.949732 & -1.320184 & 0.103510 \\
0.000000 & 2.000000 & -2.415229 & 1.002262 \\
0.000000 & 0.000000 & 3.000000 & -0.780869 \\
0.000000 & 0.000000 & 0.000000 & 4.000000
\end{bmatrix}$

$WR = \begin{bmatrix}
1.000000 \\
2.000000 \\
3.000000
\end{bmatrix}$
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 ["76" on page 1257].
2. $IWORK$ is an integer work array of size 4.
3. On output, $A$ has been overwitten by the Schur form of the balanced matrix.

Call Statement and Input:

```
BALANC JOBVL JOBVR SENSE N  A  LDA  WR  WI  VL LDVL VR LDVR ILO  IHI
CALL DGEEVX('B', 'V', 'V', 'B', 3, A,  3,  WR, WI, VL,  3, VR,  3, ILO, IHI,
SCALE  ABNRM  RCONDE  RCONDV  WORK LWORK IWORK  INFO
```

---

| 4.000000 |

$WI = |
    0.000000 |
    0.000000 |
    0.000000 |
    0.000000 |

$VL = |
  -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  -0.408248  -0.816497  -0.894427 |

$VR = |
  -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 |

$RCONDE = |
    0.087287 |
    0.053722 |
    0.096561 |
    0.282843 |

$RCONDV = |
    0.448959 |
    0.244976 |
    0.289148 |
    0.508520 |

$SCALE = |
    1.000000 |
    2.000000 |
    1.000000 |
    0.500000 |

$ILO = 1$
$IHI = 4$
$ABNRM = 7.5$
$INFO = 0$
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:

- LDVL is set to 1 to avoid an error condition.
• DUMMY1 is a placeholder for VL. VL is not used.
• DUMMY2 is a placeholder for RCONDE. RCONDE is not used.
• DUMMY3 is a placeholder for RCONDV. RCONDV is not used.

**Note:**

1. This matrix is used in Example 6.5 in referenced text ["76" on page 1257].
2. On output, A has been overwritten by the Schur form of the balanced matrix.
3. RWORK is a real array of length 8.

Call Statement and Input:

```fortran
<table>
<thead>
<tr>
<th>INFO</th>
<th>JOBVL</th>
<th>JOBVR</th>
<th>N</th>
<th>A</th>
<th>LDA</th>
<th>W</th>
<th>VL</th>
<th>LDVL</th>
<th>VR</th>
<th>LDVR</th>
<th>WORK</th>
<th>LWORK</th>
<th>RWORK</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>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALL ZGEEV( 'N', 'V', 4, A, 4, W, DUMMY1, 1, VR, 4, WORK, 0, RWORK, INFO )</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>

-or-

<table>
<thead>
<tr>
<th>INFO</th>
<th>BALANC</th>
<th>JOBVL</th>
<th>JOBVR</th>
<th>SENSE</th>
<th>N</th>
<th>A</th>
<th>LDA</th>
<th>W</th>
<th>VL</th>
<th>LDVL</th>
<th>VR</th>
<th>LDVR</th>
<th>ILO</th>
<th>IHI</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALL ZGEEVX( 'N', 'N', 'V', 'N', 4, A, 4, W, DUMMY1, 1, VR, 4, ILO, IHI, SCALE, ABNRM, RCONDE, RCONDV, WORK, LWORK, RWORK, INFO )</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>
<td></td>
</tr>
</tbody>
</table>
```

A = |
( 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) |

Output:

• For ZGEEV:

A = |
( 2.0000, 6.0000) (-4.8694, -0.6574) (-6.7837, -7.6341) (17.1461, -0.1118) |
( 0.0000, 0.0000) ( 4.0000, 8.0000) ( 0.1351, 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) |

• For ZGEEVX:

A = |
( 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) |

W = |
( 2.0000, 6.0000) |
( 4.0000, 8.0000) |
( 3.0000, 7.0000) |
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 [“76” on page 1257].
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:

```
INFO JOBVL JOBVR N A LDA W VL LDVL VR LDVR WORK LWORK RWORK
| | | | | | | | | | | |
CALL ZGEEV( 'V', 'V', 4, A, 4, W, DUMMY1, 1, VR, 4, WORK, 0, RWORK, INFO )
```

-or-

```
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 )
```

$A = \begin{pmatrix}
(5.0, 9.0) & (5.0, 5.0) & (-6.0, -6.0) & (-7.0, -7.0) \\
(3.0, 3.0) & (6.0, 10.0) & (-5.0, -5.0) & (-6.0, -6.0) \\
(2.0, 2.0) & (3.0, 3.0) & (-1.0, -1.0) & (-5.0, -5.0) \\
(1.0, 1.0) & (2.0, 2.0) & (-3.0, -3.0) & (0.0, 4.0)
\end{pmatrix}$
**Output:**

- **For ZGEEV:**

| $A$ | $(2.0000, 6.0000)$ | $(-4.8694, -0.6574)$ | $(-6.7837, -7.6341)$ | $(17.1461, -0.1118)$ |
| $(0.0000, 0.0000)$ | $(4.0000, 8.0000)$ | $(0.1351, 1.7912)$ | $(-2.7907, -2.0982)$ |
| $(0.0000, 0.0000)$ | $(0.0000, 0.0000)$ | $(3.0000, 7.0000)$ | $(3.6172, 0.1977)$ |
| $(0.0000, 0.0000)$ | $(0.0000, 0.0000)$ | $(0.0000, 0.0000)$ | $(1.0000, 5.0000)$ |

- **For ZGEEVX:**

| $A$ | $(2.0000, 6.0000)$ | $(-1.1081, 4.9368)$ | $(-3.3663, 3.6542)$ | $(-19.9524, 4.0936)$ |
| $(0.0000, 0.0000)$ | $(4.0000, 8.0000)$ | $(0.1597, 0.5962)$ | $( -2.1519, 5.6785)$ |
| $(0.0000, 0.0000)$ | $(0.0000, 0.0000)$ | $(3.0000, 7.0000)$ | $( 0.8130, 4.9939)$ |
| $(0.0000, 0.0000)$ | $(0.0000, 0.0000)$ | $(0.0000, 0.0000)$ | $( 1.0000, 5.0000)$ |

| $W$ | $(2.0000, 6.0000)$ |
| $(4.0000, 8.0000)$ |
| $(3.0000, 7.0000)$ |
| $(1.0000, 5.0000)$ |

| $VR$ | $(0.3780, 0.0000)$ | $(0.5774, 0.0000)$ | $(0.5774, 0.0000)$ | $(0.7559, 0.0000)$ |
| $(0.7559, 0.0000)$ | $(0.5774, 0.0000)$ | $(0.5774, 0.0000)$ | $(0.3780, 0.0000)$ |
| $(0.3780, 0.0000)$ | $(0.5774, 0.0000)$ | $(0.0000, 0.0000)$ | $(0.3780, 0.0000)$ |
| $(0.3780, 0.0000)$ | $(0.0000, 0.0000)$ | $(0.5774, 0.0000)$ | $(0.3780, 0.0000)$ |

| $VL$ | $(-0.5774, 0.0000)$ | $(-0.3780, 0.0000)$ | $(-0.3780, 0.0000)$ | $(0.0000, 0.0000)$ |
| $(0.0000, 0.0000)$ | $(-0.3780, 0.0000)$ | $(-0.3780, 0.0000)$ | $(0.0000, 0.0000)$ |
| $(0.5774, 0.0000)$ | $(0.3780, 0.0000)$ | $(0.7559, 0.0000)$ | $(0.3780, 0.0000)$ |

| RCONDE | 0.2182 |
| 0.2182 |
| 0.2182 |

| RCONDV | 0.3089 |
| 0.6450 |
| 0.1770 |
| 0.5504 |

| SCALE | 1.000000 |
| 1.000000 |
| 1.000000 |

| 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 ["76 on page 1257"].
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:

```
BALANC JOBVL JOBVR SENSE N  A  LDA  W  VL LDVL VR LDVR ILO  IHI
CALL ZGEEVX('B', 'V', 'V', 'B', 4, A, LDA, W, VL, 4, VR, 4, ILO, IHI,
  SCALE, ABNRM, RCONDE, RCONDV, WORK, LWORK, RWORK, INFO

A =  
  | ( 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) |
```

Output:

```
A =  
  | ( 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) |

W =  
  | ( 2.0000, 6.0000) |
  | ( 4.0000, 8.0000) |
  | ( 3.0000, 7.0000) |
  | ( 1.0000, 5.0000) |

VR =  
  | ( 0.3780, 0.0000) ( 0.5774, 0.0000) ( 0.5774, 0.0000) ( 0.7559, 0.0000) |
  | ( 0.7559, 0.0000) ( 0.3780, 0.0000) ( 0.5774, 0.0000) ( 0.3780, 0.0000) |
  | ( 0.3780, 0.0000) ( 0.0000, 0.0000) ( 0.5774, 0.0000) ( 0.3780, 0.0000) |
  | ( 0.3780, 0.0000) ( 0.0000, 0.0000) ( 0.0000, 0.0000) (  1.0000,  5.0000) |

VL =  
  | (-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) |

RCONDE =  
  | 0.1633 |
  | 0.2108 |
  | 0.2108 |
  | 0.2887 |

RCONDV =  
  | 0.4507 |
  | 0.4293 |
  | 0.1317 |
```
SSYEV, DSYEV, CHEEV, ZHEEV, SSPEVX, DSPEVX, CHPEVX, ZHPEVX, SSYEVX, DSYEVX, CHEEVX, ZHEEVX, SSYEVR, DSYEVR, CHEEVR, and ZHEEVR (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, eigenvectors are returned in matrix \( Z \). The formula is shown below:

\[
Az = wZ
\]

where \( A = A^T \) or \( A = A^H \).
### Table 205: 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>SSYEVA, SSPEVXA, SSYEVXA, ISYEVRA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DSYEVA, DSPEVXA, DSYEVA</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>CHEREVA, CHPEVXA, CHEREVXA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZHEEVA, ZHPEVXA, ZHEEVA</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>Short-precision complex</td>
<td>CHEREVA, CHPEVXA, CHEREVXA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZHEEVA, ZHPEVXA, ZHEEVA</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision complex</td>
<td>CHEREVA, CHPEVXA, CHEREVXA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ZHEEVA, ZHPEVXA, ZHEEVA</td>
</tr>
</tbody>
</table>

\[ \Delta \text{ LAPACK} \]

### Syntax

Table 206: Syntax for supported computing languages.

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SSYEV</td>
</tr>
<tr>
<td></td>
<td>CALL CHEEV</td>
</tr>
<tr>
<td></td>
<td>CALL SSPEVX</td>
</tr>
<tr>
<td></td>
<td>CALL CHPEVX</td>
</tr>
<tr>
<td></td>
<td>CALL SSYEVX</td>
</tr>
<tr>
<td></td>
<td>CALL CHEEVR</td>
</tr>
<tr>
<td></td>
<td>CALL SSYEVR</td>
</tr>
</tbody>
</table>
Table 206: Syntax for supported computing languages. (continued)

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
</table>
| C and C++ | ssyev | dsyev (jobz, uplo, n, a, lda, w, work, lwork, info); 
cheev | zheev (jobz, uplo, n, a, lda, w, work, lwork, rwork, info); 
sspevx | dspevx (jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz, work, liwork, lrwork, info); 
chpevx | zhpevx (jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz, work, rwork, lwork, iwork, ifail, info); ssyevx | dsyevx (jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, work, lwork, rwork, iwork, ifail, info); cheevx | zheevx (jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, work, lwork, rwork, iwork, ifail, info); |
| LAPACK | info = LAPACKE_ssyev | LAPACKE_dsyev (matrix_layout, jobz, uplo, n, a, lda, w); 
info = LAPACKE_cheev | LAPACKE_zheev (matrix_layout, jobz, uplo, n, a, lda, w); 
info = LAPACKE_sspevx | LAPACKE_dspevx (matrix_layout, jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz, ifail); 
info = LAPACKE_chpevx | LAPACKE_zhpevx (matrix_layout, jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz, ifail); 
info = LAPACKE_ssyevx | LAPACKE_dsyevx (matrix_layout, jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, ifail); 
info = LAPACKE_cheevx | LAPACKE_zheevx (matrix_layout, jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, ifail); |
| LAPACK | info = LAPACKE_ssyevr | LAPACKE_dsyevr (matrix_layout, jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz); |< 
info = LAPACKE_cheevr | LAPACKE_zheevr (matrix_layout, jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz); |< 

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. The value of the *matrix_layout* argument 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; \textit{jobz} = 'N' or 'V'.

\textbf{range}

indicates the type of computation to be performed, where:

If \textit{range} = 'A', all eigenvalues are to be found.
If \textit{range} = 'V', all eigenvalues in the interval \([vl, vu]\) are to be found.
If \textit{range} = 'I', the \textit{il}-th through \textit{iu}-th eigenvalues are to be found.

Specified as: a single character; \textit{range} = 'A', 'V', or 'I'.

\textbf{uplo}

indicates whether the upper or lower triangular part of the matrix \(A\) is 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'.

\textbf{n}

is the order of matrix \(A\) used in the computation.
Specified as: an integer; \(n \geq 0\).

\textbf{ap}

is the real symmetric or complex Hermitian matrix \(A\) of order \(n\). It is stored in an array, referred to as \textit{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 205 on page 868.

\textbf{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 \textit{lda} by (at least) \(n\) array, containing numbers of the data type indicated in Table 205 on page 868.

\textbf{lda}

is the leading dimension of the array specified for \(A\).
Specified as: an integer; \(lda > 0\) and \(lda \geq n\).

\textbf{vl}

has the following meaning:

If \textit{range} = 'V', it is the lower bound of the interval to be searched for eigenvalues.
If \textit{range} \neq 'V', this argument is ignored.

Specified as: a number of the data type indicated in Table 205 on page 868. If \textit{range} = 'V', \(vl < vu\).

\textbf{vu}

has the following meaning:

If \textit{range} = 'V', it is the upper bound of the interval to be searched for eigenvalues.
If \textit{range} \neq 'V', this argument is ignored.

Specified as: a number of the data type indicated in Table 205 on page 868. If \textit{range} = 'V', \(vl < vu\).

\textbf{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.

\[ \text{il} \]
Specified as: an integer; if \( \text{range} = 'I' \), \( 1 \leq \text{il} \leq \max(1, n) \).

\[ \text{iu} \]
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; if \( \text{range} = 'I' \), \( \min(\text{il}, n) \leq \text{iu} \leq n \).

\[ \text{abstol} \]
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 \( \text{abstol} \) 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 \( \text{abstol} \) 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 \( \text{jobz} = 'V' \), setting \( \text{abstol} \) to \( \text{unfl} \) (the underflow threshold) yields the most orthogonal eigenvectors.

Notes:
1. The approximate values of the constants used for \( \text{abstol} \) are listed below:

\[ \text{For SSPEVX, CHPEVX, SSYEVX, CHEEVX, SSYEVR, and CHEEVR} \]

\[ \epsilon \]
0.119209289550781250E-06

\[ \text{unfl} \]
0.1175494351E-37

\[ \sqrt{\text{unfl}} \]
0.1084202172E-18

\[ \text{For DSPEVX, ZHPEVX, DSYEVX, ZHEEVX, DSYEVR, and ZHEEVR} \]

\[ \epsilon \]
0.222044604925031308E-15

\[ \text{unfl} \]
0.222507385850720138E-307
2. The value of \texttt{abstol} can affect which algorithm is used to compute the eigenvalues and eigenvectors. See Function.

3. For \texttt{SSYEVR}, \texttt{CHEEVR}, \texttt{DSYEVR}, and \texttt{ZHEEVR}, if high relative accuracy is important, set the \texttt{abstol} argument to the safe minimum value of ~1.1\sqrt{\texttt{unfl}}\times 10^{-153}.

Specified as: a number of the data type indicated in Table 205 on page 868.

\texttt{m}

See On Return.

\texttt{w}

See On Return.

\texttt{z}

See On Return.

\texttt{ldz}

is the leading dimension of the array specified for \texttt{Z}.

Specified as: an integer; \(\text{ldz} > 0\) and, if \texttt{jobz} = 'V', \(\text{ldz} \geq n\).

\texttt{isuppz}

See On Return.

\texttt{work}

is a work area used by these subroutines.

For \texttt{SSYEV}, \texttt{DSYEV}, \texttt{CHEEV}, \texttt{ZHEEV}, \texttt{SSYEVX}, \texttt{DSYEVX}, \texttt{CHEEVX}, \texttt{ZHEEVX}, \texttt{SSYEVR}, \texttt{DSYEVR}, \texttt{CHEEVR}, and \texttt{ZHEEVR}

The size of the work area is specified by the \texttt{lwork} argument.

For \texttt{SSPEVX} and \texttt{DSPEVX}

The size of the work area is \(8n\).

For \texttt{CHPEVX} and \texttt{ZHPEVX}

The size of the work area is \(2n\).

Specified as: an area of storage; which contains numbers of the data type indicated in Table 205 on page 868.

\texttt{lwork}

is used to determine the size of the \texttt{WORK} array.

Specified as: an integer, where:

- If \texttt{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 \texttt{lwork} = -1, a workspace query is assumed. The subroutine only calculates the optimal size of the \texttt{WORK} array and returns this value as the first entry of the \texttt{WORK} array.

Otherwise:

For \texttt{SSYEV} and \texttt{DSYEV}

\(\texttt{lwork} \geq \max(1, 3n-1)\).

For \texttt{CHEEV} and \texttt{ZHEEV}

\(\texttt{lwork} \geq \max(1, 2n-1)\).

For \texttt{SSYEVX} and \texttt{DSYEVX}

\(\texttt{lwork} \geq \max(1, 8n)\).

For \texttt{CHEEVX} and \texttt{ZHEEVX}

\(\texttt{lwork} \geq \max(1, 2n)\).
For SSYEVR and DSYEVR
\[ lwork \geq \max(1, 26n). \]

For CHEEVR and ZHEEVR
\[ lwork \geq \max(1, 2n). \]

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 storage work area of size \( \max(1, 3n-2) \).

For CHPEVX, ZHPEVX, CHEEVRX, and ZHEEVRX
\( rwork \) is a storage work area of size \( 7n \).

For SSYEVR, DSYEVR, CHEEVR, and ZHEEVR
The size of the \( rwork \) storage work area is specified by the \( lrwork \) argument.

Specified as: an area of storage, which contains real numbers of the data type indicated in Table 205 on page 868.

\( lrwork \)

is used to determine the size of the \( RWORK \) array.

Specified as: an integer, where:

- If \( lrwork = 0 \), the subroutine dynamically allocates the workspace required during this computation. The dynamically allocated workspace is freed before returning the control to the calling program.
- If \( lrwork = -1 \), a workspace query is assumed. The subroutine calculates only the optimal size of the \( RWORK \) array and returns this value as the first entry of the \( RWORK \) array.
- Otherwise, for CHEEVR and ZHEEVR:

\[ lrwork \geq \max(1, 24n) \]

Note: These formulas represent the minimum workspace required. For best performance, specify either \( lrwork = -1 \) (to obtain the optimal size of workspace) or \( lrwork = 0 \) (to direct the subroutine to dynamically allocate the workspace).

\( iwork \)

is a work area.

For SSYEV, DSYEV, CHEEV, ZHEEV, SSPEVX, DSYPEVX, CHPEVX, ZHPEVX, SSYEVX, DSYEVX, CHEEVX, and ZHEEVX
\( iwork \) is a storage work area of size \( 5n \).

For SSYEVR, DSYEVR, CHEEVR, and ZHEEVR
The size of the \( iwork \) storage area is specified by the \( liwork \) argument.

Specified as: an area of storage containing integers.

\( liwork \)

is used to determine the size of the \( IWORK \) array.

Specified as: an integer, where:

- If \( liwork = 0 \), the subroutine dynamically allocates the workspace that is required during this computation. The dynamically allocated workspace is freed before returning the control to the calling program.
- If \( liwork = -1 \), a workspace query is assumed. The subroutine only calculates the optimal size of the \( IWORK \) array and returns this value as the first entry of the \( IWORK \) array.
- Otherwise, for SSYEVR, DSYEVR, CHEEVR, and ZHEEVR:

\[ liwork \geq \max(1, 10n) \]
**Note:** These formulas represent the minimum workspace required. For best performance, specify either `liwork = -1` (to obtain the optimal size of workspace) or `liwork = 0` (to direct the subroutine to dynamically allocate the workspace).

**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 205 on page 868.

**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 $A$.

For **SSYEVR, DSYEVR, CHEEVR, and ZHEEVR**

On exit, the matrix $A$ is destroyed.

Returned as: an array of dimension $lda$ by (at least) $n$, containing numbers of the data type indicated in Table 205 on page 868.

**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 $n$, which contains the computed eigenvalues in ascending order.

For **SSYEVX, SSPEVX, DSPEVX, CHPEVX, ZHPEVX, DSYEVX, CHEEVX, ZHEEVX, SSYEVR, DSYEVR, CHEEVR, and ZHEEVR**

is the vector $w$, which contains the computed eigenvalues in ascending order in the first $m$ elements of $w$.

Returned as: a one-dimensional array of (at least) length $n$, which contains numbers of the data type indicated in Table 205 on page 868.

**z**

has the following meaning, where:

If $jobz = 'N'$, $z$ is ignored.
If \( \textit{jobz} = 'V' \), 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 \( \textit{ifail} \).

**Note:** You must ensure that at least \( \max(1, m) \) columns are supplied in the array \( z \); if \( \textit{range} = 'V' \), the exact value of \( m \) is not known in advance and an upper bound must be used. Supplying \( n \) as upper bound is always safe.

Returned as: an \( \textit{ldz} \) by (at least) \( \max(1, m) \) array, containing numbers of the data type indicated in Table 205 on page 868.

\( \textit{isuppz} \) is a work area used by these subroutines.

If \( \textit{jobz} = 'V' \) and \( \textit{range} = 'A' \) or \( 'I' \) and \( iu-il = n-1 \), then the \( \textit{isuppz} \) array contains the indices that indicate the nonzero elements in the matrix \( Z \). The \( i \)-th eigenvector is nonzero only in elements \( \textit{isuppz}(2*i-1) \) through \( \textit{isuppz}(2*i) \).

If \( \textit{jobz} = 'N' \) or \( \textit{range} = 'V' \) or \( 'I' \) and \( iu-il \neq n-1 \), then the \( \textit{isuppz} \) array is not referenced.

Returned as: an integer array of dimension \( 2*\max(1, m) \).

**Notes:**
1. The \( \textit{isuppz} \) array is set only when you want to compute all eigenvalues and eigenvectors and conform to IEEE 754 standard.
2. The \( \textit{isuppz} \) array contains nonzero elements in the matrix \( Z \) before applying orthogonal matrix, which is used in reduction of the matrix to tridiagonal form. It might not be consistent with the nonzero elements in the final output of the eigenvectors in matrix \( Z \).
3. Ensure that at least \( 2*\max(1, m) \) columns are specified in the \( \textit{isuppz} \) array. The exact value of \( m \) is not known in advance and an upper bound must be used. Supplying \( 2*\max(1, n) \) columns is always safe.

\( \textit{work} \) is a work area used by these subroutines.

Returned as: an area of storage where:

If \( \textit{lwork} = -1 \), \( \textit{work} \) is (at least) of length 1 and \( \textit{work}_1 \) contains the calculated optimal size of the \( \textit{WORK} \) array.

If \( \textit{lwork} \neq -1 \) and \( \textit{lwork} \neq 0 \), \( \textit{work} \) is (at least) of length \( \textit{lwork} \) and \( \textit{work}_1 \) contains the value specified for \( \textit{lwork} \).

Except for \( \textit{work}_1 \), the contents of \( \textit{work} \) are overwritten on return.

\( \textit{rwork} \) is a work area used by these subroutines.

For \( \text{CHEEVR} \) and \( \text{ZHEEVR} \)

Returned as: an area of storage where:

- If \( \textit{lrwork} = -1 \), the \( \textit{rwork} \) array is (at least) of length 1 and the \( \textit{rwork}_1 \) entry contains the calculated optimal size of the \( \textit{RWORK} \) array.
- If \( \textit{lrwork} \neq -1 \) and \( \textit{lrwork} \neq 0 \), the \( \textit{rwork} \) array is (at least) of length \( \textit{lrwork} \) and the \( \textit{rwork}_1 \) entry contains the value specified for \( \textit{lrwork} \).

Except for the \( \textit{rwork}_1 \) value, the contents of the \( \textit{rwork} \) array are overwritten on return.

\( \textit{iwork} \)

For \( \text{SSYEVX}, \text{DSYEVX}, \text{CHEEVR} \), and \( \text{ZHEEVR} \)

Returned as: an area of storage where:
• If \(liwork = -1\), then the \(iwork\) array is (at least) of length 1 and the \(iwork_1\) entry contains the calculated optimal size of the \(IWORK\) array.

• If \(liwork = -1\) and \(liwork \neq 0\), then the \(iwork\) array is (at least) of length \(liwork\) and the \(iwork_1\) entry contains the value specified for the \(liwork\) argument.

Except for the \(iwork_1\) value, the contents of the \(iwork\) array are overwritten on return.

\(ifail\) 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:

For \(SSYEV, DSYEV, CHEEV, ZHEEV, LAPACKE_ssyev, LAPACKE_dsyev, LAPACKE_cheev, and LAPACKE_zheev\)

If \(info = 0\), then the algorithm converged. This indicates a normal exit.

If \(info = i\), the algorithm failed to converge. \(i\) off-diagonal elements of an intermediate tridiagonal form did not converge to zero.

For \(SSPEVX, DSPEVX, CHPEVX, ZHPEVX, SSYEVX, DSYEVX, CHEEVX, ZHEEVX, LAPACKE_sspevx, LAPACKE_dspevx, LAPACKE_chpevx, LAPACKE_zhpevx, LAPACKE_ssyevx, LAPACKE_dsyevx, LAPACKE_cheevx, and LAPACKE_zheevx\)

If \(info = 0\), then all eigenvectors converged. This indicates a normal exit.

If \(info = i\), then \(i\) eigenvectors failed to converge.

For \(SSYEVR, DSYEVR, CHEEVR, ZHEEVR, LAPACKE_ssyevr, LAPACKE_dsyevr, LAPACKE_cheevr, and LAPACKE_zheevr\)

If \(info = 0\), all eigenvectors converged. This value indicates a normal exit.

If \(info > 0\), the algorithm failed to converge.

• If \(jobz = 'N'\) and \(info = 1\), bisection failed to converge.

• If \(jobz = 'N'\) and \(info = 2\), not all of the eigenvalues were found.

• If \(jobz = 'N'\) and \(info = 3\), bisection failed to converge and not all of the eigenvalues were found.

• If \(jobz = 'N'\) and \(info = 4\), Gershgorin interval that was initially used was too small. No eigenvalues were computed.

• If \(jobz = 'V'\) and if \(info = i\), \(i\) eigenvectors failed to converge.

Returned as:

• For \(SSYEV, DSYEV, CHEEV, ZHEEV, SSYEVX, SSPEVX, DSPEVX, CHPEVX, ZHPEVX, DSYEVX, CHEEVX, ZHEEVX, SSYEVR, DSYEVR, CHEEVR, and ZHEEVR\):

An integer; \(info \geq 0\).

• For \(LAPACKE_ssyev, LAPACKE_dsyev, LAPACKE_cheev, LAPACKE_zheev, LAPACKE_ssyevx, LAPACKE_dspevx, LAPACKE_chpevx, LAPACKE_zhpevx, LAPACKE_dsyevx, LAPACKE_cheevx, LAPACKE_zheevx, LAPACKE_ssyevr, LAPACKE_dsyevr, LAPACKE_cheevr, and LAPACKE_zheevr\):

An integer function value; \(info \geq 0\).

Notes:

1. This subroutine accepts lowercase letters for the \(jobz\), \(range\), and \(uplo\) arguments.
2. In your C program, the arguments \( info \) and \( m \) 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. \( A, Z, w, \) \( \text{lsupp} \), \( i \text{fail} \), \( \text{work} \), \( \text{rwork} \), \( \text{iwork} \) 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 80 or “Upper-Packed Storage Mode” on page 82, respectively.

   For a description of how complex Hermitian matrices are stored in lower- or upper-packed storage mode, see “Complex Hermitian Matrix” on page 84.

6. For best performance, specify \( l \text{work} = 0 \), \( l \text{rwork} = 0 \), and \( l \text{iwork} = 0 \).

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 \( \text{jobz} = \text{‘N’} \), compute all the eigenvalues of the symmetric tridiagonal matrix using the Pal-Walker-Kahan variant of the QL or QR algorithm.
   - Otherwise, for \( \text{jobz} = \text{‘V’} \), 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 \( \text{abstol} \) and whether or not all eigenvalues are requested.
   a. If \( \text{abstol} \leq 0 \) and if all eigenvalues were requested (that is, \( \text{range} = \text{‘A’} \) or \( \text{range} = \text{‘I’} \) with \( \text{il} = 1 \) and \( \text{iu} = n \)), do the following:
      - If \( \text{jobz} = \text{‘N’} \), compute all the eigenvalues of the symmetric tridiagonal matrix using the Pal-Walker-Kahan variant of the QL or QR algorithm.
      - Otherwise, for \( \text{jobz} = \text{‘V’} \), compute all eigenvalues and eigenvectors of the symmetric tridiagonal matrix using the implicit QL or QR method.
   b. Otherwise, if \( \text{abstol} > 0 \), if a subset of the eigenvalues was requested via \( \text{range} = \text{‘I’} \) or \( \text{range} = \text{‘V’} \), 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 $\varepsilon(\text{norm}(T))$ 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 **SSYEVR, DSYEVR, CHEEVR, and ZHEEVR**

These subroutines compute selected eigenvalues, and optionally, the eigenvectors of a real symmetric or complex Hermitian matrix $A$, stored 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 checking the parameters.)

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 if all eigenvalues were requested (that is, $\text{range} = 'A'$ or $\text{range} = 'I'$, $il = 1$, and $iu = n$), complete one of the following steps:

      • If $\text{jobz} = 'N'$, compute all the eigenvalues of the symmetric tridiagonal matrix by using the Pal-Walker-Kahan variant of the QL or QR algorithm.

      • If $\text{jobz} = 'V'$, compute all eigenvalues and eigenvectors of the symmetric tridiagonal matrix by using the Multiple Relatively Robust Representations method.

   b. Otherwise, if $abstol > 0$, if a subset of the eigenvalues was requested by using $\text{range} = 'I'$ or $\text{range} = 'V'$, or if the previous step failed to compute all eigenvalues, complete the following steps:

      1) Compute the requested eigenvalues using bisection. If $abstol \leq 0$, then $\varepsilon(\text{norm}(T))$ is used in the place of $abstol$, where $\text{norm}(T)$ is the one-norm of the tridiagonal matrix obtained by reducing matrix $A$ to tridiagonal form.

      2) If the eigenvectors were also requested, compute the eigenvectors by using inverse iteration.

4. Rescale eigenvalues appropriately if matrix $A$ was scaled.

For more information on these methods, see references ["9" on page 1253], ["35" on page 1255], ["37" on page 1255], and ["76" on page 1257].

**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. 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 $\text{ifail}$. 
5. The subroutine computed the eigenvalues using multiple algorithms. Performance may be degraded.
6. If $\text{info} = i$, 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. $\text{jobz} \neq 'N'$ or 'V'
2. $\text{range} \neq 'A'$, 'V', or 'I'
3. $\text{uplo} \neq 'U'$ or 'L'
4. $n < 0$
5. $\text{range} = 'V'$, $n > 0$, and $\text{vu} \leq \text{vl}$
6. $\text{range} = 'I'$ and $(\text{il} < 1$ or $\text{il} > \text{max}(1, n))$
7. $\text{range} = 'I'$ and $(\text{iu} < \text{min}(n, \text{il})$ or $\text{iu} > n)$
8. $\text{lda} \leq 0$
9. $\text{lda} < n$
10. $\text{ldz} \leq 0$
11. $\text{jobz} = 'V'$ and $\text{ldz} < n$
12. $\text{lwork} \neq 0$ and $\text{lwork} \neq -1$ and $\text{lwork} <$ the minimum required value
13. $\text{lrwork} \neq 0$, $\text{lrwork} \neq -1$, and $\text{lrwork} <$ the minimum required value
14. $\text{liwork} \neq 0$, $\text{liwork} \neq -1$, and $\text{liwork} <$ 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 [“76” on page 1257].

Matrix $A$ is:

$$
\begin{pmatrix}
5.0 & 4.0 & 1.0 & 1.0 \\
4.0 & 5.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 4.0 & 2.0 \\
1.0 & 1.0 & 2.0 & 4.0
\end{pmatrix}
$$

Call Statement and Input:

```fortran
CALL DSPEVX ('N', 'A', 'L', 4, AP, VL, VU, IL, IU, -1.0, M, W, Z, 4, WORK, IWORK, IFAIL, INFO)
```

$\text{AP} = (5.0, 4.0, 1.0, 1.0, 5.0, 1.0, 1.0, 4.0, 2.0, 4.0)$

Eigensystem analysis subroutines
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 $range = 'I'$.

**Note:** This matrix is Example 4.1 in referenced text ["76" on page 1257].

Matrix $A$ is the same matrix used for DSPEVX in Example 1.

Call Statement and Input:

```
CALL DSPEVX ('V', 'I', 'U', 4, AP, VL, VU, 1, 3, -1.0, M, W, Z, 4, WORK, IWORK, IFAIL, INFO)
```

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 |
```

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 ["76" on page 1257].
Matrix $A$ is the same matrix used for DSPEVX in Example 1.

Call Statement and Input:

```
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 |
       |     .    |
       |     .    |
       | -0.316228 -0.632456 |
Z    =  | -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 [“76” on page 1257].

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:

```
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    =  |  1.000000 |
       |  3.000000 |
```
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 = 'I'.

Note: This matrix is Example 6.3 in referenced text ["76" on page 1257].

Matrix $A$ is the same matrix used for ZHPEVX in Example 4.

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 ('V', 'I', 'U', 3, AP, VL, VU, 1, 2, -1.0, M, W, Z, 3, WORK, RWORK, IWORK, IFAIL, INFO)
AP   =  ((2.0, . ), (0.0, 1.0), (2.0, . ), (0.0, 0.0), (0.0, 0.0), (3.0, . ))
```

Output:

```
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    =  |  1.000000 |
       |  3.000000 |
       |      .    |
       | (0.0000, -0.7071), (0.0000, 0.1591) |
       | (0.7071,  0.0000), (0.1591, 0.0000) |
       | (0.0000,  0.0000), (0.9744, 0.0000) |
```

```
Z    =  |  0.0000, -0.7071), (0.0000, 0.1591) |
       | (0.7071,  0.0000), (0.1591, 0.0000) |
       | (0.0000,  0.0000), (0.9744, 0.0000) |
```

```
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 ["76" on page 1257].

Matrix $A$ is the same matrix used for ZHPEVX in Example 4.

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 ('V', 'V', 'U', 3, AP, 2.0, 4.0, IL, IU, -1.0, M, W, Z, 3, WORK, RWORK, IWORK, IFAIL, INFO)
AP   =  ((2.0, . ), (0.0, 1.0), (2.0, . ), (0.0, 0.0), (0.0, 0.0), (3.0, . ))
```

```
W    =  |  1.000000 |
       |  3.000000 |
       |      .    |
       | (0.0000, -0.7071), (0.0000, 0.1591) |
       | (0.7071,  0.0000), (0.1591, 0.0000) |
       | (0.0000,  0.0000), (0.9744, 0.0000) |
```

```
Z    =  |  0.0000, -0.7071), (0.0000, 0.1591) |
       | (0.7071,  0.0000), (0.1591, 0.0000) |
       | (0.0000,  0.0000), (0.9744, 0.0000) |
```

```
IFAIL = (0,0,.)
INFO = 0
```

882 IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and Reference
Output:

\[ M = 2 \]

\[ \text{AP} = ((2.0, 0.0), (-1.0, 0.0), (2.0, 0.0), (0.0, 0.0), (0.0, 0.0), (3.0, 0.0)) \]

\[ \text{W} = \begin{bmatrix} 3.000000 \\ 3.000000 \\ . \end{bmatrix} \]

\[ \text{Z} = \begin{bmatrix} (0.0000, -0.6634), (0.0000, -0.2447) \\ (-0.6634, 0.0000), (-0.2447, 0.0000) \\ (-0.3460, 0.0000), (0.9382, 0.0000) \end{bmatrix} \]

\[ \text{IFAIL} = (0, 0, .) \]
\[ \text{INFO} = 0 \]

Example 7

This example shows how to find the eigenvalues only of a symmetric matrix \( A \) of order 4, where:

- DUMMY1 is a placeholder for VL. VL is not used.
- DUMMY2 is a placeholder for VU. VU is not used.
- DUMMY3 is a placeholder for IL. IL is not used.
- DUMMY4 is a placeholder for IU. IU is not used.

Notes:

1. This matrix is Example 4.1 in referenced text ["76" on page 1257].
2. Because \( lwork = 0 \), the subroutine dynamically allocates WORK.
3. Because \( lrwork = 0 \), the subroutine dynamically allocates RWORK.
4. Because \( liwork = 0 \), the subroutine dynamically allocates IWORK.

Matrix \( A \) is the same matrix used for DSPEVX in Example 1.

Call Statement and Input:

- or -

```
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', 'A', 'L', 4, A, 4, DUMMY1, DUMMY2, DUMMY3, DUMMY4, -1.0, M, W, Z, 4, WORK, 0, IWORK, IFAIL, INFO)
```

- or -

```
JOBZ RANGE UPLO N A LDA VL VU IL IU ABSTOL M W Z LDZ ISSUPZ WORK LWORK IWORK LDWORK INFO
CALL DSYEVR ('N', 'A', 'L', 4, A, 4, DUMMY1, DUMMY2, DUMMY3, DUMMY4, -1.0, M, W, Z, 4, ISSUPZ, WORK, 0, IWORK, 0, INFO)
```
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.

Notes:

1. This matrix is Example 4.1 in referenced text [“76” on page 1257].
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:

\[
\begin{align*}
\text{JOBZ} & \quad \text{UPLD} \quad \text{N} \quad \text{A} \quad \text{LDA} \quad \text{W} \quad \text{WORK} \quad \text{lwork} \quad \text{INFO} \\
\text{CALL DSYEV (’V’, ’U’, 4, A, 4, W, WORK, 0, INFO)}
\end{align*}
\]

\[
\begin{bmatrix}
5.0 & 4.0 & 1.0 & 1.0 \\
4.0 & 5.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 4.0 & 1.0 \\
1.0 & 1.0 & 2.0 & 4.0 \\
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
0.316228 & 0.316228 & 0.316228 & 0.316228 \\
0.316228 & 0.316228 & 0.316228 & 0.316228 \\
0.316228 & 0.316228 & 0.316228 & 0.316228 \\
0.316228 & 0.316228 & 0.316228 & 0.316228 \\
\end{bmatrix}
\]
Example 9

This example shows how to find the eigenvalues and eigenvectors of a real symmetric matrix \( A \) of order 4, where:

- DUMMY1 is a placeholder for VL. VL is not used.
- DUMMY2 is a placeholder for VU. VU is not used.

This example also illustrates the use of the \( il \) and \( iu \) arguments when \( \text{range} = 'I' \).

Notes:

1. This matrix is Example 4.1 in referenced text [“76” on page 1257].
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:

```fortran
CALL DSYEVX ('V', 'I', 'U', 4, A, 4, DUMMY1, DUMMY2, 1, 3, -1.0, M, W, Z, 4, WORK, 0, IWORK, IFAIL, INFO)
```

Output:

\[ M = 3 \]

\[ A = \begin{bmatrix}
1.000000 & 0.000000 & 0.414214 & 0.224745 \\
0.000000 & 6.000000 & 2.828427 & 0.224745 \\
0.000000 & 7.000000 & -2.449490 & 0.000000 \\
0.000000 & 0.000000 & 4.000000 & 0.000000 \\
\end{bmatrix} \]

\[ W = \begin{bmatrix}
1.000000 \\
2.000000 \\
5.000000 \\
\end{bmatrix} \]

\[ Z = \begin{bmatrix}
0.707107 & 0.000000 & -0.316228 \\
-0.707107 & 0.000000 & -0.316228 \\
0.000000 & 0.707107 & 0.632456 \\
0.000000 & 0.707107 & 0.632456 \\
\end{bmatrix} \]

\[ \text{IFAIL} = (0,0,0,.) \]

\[ \text{INFO} = 0 \]
Example 10
This example shows how to find the eigenvalues and eigenvectors of a real symmetric matrix $A$ of order 4, where:

- DUMMY1 is a placeholder for VL. VL is not used.
- DUMMY2 is a placeholder for VU. VU is not used.

This example also illustrates the use of the $il$ and $iu$ arguments when range = 'I'.

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.

Matrix $A$ is:

$$
\begin{pmatrix}
5.0 & 4.0 & 0.0 & 0.0 \\
4.0 & 5.0 & 1.0 & 0.0 \\
0.0 & 1.0 & 4.0 & 1.0 \\
0.0 & 0.0 & 1.0 & 4.0 \\
\end{pmatrix}
$$

Call Statement and Input:

```
CALL DSYEVR ('V', 'I', 'U', 4, A, 4, DUMMY1, DUMMY2, 1, 4, -1.0, M, W, Z, 4, ISSUPZ, WORK, 0, IWORK, 0, INFO)
```

Output:

$M = 4$

Matrix $A$ is overwritten on output.

$W = \begin{pmatrix}
0.821597 \\
3.075215 \\
5.000000 \\
9.103188
\end{pmatrix}$

$Z = \begin{pmatrix}
-0.668539 & 0.214534 & -0.174078 & 0.690455 \\
0.698357 & -0.103233 & 0.800000 & 0.708266 \\
-0.243858 & -0.659434 & 0.696311 & 0.144331 \\
0.076724 & 0.713067 & 0.696311 & 0.028283
\end{pmatrix}$

$\text{ISUPPZ} = \begin{pmatrix}
1 \\
4 \\
1 \\
4 \\
1 \\
4
\end{pmatrix}$

$\text{INFO} = 0$
Example 11

This example shows how to find the eigenvalues and eigenvectors of a real symmetric matrix $A$ of order 4, where:

- DUMMY3 is a placeholder for IL. IL is not used.
- DUMMY4 is a placeholder for IU. IU is not used.

This example also illustrates the use of the $vl$ and $vu$ arguments when range = 'V'.

Notes:

1. This matrix is Example 4.1 in referenced text ["76" on page 1257].
2. Because $lwork = 0$, the subroutine dynamically allocates WORK.
3. Because $lrwork = 0$, the subroutine dynamically allocates RWORK.
4. Because $liwork = 0$, the subroutine dynamically allocates IWORK.

Matrix $A$ is the same matrix used for DSPEVX in Example 1.

Call Statement and Input:

```
JOBZ RANGE UPLO  N  A  LDA VL   VU     IL      IU    ABSTOL M  W  Z LDZ WORK LWORK IWORK  IFAIL  INFO
CALL DSYEVX ('V', 'V',  'U', 4, A,  4, 3.0, 11.0, DUMMY3, DUMMY4, -1.0, M, W, Z, 4, WORK,  0,  IWORK, IFAIL, INFO)
```

>!-or-!<

>!

```
JOBZ RANGE UPLO  N  A  LDA VL   VU     IL       IU  ABSTOL  M  W  Z LDZ WORK LWORK IWORK  ISUPPZ INFO
CALL DSYEVR ('V', 'V',  'U', 4, A,  4, 3.0, 11.0, DUMMY3, DUMMY4, -1.0, M, W, Z, 4, WORK,  0,  IWORK, ISUPPZ, INFO)
```

<

Output:

- For DSYEVX,

```
M    =    2
```

```
A    =  | 1.000000 0.000000 0.414214  0.224745 |
      | .       6.000000 2.828427  0.224745 |
      | .       . 7.000000 -2.449490 |
      | .       .       . 4.000000 |
```

- For DSYEVR, $A$ is overwritten on output. <

```
W    =  | 5.000000 |
      | 10.000000 |
      | .        |
      | .        |
```

```
Z    =  | -0.316228 -0.632456 |
      | -0.316228 -0.632456 |
      | 0.632456 -0.316228 |
      | 0.632456 -0.316228 |
```
For DSYEVX, IFAIL = (0,0,.,.)

For DSYEVR, ISUPPZ is not used when RANGE = 'V'.

INFO = 0

Example 12
This example shows how to find the eigenvalues only of a complex Hermitian matrix A of order 3, where:

• DUMMY1 is a placeholder for VL. VL is not used.
• DUMMY2 is a placeholder for VU. VU is not used.
• DUMMY3 is a placeholder for IL. IL is not used.
• DUMMY4 is a placeholder for IU. IU is not used.

Notes:
1. This matrix is Example 6.3 in referenced text [“76” on page 1257].
2. Because lwork = 0, the subroutine dynamically allocates WORK.
3. Because lrwork = 0, the subroutine dynamically allocates RWORK.
4. Because liwork = 0, the subroutine dynamically allocates IWORK.
5. For ZHEEVX, matrix A is the same matrix used for ZHPEVX in Example 4.

Call Statement and Input:

\[
\begin{align*}
\text{JOBZ} & \quad \text{UPLO} \quad \text{N} \quad \text{A} \quad \text{LDA} \quad \text{W} \quad \text{WORK} \quad \text{LWORK} \quad \text{RWORK} \quad \text{INFO} \\
\text{CALL ZHEEV} ('N', 'L', 3, A, 3, W, WORK, 0, RWORK, INFO)
\end{align*}
\]

–or–

\[
\begin{align*}
\text{JOBZ} & \quad \text{RANGE} \quad \text{UPLO} \quad \text{N} \quad \text{A} \quad \text{LDA} \quad \text{VL} \quad \text{VU} \quad \text{IL} \quad \text{IU} \quad \text{ABSTOL} \quad \text{M} \quad \text{W} \quad \text{Z} \quad \text{LDZ} \quad \text{WORK} \quad \text{LWORK} \quad \text{RWORK} \quad \text{IWORK} \quad \text{IFAIL} \quad \text{INFO} \\
\text{CALL ZHEEVX} ('N', 'A', 'L', 3, A, LDA, DUMMY1, DUMMY2, DUMMY3, DUMMY4, -1.0, M, W, Z, 3, WORK, 0, RWORK, IWORK, IFAIL, INFO)
\end{align*}
\]

| ( 2.0, 0.0) | ( 0.0, -1.0) (2.0, 0.0) | . |
| ( 0.0, 0.0) (0.0, 0.0) (3.0, 0.0) |

Output:

M = 3

• For ZHEEV, A is overwritten on output.
• For ZHEEVX,

\[
A = \begin{bmatrix}
(2.0, 0.0) & (0.0, 1.0) & (0.0, 0.0) \\
(0.0, 0.0) & (2.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (0.0, 0.0) & (3.0, 0.0)
\end{bmatrix}
\]

• For ZHEEVR, A is overwritten on output.

\[
W = \begin{bmatrix}
1.000000 \\
3.000000 \\
3.000000
\end{bmatrix}
\]

Z is not used when JOBZ = 'N'.

• For ZHEEVR, ISUPPZ is not used when JOBZ = 'N'.

For ZHEEVX, IFAIL is not used when JOBZ = 'N'.

INFO = 0

Example 13

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. This matrix is Example 6.3 in referenced text ["76" on page 1257].
2. Because \( lwork = 0 \), the subroutine dynamically allocates WORK.
3. Because \( lrwork = 0 \), the subroutine dynamically allocates RWORK.
4. Because \( liwork = 0 \), the subroutine dynamically allocates IWORK.

Matrix \( A \) is the same matrix used for ZHPEVX in Example 4.

Call Statement and Input:

\[
\begin{array}{cccccccc}
& \text{JOBZ} & \text{UPLO} & \text{N} & \text{A} & \text{LDA} & \text{W} & \text{WORK} & \text{LWORK} & \text{RWORK} & \text{INFO} \\
\text{CALL ZHEEV ('V', 'U', 3, A, 3, W, WORK, 0, RWORK, INFO)}
\end{array}
\]

\[
A = \begin{bmatrix}
(2.0, 0.0) & (0.0, 1.0) & (0.0, 0.0) \\
(0.0, 0.0) & (2.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (0.0, 0.0) & (3.0, 0.0)
\end{bmatrix}
\]

Output:

\[
A = \begin{bmatrix}
(0.0000, 0.7071) & (0.0000, 0.7071) & (0.0000, 0.0000) \\
(-0.7071, 0.0000) & (0.7071, 0.0000) & (0.0000, 0.0000) \\
(0.0000, 0.0000) & (0.0000, 0.0000) & (1.0000, 0.0000)
\end{bmatrix}
\]

\[
W = \begin{bmatrix}
1.000000 \\
3.000000 \\
3.000000
\end{bmatrix}
\]

INFO = 0
Example 14

This example shows how to find the eigenvalues and eigenvectors of a complex Hermitian matrix $A$ of order 3, where:

- DUMMY1 is a placeholder for VL. VL is not used.
- DUMMY2 is a placeholder for VU. VU is not used.

This example also illustrates the use of the $il$ and $iu$ arguments when $range = 'I'$.

Notes:

1. This matrix is Example 6.3 in the referenced text ["76" on page 1257].
2. Because $lwork = 0$, the subroutine dynamically allocates WORK.
3. Because $lrwork = 0$, the subroutine dynamically allocates RWORK.
4. Because $liwork = 0$, the subroutine dynamically allocates IWORK.

Matrix $A$ is the same matrix used for ZHPEVX in Example 4.

Call Statement and Input:

```
JOBZ RANGE UPLO N A LDA VL VU IL IU ABSTOL M W Z LDZ WORK LWORK RWORK IWORK IFAIL INFO
CALL ZHEEVX ('V', 'I', 'U', 3, A, LDA, DUMMY1, DUMMY2, 1, 2, -1.0, M, W, Z, 3, WORK, 0, RWORK, IWORK, IFAIL, INFO)
```

Output:

- $M = 2$
- $A = \begin{bmatrix}
(2.0, 0.0) & (-1.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (2.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (0.0, 0.0) & (3.0, 0.0)
\end{bmatrix}$
- $W = \begin{bmatrix}
1.000000 \\
3.000000 \\
. \\
\end{bmatrix}$
- $Z = \begin{bmatrix}
(0.0000, -0.7071), (0.0000, 0.1591) \\
(0.7071, 0.0000), (0.1591, 0.0000) \\
(0.0000, 0.0000), (0.9744, 0.0000)
\end{bmatrix}$
- $IFAIL = (0,0,.)$
- $INFO = 0$

Example 15

This example shows how to find the eigenvalues and eigenvectors of a complex Hermitian matrix $A$ of order 3, where:

- DUMMY1 is a placeholder for VL. VL is not used.
- DUMMY2 is a placeholder for VU. VU is not used.

This example also illustrates the use of the $il$ and $iu$ arguments when $range = 'I'$.

Notes:
1. This matrix is Example 6.3 in referenced text ["76" on page 1257].
2. Because \( lwork = 0 \), the subroutine dynamically allocates WORK.
3. Because \( lrwork = 0 \), the subroutine dynamically allocates RWORK.
4. Because \( liwork = 0 \), the subroutine dynamically allocates IWORK.

Matrix \( A \) is the same matrix used in Example 4.

Call Statement and Input:

```
CALL ZHEEVR ('V', 'I', 'U', 3, A, LDA, DUMMY1, DUMMY2, 1, 3, -1.0, M, W, Z, 3, ISUPPZ, WORK, 0, RWORK, 0, IWORK, 0, INFO)
```

Output:

\[ M = 3 \]

Matrix \( A \) is overwritten on output.

\[
W = \begin{bmatrix}
1.000000 \\
3.000000 \\
3.000000
\end{bmatrix}
\]

\[
Z = \begin{bmatrix}
(0.0000, -0.7071), (0.0000, 0.7071), (0.0000, 0.0000) \\
(0.7071, 0.0000), (0.7071, 0.0000), (0.0000, 0.0000) \\
(0.0000, 0.0000), (0.0000, 0.0000), (1.0000, 0.0000)
\end{bmatrix}
\]

\[
\text{ISUPPZ} = \begin{bmatrix}
1 \\
2 \\
1 \\
2 \\
3 \\
3
\end{bmatrix}
\]

\[ \text{INFO} = 0 \]

**Example 16**

This example shows how to find the eigenvalues and eigenvectors of a complex Hermitian matrix \( A \) of order 3, where:

- \( \text{DUMMY3} \) is a placeholder for IL. IL is not used.
- \( \text{DUMMY4} \) is a placeholder for IU. IU is not used.

This example also illustrates the use of the \( vl \) and \( vu \) arguments when \( \text{range} = 'V' \).

**Notes:**

1. This matrix is Example 6.3 in referenced text ["76" on page 1257].
2. Because \( lwork = 0 \), the subroutine dynamically allocates WORK.
3. Because \( lrwork = 0 \), the subroutine dynamically allocates RWORK.
4. Because \( liwork = 0 \), the subroutine dynamically allocates IWORK.

Matrix \( A \) is the same matrix used for ZHPEVX in Example 4.
Call Statement and Input:

```
JOBZ RANGE UPLO 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, DUMMY3, DUMMY4, -1.0, M, W, Z, 3, WORK, 0, RWORK, IWORK, IFAIL, INFO)
```

-or-

```
JOBZ RANGE UPLO N  A  LDA  VL  VU  IL  IU  ABSTOL M  W  Z  LDZ ISUPPZ WORK LWORK RWORK LWORK IWORK LRWORK INFO
CALL ZHEEVR ('V', 'V', 'U', 3, A, LDA, 2.0, 4.0, DUMMY3, DUMMY4, -1.0, M, W, Z, 3, ISUPPZ, WORK, 0, RWORK, 0, IWORK, 0, INFO)
```

Output:

\[
M = 2
\]

• For ZHEEVX,

\[
A = \begin{pmatrix}
(2.0, 0.0) & (-1.0, 0.0) & (0.0, 0.0) \\
. & (2.0, 0.0) & (0.0, 0.0) \\
. & . & (3.0, 0.0)
\end{pmatrix}
\]

• For ZHEEVR, A is overwritten on output.

\[
W = \begin{pmatrix}
3.000000 \\
3.000000 \\
\end{pmatrix}
\]

• For ZHEEVX,

\[
Z = \begin{pmatrix}
(0.0000, 0.6634), & (0.0000, 0.2447) \\
(-0.6634, 0.0000), & (-0.2447, 0.0000) \\
(-0.3460, 0.0000), & (0.9382, 0.0000)
\end{pmatrix}
\]

• For ZHEEVR

\[
Z = \begin{pmatrix}
(0.0000,-0.6634), & (0.0000, -0.2447) \\
(-0.6634, 0.0000), & (-0.2447, 0.0000) \\
(-0.3460, 0.0000), & (0.9382, 0.0000)
\end{pmatrix}
\]

For ZHEEVX, IFAIL = (0, 0, 0).

>!For ZHEEVR, ISUPPZ is not used when RANGE = 'V'.

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 207: 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*&lt;br&gt;SSYEVD*</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DSPEVD*&lt;br&gt;DSYEVD*</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>Short-precision complex</td>
<td>CHPEVD*&lt;br&gt;CHEEVD*</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision complex</td>
<td>ZHPEVD*&lt;br&gt;ZHEEVD*</td>
</tr>
</tbody>
</table>

\*LAPACK
<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fortran</strong></td>
<td>CALL SSPEVD</td>
</tr>
<tr>
<td></td>
<td>CALL CHPEVD</td>
</tr>
<tr>
<td></td>
<td>CALL SSYEVD</td>
</tr>
<tr>
<td></td>
<td>CALL CHEEVD</td>
</tr>
<tr>
<td><strong>C and C++</strong></td>
<td>sspevd</td>
</tr>
<tr>
<td></td>
<td>chpevd</td>
</tr>
<tr>
<td></td>
<td>ssyevd</td>
</tr>
<tr>
<td></td>
<td>cheevd</td>
</tr>
<tr>
<td><strong>LAPACKE</strong></td>
<td>info = LAPACKE_sspevd</td>
</tr>
<tr>
<td></td>
<td>info = LAPACKE_chpevd</td>
</tr>
<tr>
<td></td>
<td>info = LAPACKE_ssyevd</td>
</tr>
<tr>
<td></td>
<td>info = LAPACKE_cheevd</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. The value of the `matrix_layout` argument 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 ≥ 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 207 on page 893.

**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 207 on page 893.

**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 207 on page 893.

**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 SSYEVX and DSYEVX
\( lwork \) 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 207 on page 893.

\( lrwork \)
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 \( liwork = 0 \) and \( lrwork \neq -1 \) 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 \( liwork = -1 \) or \( lrwork \neq -1 \) or \( lwork = -1 \), 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 \( n \leq 1 \), \( liwork \) must be (at least) 1
- If \( jobz = 'N' \) and \( n > 1 \), \( liwork \) must be (at least) 1
- If \( jobz = 'V' \) and \( n > 1 \), \( lwork \) must be (at least) \( 3+ 5n \)

**On Return**

- 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 207 on page 893.

- If \( jobz = 'V' \) and \( info = 0 \), \( a \) contains the orthonormal eigenvectors corresponding to the computed eigenvalues, with the \( i \)-th column of \( A \) holding the eigenvector associated with \( w_i \).
  - If \( jobz = 'V' \) and \( info \neq 0 \), no eigenvectors are valid.
  - If \( jobz = 'N' \):
    - If \( uplo = 'U' \), the upper triangle of matrix \( A \) is overwritten.
    - If \( uplo = 'L' \), the lower triangle of matrix \( A \) is overwritten.

- Returned as: an array of dimension \( lda \) by (at least) \( n \), containing numbers of the data type indicated in Table 207 on page 893.

- If \( info = 0 \), \( w \) is the vector \( w \), containing the computed eigenvalues in ascending order.
  - If \( info \neq 0 \), no eigenvalues are valid.

- Returned as: a one-dimensional array of (at least) length \( n \), containing numbers of the data type indicated in Table 207 on page 893.

- \( z \) has the following meaning, where:
  - If \( jobz = 'N' \), then \( z \) is ignored.
  - If \( jobz = 'V' \) and \( info = 0 \), \( z \) contains the orthonormal eigenvectors corresponding to the computed eigenvalues, with the \( i \)-th column of \( z \) holding the eigenvector associated with \( w_i \).
  - If \( jobz = 'V' \) and \( info \neq 0 \), no eigenvectors are valid.

- Returned as: an \( ldz \) by \( n \) array, containing numbers of the data type indicated in Table 207 on page 893.

- **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 \( \text{work}_1 \) is set to the optimal \( lwork \) value and contains numbers of the data type indicated in Table 207 on page 893.
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 lrwork ≥ 1 or lrwork = -1 or liwork = -1 or lwork = -1, then rwork₁ is set to the optimal lrwork value and contains numbers of the data type indicated in Table 207 on page 893.

Except for rwork₁, 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 ≥ 1 or liwork = -1 or lrwork = -1 or lwork = -1, then iwork₁ is set to the optimal liwork value and contains numbers of the data type indicated in Table 207 on page 893.

Except for iwork₁, 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 mod(info, n + 1). No eigenvalues are valid.

Returned as:

• For SSPEVD, DSPEVD, CHPEVD, ZHPEVD, SSYEV, DSYEV, CHEEVD, and ZHEEVD returned as: an integer; info ≥ 0.

• For LAPACKE_sspevd, LAPACKE_dspevd, LAPACKE_chpevd, LAPACKE_zhpevd, LAPACKE_ssyevd, LAPACKE_dsyevd, LAPACKE_cheevd, and LAPACKE_zheevd returned as an integer function value; info ≥ 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 80 or “Upper-Packed Storage Mode” on page 82, respectively.

   For a description of how complex Hermitian matrices are stored in lower- or upper-packed storage mode, see “Complex Hermitian Matrix” on page 84.

6. For a description of how real symmetric matrices are stored in lower or upper storage mode, see “Lower Storage Mode” on page 83 or “Upper Storage Mode” on page 83, respectively.

   For a description of how complex Hermitian matrices are stored in lower or upper storage mode, see “Complex Hermitian Matrix” on page 84.

7. For best performance specify lwork = 0, liwork = 0, and lrwork = 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.
- 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 $A = A^H$.

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 $jobz$:
   - If $jobz = 'N'$, 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 [“9” on page 1253], [“76” on page 1257], and [“35” on page 1255].

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 mod($info$, $n + 1$). No eigenvalues are valid.
Input-Argument Errors

1. \texttt{jobz} ≠ 'N' or 'V'
2. \texttt{uplo} ≠ 'U' or 'L'
3. \texttt{n} < 0
4. \texttt{lda} ≤ 0
5. \texttt{n} > \texttt{lda}
6. \texttt{ldz} ≤ 0
7. \texttt{n} > \texttt{ldz} and \texttt{jobz} = 'V'
8. \texttt{lwork} ≠ 0 and \texttt{lwork} ≠ -1 and \texttt{liwork} ≠ -1 and \texttt{lrwork} ≠ -1, and \texttt{lwork} < the minimum required value
9. \texttt{liwork} ≠ 0 and \texttt{liwork} ≠ -1 and \texttt{lwork} ≠ -1 and \texttt{lrwork} ≠ -1, and \texttt{liwork} < the minimum required value
10. \texttt{lrwork} ≠ 0 and \texttt{lrwork} ≠ -1 and \texttt{liwork} ≠ -1 and \texttt{lwork} ≠ -1 and \texttt{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 \texttt{lwork} = 0, the subroutine dynamically allocates WORK.
2. Because \texttt{liwork} = 0, the subroutine dynamically allocates IWORK.
3. Z is not used when \texttt{jobz} = 'N'.
4. On output, array AP is overwritten.
5. This matrix is Example 4.1 in referenced text [“76” on page 1257].

Matrix A is:

\[
\begin{pmatrix}
5.0 & 4.0 & 1.0 & 1.0 \\
4.0 & 5.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 4.0 & 2.0 \\
1.0 & 1.0 & 2.0 & 4.0 \\
\end{pmatrix}
\]

Call Statement and Input:

\[
\text{CALL DSPEVD ('N', 'L', 4, AP, W, Z, 4, WORK, 0, IWORK, 0, INFO)}
\]

\[
\text{AP} = (5.0, 4.0, 1.0, 1.0, 5.0, 1.0, 1.0, 4.0, 2.0, 4.0)
\]

Output:

\[
\begin{pmatrix}
1.000000 \\
2.000000 \\
5.000000 \\
10.000000 \\
\end{pmatrix}
\]

INFO = 0

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 [“76” on page 1257].

Matrix \( A \) is the same as in Example 1.

Call Statement and Input:

\[
\begin{align*}
\text{JOBZ} & \quad \text{UPLO} \quad \text{N} \quad \text{AP} \quad \text{W} \quad \text{Z} \quad \text{LDZ} \quad \text{WORK} \quad \text{LWORK} \quad \text{IWORK} \quad \text{LIWORK} \quad \text{INFO} \\
\text{CALL} & \quad \text{DSPEVD} \quad (\text{‘V’}, \quad \text{‘U’}, \quad 4, \quad \text{AP}, \quad \text{W}, \quad \text{Z}, \quad 4, \quad \text{WORK}, \quad 0, \quad \text{IWORK}, \quad 0, \quad \text{INFO}) \\
\text{AP} & \quad = \quad (5.0, 4.0, 5.0, 1.0, 1.0, 4.0, 1.0, 1.0, 2.0, 4.0) \\
\end{align*}
\]

Output:

\[
\begin{align*}
\text{W} & \quad = \quad \begin{bmatrix} 1.000000 \\ 2.000000 \\ 5.000000 \\ 10.000000 \end{bmatrix} \\
\text{Z} & \quad = \quad \begin{bmatrix} -0.707107 & 0.000000 & 0.316228 & -0.632456 \\ 0.707107 & 0.000000 & 0.316228 & -0.632456 \\ 0.000000 & -0.707107 & -0.632456 & -0.316228 \\ 0.000000 & 0.707107 & -0.632456 & -0.316228 \end{bmatrix} \\
\text{INFO} & \quad = \quad 0
\end{align*}
\]

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 \( jobz = \text{‘N’} \).
5. On output, array AP is overwritten.
6. This matrix is Example 4.1 in referenced text [“76” on page 1257].

Matrix \( A \) is:

\[
\begin{bmatrix}
( 2.0, 0.0) & ( 0.0, 1.0) & ( 0.0, 0.0) \\
( 0.0, -1.0) & ( 2.0, 0.0) & ( 0.0, 0.0) \\
( 0.0, 0.0) & ( 0.0, 0.0) & ( 3.0, 0.0)
\end{bmatrix}
\]

Call Statement and Input:

\[
\begin{align*}
\text{JOBZ} & \quad \text{UPLO} \quad \text{N} \quad \text{AP} \quad \text{W} \quad \text{Z} \quad \text{LDZ} \quad \text{WORK} \quad \text{LWORK} \\
\text{CALL} & \quad \text{ZHPEVD} \quad (\text{‘N’}, \quad \text{‘L’}, \quad 3, \quad \text{AP}, \quad \text{W}, \quad \text{Z}, \quad 3, \quad \text{WORK}, \quad 0, \\
\text{RWORK} & \quad \text{LRWORK} \quad \text{IWORK} \quad \text{LIWORK} \quad \text{INFO} \\
\text{RWORK} & \quad 0, \quad \text{IWORK}, \quad 0, \quad \text{INFO}) \\
\end{align*}
\]
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 \( IW0RK \).
4. On output, array \( AP \) is overwritten.
5. This matrix is Example 4.1 in referenced text ["76" on page 1257].

Matrix \( A \) is the same as in Example 3.

Call Statement and Input:

```
CALL ZHPEVD ( 'V', 'U', 3, AP, W, Z, 3, WORK, 0, RWORK, 0, IWORK, 0, INFO )
```

Output:

```
W       =  
          |  1.000000 |
          |  3.000000 |
          |  3.000000 |
```

```
Z        =  
       ( -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 )
```

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 \( IW0RK \).
3. On output, array \( A \) is overwritten.
4. This matrix is Example 4.1 in referenced text ["76" on page 1257].
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:

```fortran
CALL DSYEVD ('N', 'L', 4, A, 4, W, WORK, 0, IWORK, 0, INFO)
```

Output:

```fortran
W = | 1.000000 |
    | 2.000000 |
    | 5.000000 |
    | 10.000000 |
```

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 [“76” on page 1257].

Matrix $A$ is the same as in Example 5.

Call Statement and Input:

```fortran
CALL DSYEVD ('V', 'U', 4, A, 4, W, WORK, 0, IWORK, 0, INFO)
```

Output:

```fortran
A = | 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 |
```

W = | 1.000000 |
    | 2.000000 |
    | 5.000000 |

Eigensystem analysis subroutines 903
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 ["76" on page 1257].

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:

```
CALL ZHEEVD ('N', 'L', 3, A, 3, W, WORK, 0, RWORK, 0, IWORK, 0, INFO)
```

Output:

\[
W = \begin{pmatrix}
1.000000 \\
3.000000 \\
3.000000
\end{pmatrix}
\]

INFO = 0

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 IWORK.
4. This matrix is Example 4.1 in referenced text ["76" on page 1257].

Matrix \( A \) is the same as in Example 7.
Call Statement and Input:

```plaintext
CALL ZHEEVD ('V', 'U', 3, A, 3, W, WORK, 0, RWORK, 0, IWORK, 0, INFO)
```

\[
A = \begin{pmatrix}
2.0 & 0.0 & 0.0 \\
0.0 & 1.0 & 0.0 \\
0.0 & 0.0 & 0.0 \\
\end{pmatrix}
\]

Output:

\[
A = \begin{pmatrix}
-0.7071 & 0.0 & 0.0 \\
0.0 & -0.7071 & 0.0 \\
0.0 & 0.0 & 1.0 \\
\end{pmatrix}
\]

\[
W = \begin{pmatrix}
1.000000 \\
3.000000 \\
3.000000 \\
\end{pmatrix}
\]

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:

\[ v_i^H A = \lambda_i v_i^H B \]

For the right eigenvectors:

\[ A v_i = \lambda_i B v_i \]

The eigenvalues are returned in two parts, \( \alpha \) and \( \beta \), where:

For SGGEV, DGGEV, SGGEVX, and DGGEVX, \( \alpha \) and \( \beta \) are returned in vectors `alphar`, `alphai`, and `beta`, where `alphar` contains the real part of \( \alpha \) and `alphai` contains the imaginary part of \( \alpha \).

- If `alphai_j` = 0, then the \( j \)-th eigenvalue is real.
- If `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 `alpha` and `beta`.

For SGGEV, DGGEV, SGGEVX, and DGGEVX:

- If `alphai_j` = 0, then the \( j \)-th eigenvalue is real:
  - \( \alpha_j = \text{alphar}_j \)
  - \( \beta_j = \text{beta}_j \)
- If `alphai_j` > 0, then the \( j \)-th and \( (j+1) \)-th eigenvalues \( \alpha_{j+1} \) are a complex conjugate pair:
  - \( \alpha_j = (\text{alphar}_j, \text{alphai}_j) \)
\[ \beta_j = \beta_j \]
\[ \alpha_{j+1} = (\alpha_{j+1}, -\alpha_j) \]
\[ \beta_{j+1} = \beta_{j+1} \]

For CGGEV, ZGGEV, CGGEVX, and ZGGEVX:

- \( \alpha_j = \alpha_j \)
- \( \beta_j = \beta_j \)

Left eigenvectors are returned in matrix \( VL \) and right eigenvectors are returned in matrix \( VR \).

**Table 209: Data Types.**

<table>
<thead>
<tr>
<th>( A, B, VL, VR, bwork, work \alpha, \beta )</th>
<th>( \text{alphar, alphai, lscale, rscale, abnrm, rconde, rcondv, rwork} )</th>
<th>Subroutine</th>
</tr>
</thead>
</table>
| Short-precision real | Short-precision real | SGGEV\(^\Delta\)  
SGGEVX\(^\Delta\) |
| Long-precision real | Long-precision real | DGGEV\(^\Delta\)  
DGGEVX\(^\Delta\) |
| Short-precision complex | Short-precision real | CGGEV\(^\Delta\)  
CGGEVX\(^\Delta\) |
| Long-precision complex | Long-precision real | ZGGEV\(^\Delta\)  
ZGGEVX\(^\Delta\) |

\(^\Delta\) LAPACK

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fortran</strong></td>
<td>CALL SGGEV</td>
</tr>
<tr>
<td></td>
<td>CALL CGGEV</td>
</tr>
<tr>
<td></td>
<td>CALL SGGEVX</td>
</tr>
<tr>
<td></td>
<td>CALL CGGEVX</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.

**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.

---

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>`sggev</td>
<td>dggev (jobvl, jobvr, n, a, lda, b, ldb, alphar, alpha, beta, vl, ldvl, vr, ldvr, work, lwork, info);`</td>
</tr>
<tr>
<td>`cggev</td>
<td>zggev (jobvl, jobvr, n, a, lda, b, ldb, alpha, beta, vl, ldvl, vr, ldvr, work, lwork, rwork, info);`</td>
</tr>
<tr>
<td>`sggevx</td>
<td>dggevx (balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alphar, alpha, beta, vl, ldvl, vr, ldvr, iilo, ihi, iscale, rscale, abnrm, bbnrm, rconde, rcondv, work, lwork, iwork, bwork, info);`</td>
</tr>
<tr>
<td>`cggevx</td>
<td>zggevx (balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alpha, beta, vl, ldvl, vr, ldvr, iilo, ihi, lscale, rscale, abnrm, bbnrm, rconde, rcondv, work, lwork, rwork, iwork, bwork, info);`</td>
</tr>
</tbody>
</table>

For **LAPACKE**:

- `info = LAPACK_SGGEV | LAPACK_DGGEV (matrix_layout, jobvl, jobvr, n, a, lda, b, ldb, alphar, alpha, beta, vl, ldvl, vr, ldvr);`
- `info = LAPACK_CGGEV | LAPACK_ZGGEV (matrix_layout, jobvl, jobvr, n, a, lda, b, ldb, alpha, beta, vl, ldvl, vr, ldvr);`
- `info = LAPACK_SGGEVX | LAPACK_DGGEVX (matrix_layout, balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alphar, alpha, beta, vl, ldvl, vr, ldvr, iilo, ihi, iscale, rscale, abnrm, bbnrm, rconde, rcondv);`
- `info = LAPACK_CGGEVX | LAPACK_ZGGEVX (matrix_layout, balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alpha, beta, vl, ldvl, vr, ldvr, iilo, ihi, lscale, rscale, abnrm, bbnrm, rconde, rcondv);`
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'.

**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 209 on page 906.

**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 209 on page 906.

**ldb**
is the leading dimension of the array specified for matrix B.
Specified as: an integer; \( ldb > 0 \) and \( ldb \geq n \).

**alphar**
See On Return.

**alphai**
See On Return.
**alpha**  
See On Return.

**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** ≥ **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** ≥ **n**.

**ilo**  
See On Return.

**ihi**  
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 209 on page 906.

**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** ≥ max(1, 8n)
- For CGGEV and ZGGEV:  
  - **lwork** ≥ max(1, 2n)
For SGGEVX and DGGEVX:
- If sense = 'N':
  - If balanc = 'N' or 'P' and jobvl = 'N' and jobvr = 'N', lwork ≥ max(1,2n);
    Otherwise, lwork ≥ max(1,6n)
  - If balanc = 'S' or 'B' or if jobvl = 'V' or jobvr = 'V', lwork ≥ max(1, 6n)
  - If sense = 'N', lwork ≥ max(1, 2n)
  - If sense = 'E', lwork ≥ max(1, 10n)
  - If sense = 'V' or 'B', lwork ≥ 2n²+8n+16

For CGGEVX and ZGGEVX:
- If sense = 'N', lwork ≥ max(1, 2n)
- If sense = 'E', lwork ≥ max(1, 4n)
- If sense = 'V' or 'B', lwork ≥ max(1,2n(n+1))

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 209 on page 906.

**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 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 209 on page 906.

**bwork**
is a storage work area of size n. If sense = 'N', bwork is not referenced by the subroutine.
Specified as: an area of storage containing numbers of the data type indicated in Table 209 on page 906.

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 SGGEVX, DGGEVX, CGGEVX, and ZGGEVX, if jobvl = 'V' or jobvr = 'V' and balanc ≠ 'N', A contains the Schur form of the balanced matrices.
Returned as: an lda by (at least) n array, containing numbers of the data type indicated in Table 209 on page 906.

**b**
is the updated general matrix B of order n. On output, B is overwritten; that is, the original input is not preserved.
- For SGGEVX, DGGEVX, CGGEVX, and ZGGEVX, if jobvl = 'V' or jobvr = 'V' and 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 209 on page 906.

\textit{alphar}

is the vector of length \( n \), containing the real part of the numerators of the eigenvalues. For details, see “Function” on page 914.

Returned as: an array of (at least) length \( n \), containing numbers of the data type indicated in Table 209 on page 906.

\textit{alphai}

is the vector of length \( n \), containing the imaginary part of the numerators of the eigenvalues. For details, see “Function” on page 914.

Returned as: an array of (at least) length \( n \), containing numbers of the data type indicated in Table 209 on page 906.

\textit{alpha}

is the vector \( \alpha \) of length \( n \), containing the numerators of the eigenvalues. For details, see “Function” on page 914.

Returned as: an array of (at least) length \( n \), containing numbers of the data type indicated in Table 209 on page 906.

\textit{beta}

is the vector \( \beta \) of length \( n \), containing the denominators of the eigenvalues. For details, see “Function” on page 914.

Returned as: an array of (at least) length \( n \), containing numbers of the data type indicated in Table 209 on page 906.

\( \text{vl} \)

contains the left eigenvectors.

• For S\textsc{ggev}, D\textsc{ggev}, S\textsc{ggevx}, and D\textsc{ggevx}:

  – If \( \textit{jobvl} = \text{'V'} \), 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 \( \text{vl} \) 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 \( \text{vl} \) 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 \( \textit{jobvl} = \text{'N'} \), \( \text{vl} \) is not referenced.

• For C\textsc{ggev}, Z\textsc{ggev}, C\textsc{ggevx}, and Z\textsc{ggevx}:

  – If \( \textit{jobvl} = \text{'V'} \), the left eigenvectors are stored in the columns of \( \text{vl} \), in the same order as their eigenvalues.
  – If \( \textit{jobvl} = \text{'N'} \), \( \text{vl} \) is not referenced.

Returned as: an array of size \( (\text{ldvl}, n) \) containing numbers of the data type indicated in Table 209 on page 906.

\( \text{vr} \)

contains the right eigenvectors.

• For S\textsc{ggev}, D\textsc{ggev}, S\textsc{ggevx}, and D\textsc{ggevx}:

  – If \( \textit{jobvr} = \text{'V'} \), the right eigenvectors are stored in the columns of \( \text{vr} \), in the same order as their eigenvalues.
    - If the \( j \)-th eigenvalue is real or complex, then the \( j \)-th column of \( \text{vr} \) 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 \( \text{vr} \) 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 \( \text{jobvr} = 'N', \) \( vr \) is not referenced.

- For CGGEV, ZGGEV, CGGEVX, and ZGGEVX:
  - If \( \text{jobvr} = 'V', \) the right eigenvectors are stored in the columns of \( vr \), in the same order as their eigenvalues.
  - If \( \text{jobvr} = 'N', \) \( vr \) is not referenced.

Returned as: an array of size \((ldvr, n)\) containing numbers of the data type indicated in Table 209 on page 906.

**ilo**

has the following meaning:

If \( \text{balanc} = 'N', \) \( ilo = 1. \)

Otherwise, the value of \( ilo \) is determined when \( A \) is balanced.

The balanced \( a_{ij} = 0 \) if \( i > j \) and \( j = 1, \ldots, (ilo-1) \) or \( i = (ihi+1), \ldots, n. \)

Returned as: an integer; \( 1 \leq ilo \leq n. \)

**ihi**

has the following meaning:

If \( \text{balanc} = 'N', \) \( ihi = n. \)

Otherwise, the value of \( ihi \) is determined when \( A \) is balanced.

The balanced \( a_{ij} = 0 \) if \( i > j \) and \( j = 1, \ldots, (ilo-1) \) or \( i = (ihi+1), \ldots, n. \)

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 \( j = 1, \ldots, (ilo-1) \)
- \( lscale_{j} = dl_{j}, \) for \( j = ilo, \ldots, ihi \)
- \( lscale_{j} = pl_{j}, \) for \( j = (ihi+1), \ldots, n \)

Returned as: a one-dimensional array of (at least) length \( n \) containing numbers of the data type indicated in Table 209 on page 906.

**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 \( j = 1, \ldots, (ilo-1) \)
- \( rscale_{j} = dr_{j}, \) for \( j = ilo, \ldots, ihi \)
- \( rscale_{j} = pr_{j}, \) for \( j = (ihi+1), \ldots, n \)

Returned as: a one-dimensional array of (at least) length \( n \) containing numbers of the data type indicated in Table 209 on page 906.

**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 209 on page 906; \( abnrm \geq 0. \)
**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 209 on page 906; \( bbnrm \geq 0 \).

**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 209 on page 906.

**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 209 on page 906.

**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 209 on page 906.

Except for \( work_1 \), the contents of \( work \) are overwritten on return.

**info**

If \( info = 0 \), the subroutine completed successfully.

If \( 1 \leq info \leq n \), 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_sggevx, and LAPACKE_dggevx, \( \alpha_{p_j}, \alpha_{ai_j}, \beta_j \) are valid for \( j = (info + 1, \ldots, n) \)
- For CGGEV, ZGGEV, CGGEVX, ZGGEVX, LAPACKE_cggev, LAPACKE_zggev, LAPACKE_cggevx, and LAPACKE_zggevx, \( \alpha_j \) and \( \beta_j \) are valid for \( j = (info + 1, \ldots, n) \)

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.

Returned as:

- For SGGEV, DGGEV, CGGEV, ZGGEV, SGGEVX, DGGEVX, and ZGGEVX, returned as: an integer; \( info \geq 0 \).
- For LAPACKE_sggev, LAPACKE_dggev, LAPACKE_cggev, LAPACKE_zggev, LAPACKE_sggevx, LAPACKE_dggevx, LAPACKE_cggevx, and LAPACKE_zggevx returned as an integer function value; \( info \geq 0 \).

**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 \textit{jobvr} = 'N', you must specify a dummy argument for \textit{vr}.
5. These subroutines accept lowercase letters for the \textit{jobvl} and \textit{jobvr} arguments.
6. When you specify \textit{sense} = 'N', you must specify a dummy argument for \textit{rconde}.
7. When you specify \textit{sense} = 'N' or 'E', you must specify dummy arguments for \textit{rcondv} and \textit{iwork}.
8. In your C program, the \textit{ilo}, \textit{ihi}, \textit{abnrm}, \textit{info} arguments must be passed by reference.
9. In your C program, the \textit{info} arguments must be passed by reference.
10. For best performance, specify \textit{lwork} = 0.
11. The eigenvalue quotients might easily over- or underflow, and \( \beta \) might be zero. However, \( \alpha \) is always less than and usually comparable with NORM(A) in magnitude, and \( \beta \) is always less than and usually comparable with NORM(B) in magnitude.

\textbf{Function}

- \textbf{SGGEV, DGGEV, CGGEV, and ZGGEV} compute the eigenvalues and, optionally, the left and/or right eigenvectors of a general matrix generalized eigenproblem.
- \textbf{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:
\[ \nu_h^H A = \lambda \nu_h^H B \]
For the right eigenvectors:
\[ A \nu_r = \lambda B \nu_r \]

The eigenvalues are returned in two parts, \( \alpha \) and \( \beta \), where:

- For \textbf{SGGEV, DGGEV, SGGEVX, and DGGEVX}, \( \alpha \) and \( \beta \) are returned in vectors \textit{alphar}, \textit{alphai}, and \textit{beta}, where \textit{alphar} contains the real part of \( \alpha \) and \textit{alphai} 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 \textbf{CGGEV, ZGGEV, CGGEVX, and ZGGEVX}, \( \alpha \) and \( \beta \) are returned in vectors \textit{alpha} and \textit{beta}.

For \textbf{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:
  - \( \alpha_j = (\text{alphar}_j, \text{alphai}_j) \)
  - \( \beta_j = \text{beta}_j \)
  - \( \alpha_{j+1} = (\text{alphar}_j, -\text{alphai}_j) \)
  - \( \beta_{j+1} = \text{beta}_j \)

For \textbf{CGGEV, ZGGEV, CGGEVX, and ZGGEVX}:
- \( \alpha_j = \text{alpha}_j \)
- \( \beta_j = \text{beta}_j \)

Left eigenvectors are returned in matrix \textbf{VL} and right eigenvectors are returned in matrix \textbf{VR}.

For \textbf{SGGEV, DGGEV, CGGEV, and ZGGEV}, the computation involves the following steps:

1. If necessary, scale the matrices \textbf{A} and \textbf{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 ["15" on page 1254], ["16" on page 1254], ["49" on page 1255], ["53" on page 1256], ["57" on page 1256], ["67" on page 1257], ["68" on page 1257], ["71" on page 1257], ["75" on page 1257], ["76" on page 1257], ["87" on page 1257], ["88" on page 1257], ["102" on page 1258], ["108" on page 1258], ["118" on page 1259], ["120" on page 1259], and ["121" on page 1259].

**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. \(balanc \notin \{\text{N}', \text{S}', \text{P}', \text{B}'\}\)
2. \(jobvl \notin \{\text{N}', \text{V}'\}\)
3. \(jobvr \notin \{\text{N}', \text{V}'\}\)
4. \(sense \notin \{\text{N}', \text{E}', \text{V}', \text{B}'\}\)
5. \(n < 0\)
6. \(lda \leq 0\)
7. \(n > lda\)
8. \(ldb \leq 0\)
9. \(n > ldb\)
10. \( ldvl \leq 0 \)
11. \( n > ldvl \) and \( jobvl = 'V' \)
12. \( ldvr \leq 0 \)
13. \( n > ldvr \) and \( jobvr = 'V' \)
14. \( jobvl = 'V' \) and \( ldvl \leq 0 \)
15. \( jobvr = 'V' \) and \( ldvr \leq 0 \)
16. \( jobvl = 'V' \) and \( ldvl < n \)
17. \( jobvr = 'V' \) and \( ldvr < n \)
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. \( ldvl \) and \( ldvr \) are 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:

```
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   | 10.0  | 1.0  | 2.0  |
|     | 1.0   | 2.0  | -1.0 |
|     | 1.0   | 1.0  | 2.0  |

| B   | 1.0  | 2.0  | 3.0  |
|     | 4.0  | 5.0  | 6.0  |
|     | 7.0  | 8.0  | 9.0  |

Output:

| ALPHAR | 2.092346 |
|        | -4.789188 |
|        | 4.490731  |

| ALPHAI | 0.000000  |
|        | 0.000000  |
|        | 0.000000  |

| BETA   | 12.711351 |
|        | 0.998541  |
Example 2

This example shows how to find the eigenvalues and the left and right eigenvectors of a long-precision real generalized eigenproblem \((\mathbf{A}, \mathbf{B})\).

Note:

1. On output, matrices \(\mathbf{A}\) and \(\mathbf{B}\) are overwritten.
2. Because \(lwork = 0\), the subroutine dynamically allocates \(\text{WORK}\).
3. This matrix is used on page 263 in referenced text [“6” on page 1253].

Call Statement and Input:

```plaintext
CALL DGGEV( 'V', 'V', 5, A, 5, B, 5, ALPHAR, ALPHAI, BETA, VL, 1, VR, 1, WORK, 0, INFO)
```

\[
\begin{bmatrix}
2.0 & 3.0 & 4.0 & 5.0 & 6.0 \\
4.0 & 4.0 & 5.0 & 6.0 & 7.0 \\
0.0 & 3.0 & 6.0 & 7.0 & 8.0 \\
0.0 & 0.0 & 2.0 & 8.0 & 9.0 \\
0.0 & 0.0 & 0.0 & 1.0 & 10.0
\end{bmatrix}
\]

\[
\begin{bmatrix}
1.0 & -1.0 & -1.0 & -1.0 & -1.0 \\
0.0 & 1.0 & -1.0 & -1.0 & -1.0 \\
0.0 & 0.0 & 0.0 & 1.0 & -1.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 1.0
\end{bmatrix}
\]

Output:

\[
\begin{bmatrix}
7.950050 \\
2.149669 \\
10.987556 \\
0.000000 \\
0.374183
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.000000 \\
0.000000 \\
0.000000 \\
0.000000 \\
0.000000
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.374183 \\
1.480299 \\
1.636872 \\
1.213574 \\
0.988837
\end{bmatrix}
\]

\[
\begin{bmatrix}
-0.003801 & 1.000000 & -0.778812 & -0.032145 & -0.005114 \\
-0.018291 & -0.546838 & 0.133707 & -0.028432 & -0.012900 \\
-0.074427 & -0.174281 & 1.000000 & 0.076908 & -0.009062 \\
-0.279354 & -0.051285 & -0.696402 & 0.285338 & 0.123783
\end{bmatrix}
\]
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:

```
INFO   JOBVVL JOBVR  N  A  LDA  B  LDB  ALPHAR  ALPHAI  BETA  VL  LDVL  VR  LDVR  WORK  LWORK  RWORK
|     |    |  |   |   |   |     |       |      |    |    |   |   |     |     |     |      |
CALL ZGGEV( 'N', 'N',  4, A,  4,  B,  4, ALPHAR, ALPHAI, BETA, VL,  1, VR,  1, WORK,   0,  RWORK, INFO)
```

\[
A = \begin{bmatrix}
(2, 4) & (1, 6) & (2, 8) & (4, 4) \\
(3, 3) & (6, 1) & (5, 3) & (8, 0) \\
(5, 1) & (8, 5) & (3, 2) & (8, 5) \\
(7, 6) & (3, 7) & (2, 1) & (5, 4) \\
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
(0, 0) & (1, 0) & (0, 0) & (0, 0) \\
(0, 0) & (0, 0) & (1, 0) & (0, 0) \\
(0, 0) & (0, 0) & (0, 0) & (1, 0) \\
(1, 0) & (0, 0) & (0, 0) & (0, 0) \\
\end{bmatrix}
\]

Output:

\[
\text{ALPHA} = \begin{bmatrix}
(15.8864, 15.0474) \\
(7.0401, 2.0585) \\
(1.7083, 4.1133) \\
(-3.6348, -1.2193) \\
\end{bmatrix}
\]

\[
\text{BETA} = \begin{bmatrix}
(1.0, 0.0) \\
(1.0, 0.0) \\
(1.0, 0.0) \\
(1.0, 0.0) \\
\end{bmatrix}
\]

\[
\text{INFO} = 0
\]

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 [“6” on page 1253].

Call Statement and Input:

```
INFO JOBVL JOBVR N A LDA B LDB ALPHAR ALPHA1 BETA VL LDVL VR LDVR WORK LWORK RWORK INFO
CALL ZGGEV('V', 'V', 3, A, 3, B, 3, ALPHAR, ALPHA1, BETA, VL, 3, VR, 3, WORK, 0, RWORK, INFO)
```

$A =$

\[
\begin{bmatrix}
1.0 & 2.0 & 3.0 & 4.0 & 21.0 & 22.0 \\
43.0 & 44.0 & 13.0 & 14.0 & 15.0 & 16.0 \\
5.0 & 6.0 & 7.0 & 8.0 & 25.0 & 26.0 \\
2.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 1.0 & 2.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 3.0 & 0.0 \\
\end{bmatrix}
\]

$B =$

\[
\begin{bmatrix}
2.0 & 0.0 & 0.0 & -1.0 & 0.0 & 0.0 \\
0.0 & 1.0 & 2.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 3.0 & 0.0 \\
1.0 & 2.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 2.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 3.0 & 0.0 & 0.0 & 0.0 \\
\end{bmatrix}
\]

Output:

$\text{ALPHA} =$

\[
\begin{bmatrix}
15.863783 & 41.115283 \\
-12.917205 & 19.973815 \\
3.215518 & -4.912439 \\
1.668461 & 0.0 \\
2.024212 & 0.0 \\
2.664836 & 0.0 \\
\end{bmatrix}
\]

$\text{BETA} =$

\[
\begin{bmatrix}
0.0634 & -0.8686 & 0.8988 & -0.1012 & -0.6456 & -0.3544 \\
-0.3652 & -0.3826 & 0.0198 & -0.3479 & 0.0601 & -0.0852 \\
0.3605 & -0.6395 & -0.2829 & 0.6348 & 0.4537 & 0.4128 \\
0.0634 & -0.8686 & 0.8988 & -0.1012 & -0.6456 & -0.3544 \\
-0.3652 & -0.3826 & 0.0198 & -0.3479 & 0.0601 & -0.0852 \\
0.3605 & -0.6395 & -0.2829 & 0.6348 & 0.4537 & 0.4128 \\
\end{bmatrix}
\]

$\text{VL} =$

\[
\begin{bmatrix}
0.3799 & -0.1986 & 0.2712 & -0.9943 & -0.1470 & 0.1422 \\
0.0132 & -0.9868 & -0.5085 & -0.4915 & 0.3239 & -0.6761 \\
-0.0976 & -0.2383 & -0.0633 & 0.1388 & -0.0395 & 0.1663 \\
\end{bmatrix}
\]

$\text{VR} =$

\[
\begin{bmatrix}
0.15863783 & 0.41115283 \\
-0.12917205 & 0.19973815 \\
0.03215518 & -0.04912439 \\
0.16684610 & 0.0 \\
0.20242123 & 0.0 \\
0.26648360 & 0.0 \\
\end{bmatrix}
\]

$\text{INFO} =$

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:

```
BALANC JOBVL JOBVR SENSE N A LDA B LDB ALPHAR ALPHA1 BETA VL LDVL VR LDVR WORK ILO IHI
```

Eigensystem analysis subroutines 919
CALL DGGEVX ('P', 'V', 'V', 'B', 6, A, 6, B, 6, ALPHAR, ALPHAI, BETA, VL, 6, VR, 6, ILO, IHI, LSCALE, RScales)

ABNRM, BBNRM, RCONDE, RCONDV, WORK, IWORK, BWORK, INFO)

Output:

A =

\[
\begin{bmatrix}
9.0 & 98.0 & 1.0 & 96.0 & 30.0 & 94.0 \\
1.0 & 92.0 & 91.0 & 90.0 & 8.0 & 88.0 \\
66.0 & 1.0 & 68.0 & 0.0 & 40.0 & 1.0 \\
0.0 & 1.0 & 64.0 & 50.0 & 66.0 & 1.0 \\
10.0 & 0.0 & 10.0 & 86.0 & 8.0 & 10.0 \\
8.0 & 1.0 & 20.0 & 12.0 & 20.0 & 15.0 \\
\end{bmatrix}
\]

B =

\[
\begin{bmatrix}
21.0 & 24.0 & 9.0 & 22.0 & 25.0 & 18.0 \\
19.0 & 9.0 & 20.0 & 25.0 & 15.0 & 6.0 \\
5.0 & 4.0 & 9.0 & 0.0 & 2.0 & 21.0 \\
18.0 & 7.0 & 8.0 & 24.0 & 9.0 & 1.0 \\
15.0 & 20.0 & 9.0 & 1.0 & 14.0 & 24.0 \\
21.0 & 23.0 & 9.0 & 14.0 & 88.0 & 88.0 \\
\end{bmatrix}
\]

ALPHAR =

\[
\begin{bmatrix}
64.6956 \\
55.0897 \\
28.2595 \\
-0.0472 \\
-0.0437 \\
13.8550 \\
\end{bmatrix}
\]

ALPHAI =

\[
\begin{bmatrix}
70.4260 \\
-59.9693 \\
0.0000 \\
28.9869 \\
-26.7715 \\
6.0000 \\
\end{bmatrix}
\]

BETA =

\[
\begin{bmatrix}
15.2997 \\
13.8280 \\
15.2267 \\
26.2667 \\
24.3216 \\
47.4385 \\
\end{bmatrix}
\]

VL =

\[
\begin{bmatrix}
-0.8391 & 0.3207 & -0.5181 & 0.5129 & 0.1237 & 0.1230 \\
-0.1545 & 0.6666 & 0.0489 & -0.4876 & 0.0269 & -0.0401 \\
-0.5621 & 0.6337 & -1.0000 & -0.3687 & 0.5350 & 0.1137 \\
-0.2946 & 0.2303 & -0.5078 & 0.4858 & -0.2545 & 0.1577 \\
-0.4076 & 0.5085 & -0.4921 & -0.1974 & -0.0241 & 0.0910 \\
-0.7701 & 0.8407 & -1.0000 & -0.4524 & 0.0892 & 0.9064 \\
\end{bmatrix}
\]

VR =

\[
\begin{bmatrix}
0.8937 & -0.6723 & -0.1299 & -0.2564 & -0.3186 & 0.3349 \\
0.1862 & 0.8314 & 0.5842 & 0.0377 & -0.0345 & -0.0368 \\
1.0000 & 0.4950 & 0.1818 & -0.5869 & 0.0451 & -0.3523 \\
-0.9722 & 0.0642 & -0.0423 & 0.1502 & 0.1791 & 0.0818 \\
-0.0277 & -0.1835 & -0.0507 & -0.1421 & -0.1836 & 0.2469 \\
\end{bmatrix}
\]
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, LSCALE,
RSSCALE, ABNRM, BBNRM, RCONDE, RCONDV, WORK, LWORK, RWORK, IWORK, BWORK, INFO)
```

**A**

\[
\begin{bmatrix}
10.0 & 1.0 \\
1.0 & 2.0 \\
1.0 & 0.0
\end{bmatrix}
\]

**B**

\[
\begin{bmatrix}
1.0 & 0.0 \\
4.0 & 1.0 \\
7.0 & 5.0
\end{bmatrix}
\]

**Output:**

**ALPHA**

\[
\begin{bmatrix}
-6.6742 & -0.5632 \\
2.1073 & 1.6591 \\
1.2922 & -3.5963
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.4561 & 0.0000
\end{bmatrix}
\]
SSYGVD, DSYGVD, CHEGVD, ZHEGVD (Eigenvalues, and Optionally, the Eigenvectors of a Positive Definite Real Symmetric or Complex Hermitian Generalized Eigenproblem Using a Divide-and-Conquer Algorithm)

Purpose

These subroutines compute eigenvalues, and optionally, the eigenvectors of a positive definite real symmetric or complex Hermitian generalized eigenproblem by using a divide-and-conquer algorithm that solves one of the following problems:

- 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:

- \( A \) represents the real symmetric or complex Hermitian matrix \( A \).
- \( B \) represents the positive definite real symmetric or complex Hermitian matrix \( B \).

The data type for each array and matrix, and the corresponding subroutines are listed in the following table:

<table>
<thead>
<tr>
<th>( A, B, \text{work} )</th>
<th>( w, \text{rwork} )</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>SSYGVD(^A)</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>DSYGVD(^A)</td>
</tr>
</tbody>
</table>
Table 210: Data types. (continued)

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Short-precision complex</th>
<th>Short-precision real</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHEGVDΔ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZHEGVDΔ</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ΔLAPACK

Syntax

Table 211: Syntax for supported computing languages.

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
</table>
| Fortran  | CALL SSYGVD | DSYGVD (itype, jobz, uplo, n, a, b, ldb, w, work, lwork, iwork, liwork, info)  
               CALL CHEGVD | ZHEGVD (itype, jobz, uplo, n, a, b, ldb, w, work, lwork, rwork, lrwork, iwork, liwork, info) |
| C and C++ | ssygvd | dsygvd (itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, iwork, liwork, info);  
               chegvd | zhegvd (itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, rwork, lrwork, iwork, liwork, info); |
| LAPACKE | info = LAPACKE_ssygvd | LAPACKE_dsygvd (matrix_layout, itype, jobz, uplo, n, a, lda, b, ldb, w);  
               info = LAPACKE_chegvd | LAPACKE_zhegvd (matrix_layout, itype, jobz, uplo, n, a, lda, b, ldb, w); |

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. The value of the **matrix_layout** argument must be LAPACK_ROW_MAJOR or LAPACK_COL_MAJOR.

**itype**
specifies the problem type, where:

- If **itype** = 1, the problem is **Ax = λBx**.
- If **itype** = 2, the problem is **ABx = λx**.
- If **itype** = 3, the problem is **BAx = λx**.

Specified as: an integer; **itype** = 1, 2, or 3.

**jobz**
indicates the type of computation to be performed, where:

- If **jobz** = 'N', only eigenvalues are computed.
- If **jobz** = 'V', both eigenvalues and eigenvectors are computed.
Specified as: a single character; \( jobz = 'N' \) or \( 'V' \).

**uplo**

indicates whether the upper or lower triangular parts of the matrices \( A \) and \( B \) are 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 matrices \( A \) and \( B \) that is used in the computation.
Specified as: an integer; \( n \geq 0 \).

**a**
is the real symmetric or complex Hermitian matrix \( A \) of order \( n \).
If \( uplo = 'U' \), the matrix is stored in upper storage mode.
If \( uplo = 'L' \), the matrix is stored in lower storage mode.
Specified as: an \( lda \) by (at least) \( n \) array, which contains numbers of the data type indicated in Table 210 on page 922.

**lda**
is the leading dimension of the array specified for the matrix \( A \).
Specified as: an integer; \( lda > 0 \) and \( lda \geq n \).

**b**
is the positive definite real symmetric or complex Hermitian matrix \( B \) of order \( n \).
If \( uplo = 'U' \), the matrix is stored in upper storage mode.
If \( uplo = 'L' \), the matrix is stored in lower storage mode.
Specified as: an \( ldb \) by (at least) \( n \) array, which contains numbers of the data type indicated in Table 210 on page 922.

**ldb**
is the leading dimension of the array specified for the matrix \( B \).
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 that contains numbers of the data type indicated in Table 210 on page 922.

**lwork**
is the number of elements in array \( WORK \).
Specified as: an integer; where:
- If \( lwork = -1 \), the subroutine performs a workspace query and returns the optimal required size of the \( WORK \) array in the \( work_1 \) entry. No computation is performed and the subroutine returns after checking for errors.
- If \( lwork = 0 \), the subroutine dynamically allocates the workspace that is required during this computation. The work area is deallocated before returning the control to the calling program.
- If \( lwork > 0 \):
For SSYGVD and DSYGVD:

If \( n \leq 1 \), \( lwork \geq 1 \).
If \( jobz = 'N' \) and \( n > 1 \), \( lwork \geq 2n + 1 \).
If \( jobz = 'V' \) and \( n > 1 \), \( lwork \geq 1 + 6n + 2n^2 \).

For CHEGVD and ZHEGVD:

If \( n \leq 1 \), \( lwork \geq 1 \).
If \( jobz = 'N' \) and \( n > 1 \), \( lwork \geq n + 1 \).
If \( jobz = 'V' \) and \( n > 1 \), \( lwork \geq 2n + n^2 \).

These formulas represent the minimum workspace that is required. For best performance, specify either \( lwork = -1 \) (to obtain the optimal required size) or \( lwork = 0 \) (to direct the subroutine to dynamically allocate the workspace).

**rwork**

See “On Return” on page 925.

**lrwork**

is the number of elements in array RWORK.

Specified as: an integer; where:

- If \( lrwork = -1 \), the subroutine performs a workspace query and returns the optimal required size of the RWORK array in the work1 entry.
- If \( lrwork = 0 \), the subroutine dynamically allocates the workspace that is required during this computation. The work area is deallocated before returning the control to the calling program.
- If \( lrwork > 0 \):
  - If \( n \leq 1 \), \( lrwork \geq 1 \).
  - If \( jobz = 'N' \) and \( n > 1 \), \( lrwork \geq n \).
  - If \( jobz = 'V' \) and \( n > 1 \), \( lrwork \geq 1 + 5n + 2n^2 \).

**iwork**

See “On Return” on page 925.

**liwork**

is the number of elements in array IWORK.

Specified as: an integer; where:

- If \( liwork = -1 \), the subroutine performs a workspace query and returns the optimal required size of the IWORK array in the work1 entry.
- If \( liwork = 0 \), the subroutine dynamically allocates the workspace that is required during this computation. The work area is deallocated before returning the control to the calling program.
- If \( liwork > 0 \):
  - If \( n \leq 1 \), \( liwork \geq 1 \).
  - If \( jobz = 'N' \) and \( n > 1 \), \( liwork \geq 1 \).
  - If \( jobz = 'V' \) and \( n > 1 \), \( liwork \geq 3 + 5n \).

**info**

See “On Return” on page 925.

**On Return**

\( a \)

is overwritten as follows:
• If \( \text{jobz} = 'V' \) and if \( \text{info} = 0 \), the matrix \( A \) contains the matrix \( Z \) of eigenvectors. The eigenvectors are normalized as follows:
  
  - if \( \text{itype} = 1 \) or 2, \( Z^H B Z = I \);
  
  - if \( \text{itype} = 3 \), \( Z^H B^{-1} Z = I \).

Returned as: an array of dimension \( lda \) by (at least) \( n \), which contains numbers of the data type indicated in Table 210 on page 922.

\( b \)

contains the results of the Cholesky factorization.

If \( \text{uplo} = 'U' \), and if \( \text{info} \leq n \), the leading \( n \) by \( n \) upper triangular part of matrix \( B \) contains the triangular factor \( U \) from the Cholesky factorization \( B = U^H U \).

If \( \text{uplo} = 'L' \), and if \( \text{info} \leq n \), the leading \( n \) by \( n \) lower triangular part of matrix \( B \) contains the triangular factor \( L \) from the Cholesky factorization \( B = LL^H \).

Returned as: an array of dimension \( ldb \) by (at least) \( n \), which contains numbers of the data type indicated in Table 210 on page 922.

\( w \)

is the vector \( w \), which contains the computed eigenvalues.

If \( \text{info} = 0 \), the eigenvalues are in ascending order.

Returned as: a one-dimensional array of (at least) length \( n \), which contains numbers of the data type indicated in Table 210 on page 922.

\( \text{work} \)

is a work area used by this subroutine for computation.

If \( \text{lwork} \neq 0 \) and \( \text{info} = 0 \), the \( \text{work}_1 \) entry in the \text{WORK} array returns the optimal \( \text{lwork} \) value.

Returned as: an area of storage, where:

- If \( \text{lwork} \geq 1 \) or \( \text{lwork} = -1 \), the \( \text{work}_1 \) entry is set to the optimal \( \text{lwork} \) value and all other elements of the \( \text{work} \) argument are overwritten.

\( \text{rwork} \)

is a work area used by this subroutine for computation.

If \( \text{lrwork} \neq 0 \) and \( \text{info} = 0 \), the subroutine returns the optimal \( \text{lrwork} \) value.

Returned as: an area of storage of dimension max (1, \( \text{lrwork} \)).

\( \text{iwork} \)

is a work array.

If \( \text{liwork} \neq 0 \) and \( \text{info} = 0 \), the \( \text{iwork}_1 \) entry returns the optimal \( \text{liwork} \) value.

Returned as: an integer array of dimension max (1, \( \text{liwork} \)).

\( \text{info} \)

has the following meaning:

If \( \text{info} = 0 \), the subroutine completed successfully.

If \( \text{info} \leq n \), and

- If \( \text{info} = i \) and \( \text{jobz} = 'N' \), the algorithm failed to converge. The \( i \) off-diagonal elements of an intermediate tridiagonal form did not converge to zero.

- If \( \text{info} = i \) and \( \text{jobz} = 'V' \), the algorithm failed to compute an eigenvalue while working on the submatrix located in rows and columns \( \text{info}/(n + 1) \) through \( \text{mod}(\text{info}, n + 1) \).

If \( \text{info} > n \) and if \( \text{info} = n + i \), for \( 1 \leq i \leq n \), the leading minor of order \( i \) of matrix \( B \) is not positive definite. The factorization of matrix \( B \) cannot be completed and eigenvalues or eigenvectors are not computed.

Returned as: an integer; \( \text{info} \geq 0 \).
Notes:
1. These subroutines accept lowercase letters for the jobz and uplo arguments.
2. In your C program, the info argument must be passed by reference.
3. a, b, w, work, rwork, and iwork must have no common elements; otherwise, results are unpredictable.
4. For a description of how real symmetric matrices are stored in lower or upper storage mode, see “Lower Storage Mode” on page 83 or “Upper Storage Mode” on page 83.
   For a description of how complex Hermitian matrices are stored in lower or upper storage mode, see “Complex Hermitian Matrix” on page 84.
5. On input, the imaginary parts of the diagonal elements of the complex Hermitian matrices A and B are assumed to be zero. Therefore, do not set these values. On output, for matrix B, they are set to zero.
6. For best performance, specify either lwork = -1 (to obtain the optimal required size) or lwork = 0 (to direct the subroutine to dynamically allocate the workspace).

Function
These subroutines compute all eigenvalues, and optionally, the eigenvectors of a positive definite real symmetric or complex Hermitian generalized eigenproblem by using the divide-and-conquer algorithm that solves one of the following problems:
• If itype = 1, the problem is \( Ax = \lambda Bx \)
• If itype = 2, the problem is \( ABx = \lambda x \)
• If itype = 3, the problem is \( BAx = \lambda x \)
In the formulas:
• A represents the real symmetric or complex Hermitian matrix A.
• B represents the positive definite real symmetric or complex Hermitian matrix B.
If eigenvectors are required, the subroutine uses a divide-and-conquer algorithm. The computation involves the following steps:
1. Compute the Cholesky factorization of matrix B.
2. Reduce the generalized eigenproblem to the following standard symmetric eigenvalue problem, which is solved for the eigenvalues, and optionally, the eigenvectors.
3. Backtransform the eigenvectors to obtain the eigenvectors of the original problem.

Error conditions
Resource Errors
1. lwork = 0 and unable to allocate work area.

Computational Errors
1. The matrix B is not positive definite. See output argument info for more details.
2. The algorithm failed to converge.
3. The algorithm failed to compute an eigenvalue while working on the submatrix.

Input-Argument Errors
1. itype < 1 or itype > 3
2. jobz ≠ 'N' or 'V'
3. uplo ≠ 'U' or 'L'
4. n < 0
5. lda ≤ 1
6. \( \text{lda} < n \)
7. \( \text{ldb} \leq 1 \)
8. \( \text{ldb} < n \)
9. \( \text{lwork} \neq 0 \) and \( \text{lwork} \neq -1 \) and \( \text{lwork} < \) the minimum required value
10. \( \text{lrwork} \neq 0 \) and \( \text{lrwork} \neq -1 \) and \( \text{lrwork} < \) the minimum required value
11. \( \text{liwork} \neq 0 \) and \( \text{liwork} \neq -1 \) and \( \text{liwork} < \) the minimum required value
12. 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 of a real symmetric positive generalized eigenproblem of the form: \( Ax = \lambda Bx \). Matrices \( A \) and \( B \) are stored in upper storage mode.

Matrix \( A \) is:

\[
\begin{bmatrix}
10 & 1 & 2 \\
. & 2 & -1 \\
. & . & 2
\end{bmatrix}
\]

Matrix \( B \) is:

\[
\begin{bmatrix}
1 & 0 & 0 \\
. & 1 & 0 \\
. & . & 1
\end{bmatrix}
\]

Call Statement and Input:

```
ITYPE JOBZ UPLO  N  A  LDA B  LDB  W  WORK  LWORK  IWORK  LIWORK  INFO
|     |    |   |  |  |   |  |   |   |     |      |     |     |
CALL DSYGVD ( 1, 'N', 'U', 3, A, 3, B, 3, W, WORK, 0, IWORK, 0, INFO)
```

Output:

Matrix \( A \) is overwritten.

\[
W = \begin{bmatrix}
0.5153 \\
2.9466 \\
10.5381
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
1 & 0 & 0 \\
. & 1 & 0 \\
. & . & 1
\end{bmatrix}
\]

INFO = 0

**Example 2**

This example shows how to find the eigenvalues and eigenvectors of a real symmetric positive generalized eigenproblem of the form: \( Ax = \lambda Bx \).

Matrix \( A \) is:

\[
\begin{bmatrix}
6 & 4 & 4 \\
. & 6 & 1
\end{bmatrix}
\]
Matrix $B$ is:

$$
\begin{bmatrix}
1 & 0 & 0 \\
. & 1 & 0 \\
. & . & 1 \\
\end{bmatrix}
$$

Call Statement and Input:

```
CALL DSYGVD ( 2, 'V', 'U', 3, A, 3, B, 3, W, WORK, 0, IWORK, 0, INFO)
```

Output:

$$
\begin{bmatrix}
0.8210916 \\
5.0000000 \\
12.178908
\end{bmatrix}
$$

$$
\begin{bmatrix}
-0.737578 & 0.4774816 & 0.4774816 \\
0.0000000 & -0.707106 & 0.7071067 \\
-0.675261 & -0.521546 & -0.521546 \\
\end{bmatrix}
$$

$$
\begin{bmatrix}
1 & 0 & 0 \\
. & 1 & 0 \\
. & . & 1 \\
\end{bmatrix}
$$

INFO = 0

**Example 3**

This example shows how to find the eigenvalues of a real symmetric positive generalized eigenproblem of the form: $BAx = \lambda x$. Matrices $A$ and $B$ are stored in upper storage mode.

Matrix $A$ is:

$$
\begin{bmatrix}
(10.0, 1.0) & (1.0, 0) & (2.0, -1) \\
( . ) & (2.0, -1) & (1.0, 0) \\
( . ) & ( . ) & (2.0, 1.5) \\
\end{bmatrix}
$$

Matrix $B$ is:

$$
\begin{bmatrix}
(1.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) \\
( . ) & (1.0, 0.0) & (0.0, 0.0) \\
( . ) & ( . ) & (1.0, 0.0) \\
\end{bmatrix}
$$

Call Statement and Input:

```
CALL ZHEGVD ( 3, 'N', 'U', 3, A, 3, B, 3, W, WORK, 0, RWORK, 0, IWORK, 0, INFO)
```
Output:

Matrix $A$ is overwritten.

\[
W = \begin{bmatrix}
0.4537610 \\
2.8973095 \\
10.648929
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
(1.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (1.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (0.0, 0.0) & (1.0, 0.0)
\end{bmatrix}
\]

INFO = 0

Example 4

This example shows how to find the eigenvalues and eigenvectors of a complex Hermitian generalized eigenproblem of the form: $ABx = \lambda x$.

Matrix $A$ is:

\[
\begin{bmatrix}
(6.0, 0.0) & ( . ) & ( . ) \\
(4.0, 0.0) & (6.0, 0.0) & ( . ) \\
(4.0, 0.0) & (1.0, 0.0) & (6.0, 0.0)
\end{bmatrix}
\]

Matrix $B$ is:

\[
\begin{bmatrix}
(1.0, 0.0) & ( . ) & ( . ) \\
(0.0, 0.0) & (1.0, 0.0) & ( . ) \\
(0.0, 0.0) & (0.0, 0.0) & (1.0, 0.0)
\end{bmatrix}
\]

Call Statement and Input:

\[
\begin{align*}
\text{ITYPE} & = 2, \quad \text{JOBZ} = 'V', \quad \text{UPLO} = 'L', \quad \text{N} = 3, \\
A & \text{LDA} = 3, \quad B & \text{LDB} = 3, \quad W & \text{LWORK} = 0, \quad \text{WORK} = 0, \\
& \text{RWORK} = 0, \quad \text{IWORK} = 0, \quad \text{INFO} = 0.
\end{align*}
\]

Output:

\[
W = \begin{bmatrix}
0.8210916 \\
5.0000000 \\
12.178908
\end{bmatrix}
\]

\[
A = \begin{bmatrix}
(-0.737578, 0.0) & (0.47748165,0.0) & (0.4774816,0.0) \\
(0.0, 0.0) & (-0.7071067,0.0) & (0.7071067,0.0) \\
(-0.675261, 0.0) & (-0.5215469,0.0) & (-0.521546,0.0)
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
(1.0, 0.0) & ( . ) & ( . ) \\
(0.0, 0.0) & (1.0, 0.0) & (0.0, 0.0)
\end{bmatrix}
\]
SSYGV, DSYGV, CHEGV, ZHEGV, 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 $\mathbf{A}\mathbf{x} = \lambda \mathbf{B}\mathbf{x}$
- If $\text{itype} = 2$, the problem is $\mathbf{A}\mathbf{B}\mathbf{x} = \lambda \mathbf{x}$
- If $\text{itype} = 3$, the problem is $\mathbf{B}\mathbf{A}\mathbf{x} = \lambda \mathbf{x}$

In the formulas:

- $\mathbf{A}$ represents the real symmetric or complex Hermitian matrix $\mathbf{A}$.
- $\mathbf{B}$ represents the positive definite real symmetric or complex Hermitian matrix $\mathbf{B}$.

Eigenvalues and eigenvectors can be selected by specifying a range of values or a range of indices for the required eigenvalues.

<table>
<thead>
<tr>
<th>$\mathbf{A}$, $\mathbf{B}$, $\mathbf{Z}$, work</th>
<th>$\mathbf{vl}$, $\mathbf{vu}$, $\mathbf{abstol}$, $\mathbf{w}$, $\mathbf{rwork}$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision real</td>
<td>$\text{SSYGV}^\Delta &lt; $ SSYGVX$^\Delta$ SSYGVX</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
<td>$\text{DSYGV}^\Delta &lt; $ DSPGVX$^\Delta$ DSPGVX</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
<td>$\text{CHEGV}^\Delta &lt; $ CHPGVX$^\Delta$ CHPGVX</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
<td>$\text{ZHEGV}^\Delta &lt; $ ZHPGVX$^\Delta$ ZHPGVX</td>
</tr>
</tbody>
</table>

$\Delta$: LAPACK
### Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fortran</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CALL SSYGV</td>
</tr>
<tr>
<td></td>
<td>CALL CHEGV</td>
</tr>
<tr>
<td></td>
<td>CALL SSPGVX</td>
</tr>
<tr>
<td></td>
<td>CALL CHPGVX</td>
</tr>
<tr>
<td></td>
<td>CALL SSYGVX</td>
</tr>
<tr>
<td><strong>C and C++</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ssygv</td>
</tr>
<tr>
<td></td>
<td>chegv</td>
</tr>
<tr>
<td></td>
<td>sspgvx</td>
</tr>
<tr>
<td></td>
<td>chpgvx</td>
</tr>
<tr>
<td></td>
<td>ssygxx</td>
</tr>
<tr>
<td></td>
<td>chegxx</td>
</tr>
<tr>
<td><strong>LAPACKE</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>info = LAPACKE_ssygxx</td>
</tr>
<tr>
<td></td>
<td>info = LAPACKE_chegx</td>
</tr>
<tr>
<td></td>
<td>info = LAPACKE_sspgvx</td>
</tr>
<tr>
<td></td>
<td>info = LAPACKE_chpgvx</td>
</tr>
<tr>
<td></td>
<td>info = LAPACKE_ssygxx</td>
</tr>
<tr>
<td></td>
<td>info = LAPACKE_chegxx</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. The value of the matrix_layout argument 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; range = 'A', 'V', or 'I'.

uplo
indicates whether the upper or lower triangular parts of the matrices $A$ and $B$ are 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 matrices $A$ and $B$ that is 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 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 212 on page 931.

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 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 212 on page 931.
\(a\) 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 212 on page 931.

\(lda\) is the leading dimension of the array specified for \(A\).
Specified as: an integer; \(lda > 0\) and \(lda \geq n\).

\(b\) is the positive definite real symmetric or complex Hermitian matrix \(B\) 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 \(ldb\) by (at least) \(n\) array, containing numbers of the data type indicated in Table 212 on page 931.

\(ldb\) is the leading dimension of the array specified for \(B\).
Specified as: an integer; \(ldb > 0\) and \(ldb \geq n\).

\(vl\) has the following meaning:
If \(range = 'V'\), it is the lower bound of the interval to be searched for eigenvalues.
If \(range \neq 'V'\), this argument is ignored.

Specified as: a number of the data type indicated in Table 212 on page 931. 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 \(range \neq 'V'\), this argument is ignored.

Specified as: a number of the data type indicated in Table 212 on page 931. If \(range = 'V'\), \(vl < vu\).

\(il\) has the following meaning:
If \(range = 'I'\), it is the index (from smallest to largest) of the smallest eigenvalue to be returned.
If \(range \neq 'I'\), this argument is ignored.
Specified as: an integer; \(il \geq 1\).

\(iu\) has the following meaning:
If \(range = 'I'\), it is the index (from smallest to largest) of the largest eigenvalue to be returned.
If \(range \neq 'I'\), this argument is ignored.
Specified as: an integer; \(\min(il, n) \leq iu \leq n\).

\(abstol\) 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:

- \(abstol + \varepsilon(\max(|a|, |b|))\)
where $\varepsilon$ is the machine precision. If $\text{abstol} \leq 0$, then $\varepsilon(\text{norm}(T))$ 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 as follows:

   **For SSPGVX, CHPGVX, SSYGVX, and CHEGVX:**
   
   \begin{align*}
   \varepsilon &= 0.119209289550781250E-06 \\
   \text{unfl} &= 0.1175494351E-37 \\
   \sqrt{\text{unfl}} &= 0.1084202172E-18 \\
   \end{align*}

   **For DSPGVX, ZHPGVX, DSYGVX, and ZHEGVX:**
   
   \begin{align*}
   \varepsilon &= 0.222044604925031308E-15 \\
   \text{unfl} &= 0.222507385850720138E-307 \\
   \sqrt{\text{unfl}} &= 0.149166814624004135E-153 \\
   \end{align*}

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 212 on page 931.

$m$

See “On Return” on page 937.

$w$

See “On Return” on page 937.

$z$

See “On Return” on page 937.

$ldz$

is the leading dimension of the array specified for $Z$. 
Specified as: an integer; \( ldz > 0 \) and \( ldz \geq n \).

\textbf{work} \\
\hspace{1cm} is a work area used by these subroutines, where:

\textbf{For SSPGVX and DSPGVX:} \\
\hspace{1cm} Its size is 8n.

\textbf{For CHPGVX and ZHPGVX:} \\
\hspace{1cm} Its size is 2n.

\textbf{For SSYGV, DSYGV, CHEGV, ZHEGV, SSYGVX, DSYGVX, CHEGVX, and ZHEGVX:} \\
\hspace{1cm} If \( lwork = 0 \), \textit{work} is ignored.
\hspace{1cm} If \( lwork \neq 0 \), the size of \textit{work} is determined as follows:
\hspace{2cm} - If \( lwork \neq -1 \), \textit{work} is (at least) of length \( lwork \).
\hspace{2cm} - If \( lwork = -1 \), \textit{work} is (at least) of length 1.

Specified as: an area of storage containing numbers of the data type indicated in Table 212 on page 931.

\textbf{lwork} \\
\hspace{1cm} is the number of elements in array \textit{WORK}.

Specified as: an integer; where:
\hspace{1cm} - If \( lwork = 0 \), the subroutine dynamically allocates the workspace that is needed for use during this computation. The work area is deallocated before control is returned to the calling program.
\hspace{1cm} - If \( lwork = -1 \), subroutine performs a workspace query and returns the optimal required size of \textit{work} in \textit{work}_1. No computation is performed and the subroutine returns after error checking is complete.
\hspace{1cm} - Otherwise:
\hspace{2cm} \textbf{For SSYGV and DSYGV:} \\
\hspace{3cm} \( lwork \geq \max (1, 3n - 1) \).
\hspace{2cm} \textbf{For CHEGV and ZHEGV:} \\
\hspace{3cm} \( lwork \geq \max (1, 2n - 1) \).
\hspace{2cm} \textbf{For SSYGVX and DSYGVX:} \\
\hspace{3cm} \( lwork \geq \max (1, 8n) \).
\hspace{2cm} \textbf{For CHEGVX and ZHEGVX:} \\
\hspace{3cm} \( lwork \geq \max (1, 2n) \).

\textbf{Note:} These formulas represent the minimum required workspace. For best performance, specify either \( lwork = -1 \) (to obtain the optimal size of workspace) or \( lwork = 0 \) (to direct the subroutine to dynamically allocate the workspace).

\textbf{rwork} \\
\hspace{1cm} is a work area of size 7n.

Specified as: an area of storage containing numbers of the data type indicated in Table 212 on page 931.

\textbf{iwork} \\
\hspace{1cm} is a work area of size 5n.

Specified as: an area of storage containing integers.

\textbf{ifail} \\
\hspace{1cm} See “On Return” on page 937.

\textbf{info} \\
\hspace{1cm} See “On Return” on page 937.
On Return

\( ap \)

is overwritten.

Returned as: a one-dimensional array of (at least) length \( n(n + 1)/2 \), containing numbers of the data type indicated in Table 212 on page 931.

\( bp \)

contains the results of the Cholesky factorization.

**For SSPGVX and DSPGVX:**

If \( uplo = 'U' \), if \( info \leq n \), the triangular factor \( U \) from the Cholesky factorization \( B = U^T U \) stored in upper-packed storage format.

If \( uplo = 'L' \), if \( info \leq n \), the triangular factor \( L \) from the Cholesky factorization \( B = LL^T \) stored in lower-packed storage mode.

**For CHPGVX and ZHPGVX:**

If \( uplo = 'U' \), if \( info \leq n \), the triangular factor \( U \) from the Cholesky factorization \( B = U^H U \) stored in upper-packed storage format.

If \( uplo = 'L' \), if \( info \leq n \), the triangular factor \( L \) from the Cholesky factorization \( B = LL^H \) 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 Table 212 on page 931.

\( a \)

is overwritten as follows:

**For SSYGV, DSYGV, CHEGV, and ZHEGV:**

- If \( jobz = 'V' \) and if \( info = 0 \), the matrix \( A \) contains the matrix \( Z \) of eigenvectors. The eigenvectors are normalized as follows:
  - If \( itype = 1 \) or \( 2 \), \( Z^H B Z = I \).
  - If \( itype = 3 \), \( Z^H B^{-1} Z = I \).
- If \( jobz = 'V' \) and if \( info \neq 0 \), the eigenvectors are not valid.
- If \( jobz = 'N' \) and,
  - 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 SSPGVX, DSPGVX, CHPGVX, ZHPGVX, SSYGVX, DSYGVX, CHEGVX, and ZHEGVX:**

- 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.

Returned as: an array of dimension \( lda \) by (at least) \( n \), containing numbers of the data type indicated in Table 212 on page 931.

\( b \)

contains the results of the Cholesky factorization.

> | For SSYGV, DSYGV, SSYGVX, and DSYGVX: |

- 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 \( B = U^T U \).
- 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 \( B = LL^T \).
For CHEGV, ZHEGV, CHEGVX, and ZHEGVX:

If $\text{uplo} = 'U'$, if $\text{info} \leq n$, the leading $n$ by $n$ upper triangular part of $B$ contains the triangular factor $U$ from the Cholesky factorization $B = U^H U$.

If $\text{uplo} = 'L'$, if $\text{info} \leq n$, the leading $n$ by $n$ lower triangular part of $B$ contains the triangular factor $L$ from the Cholesky factorization $B = LL^H$.

Returned as: an array of dimension $\text{lbd}$ by (at least) $n$, containing numbers of the data type indicated in Table 212 on page 931.

$m$

is the number of eigenvalues found.

Returned as: an integer; $0 \leq m \leq n$.

$w$

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 212 on page 931.

$z$

has the following meaning, where:

If $\text{jobz} = 'N'$, then $z$ is ignored.

If $\text{jobz} = 'V'$ and $\text{info} = 0$, 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 $\text{itype} = 1$ or $2$, $Z^T B Z = I$.

If $\text{itype} = 3$, $Z^T B^{-1} Z = I$.

For CHPGVX, ZHPGVX, CHEGVX, and ZHEGVX:

If $\text{itype} = 1$ or $2$, $Z^H B Z = I$.

If $\text{itype} = 3$, $Z^H B^{-1} Z = I$.

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 $\text{ifail}$.

**Note:** You must ensure that at least $\max(1, m)$ columns are supplied in the array $Z$; if $\text{range} = 'V'$, the exact value of $m$ is not known in advance and an upper bound must be used.

Returned as: an $\text{ldz}$ by (at least) $\max(1, m)$ array, containing numbers of the data type indicated in Table 212 on page 931.

$\text{work}$

is a work area used by this subroutine if $lwork \neq 0$, where:

If $lwork = 0$ and $lwork \neq -1$, its size is (at least) of length $\text{lwork}$.

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 $\text{work}_1$ is set to the optimal $lwork$ value and all other elements of $\text{work}$ are overwritten.

$rwork$

is a work area used by these subroutines for computation.

Returned as: an area of storage of dimension, max $(1, 3n-2)$. 
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 > 0,
- If info = n + i, for 1 ≤ i ≤ n, the leading minor of order i of matrix B is not positive definite. The factorization of matrix B cannot be completed and eigenvalues or eigenvectors are not computed.
- For SSPGVX, DSPGVX, CHPGVX, ZHPGVX, SSYGVX, DSYGVX, CHEGVX, ZHEGVX, LAPACKE_sspgvx, LAPACKE_dspgvx, LAPACKE_chpgvx, LAPACKE_zhpgvx, LAPACKE_ssygvx, LAPACKE_dsygvx, LAPACKE_chevgx, and LAPACKE_zhegvx:
  If info = i, i eigenvectors failed to converge. The indices of the eigenvectors are saved in array ifail.
- For SSYGV, DSYGV, CHEGV, ZHEGV, LAPACKE_ssygv, LAPACKE_dsygv, LAPACKE_chevg, LAPACKE_zhevg:
  If info = i, the algorithm failed to converge. The i off-diagonal elements of an intermediate tridiagonal form did not converge to zero.

Returned as:
- For SSYGV, DSYGV, CHEGV, ZHEGV, SSPGVX, DSPGVX, CHPGVX, ZHPGVX, SSYGVX, DSYGVX, CHEGVX, and ZHEGVX: an integer; info ≥ 0.
- For LAPACKE_ssygv, LAPACKE_dsygv, LAPACKE_chevg, LAPACKE_zhevg, LAPACKE_sspgvx, LAPACKE_dspgvx, LAPACKE_chpgvx, LAPACKE_zhpgvx, LAPACKE_ssygvx, LAPACKE_dsygvx, LAPACKE_chevgx, and LAPACKE_zhegvx: 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 80 or “Upper-Packed Storage Mode” on page 82, respectively.
   For a description of how complex Hermitian matrices are stored in lower- or upper-packed storage mode, see “Complex Hermitian Matrix” on page 84.
5. For a description of how real symmetric matrices are stored in lower or upper storage mode, see “Lower Storage Mode” on page 83 or “Upper Storage Mode” on page 83, respectively.
   For a description of how complex Hermitian matrices are stored in lower or upper storage mode, see “Complex Hermitian Matrix” on page 84.
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 either $lwork = -1$ (to obtain the optimal required size of workspace) or $lwork = 0$ (to direct the subroutine to dynamically allocate the workspace).

Function

These subroutines compute eigenvalues and, optionally, the eigenvectors of a positive definite real symmetric or complex Hermitian generalized eigenproblem:

- If $itype = 1$, the problem is $Ax = \lambda Bx$
- If $itype = 2$, the problem is $ABx = \lambda x$
- If $itype = 3$, the problem is $BAX = \lambda x$

In the formulas:

- $A$ 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 required 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 [“9” on page 1253].

Error conditions

Resource Errors

1. $lwork = 0$ and unable to allocate work area.

Computational Errors

1. The matrix $B$ is not positive definite. See output argument $info$ 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 that are 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 $ifail$.
6. If $info = i$, the algorithm failed to converge. The value of $i$ indicates the number of elements of an intermediate tridiagonal form that did not converge to zero.

Informational Errors

1. ESSL computed the eigenvalues using multiple algorithms. Performance may be degraded.
Input-Argument Errors

1. \( itype < 1 \) or \( itype > 3 \)
2. \( jobz \neq 'N' \) or \( 'V' \)
3. \( range \neq 'A', 'V', \) or \( 'I' \)
4. \( uplo \neq 'U' \) or \( 'L' \)
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. \( range = 'I', \) and \( (il < 1 \) or \( il > \max(1, n)) \)
12. \( range = 'I', \) and \( (iu < \min(n, il) \) 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: \( Ax = \lambda Bx \). Matrices \( A \) and \( B \) are stored in upper storage mode.

Matrix \( A \) is:

\[
\begin{pmatrix}
10 & 1 & 2 \\
. & 2 & -1 \\
. & . & 2 \\
\end{pmatrix}
\]

Matrix \( B \) is:

\[
\begin{pmatrix}
1 & 0 & 0 \\
. & 1 & 0 \\
. & . & 1 \\
\end{pmatrix}
\]

Call Statement and Input:

\[
\text{CALL DSYGV} \ (1, 'N', 'U', 3, A, 3, B, 3, W, WORK, 0, INFO)
\]

Output:

Matrix \( A \) is overwritten.

\[
W = \begin{pmatrix}
0.5153 \\
2.9466 \\
10.5381 \\
\end{pmatrix}
\]

\[
B = \begin{pmatrix}
1 & 0 & 0 \\
. & 1 & 0 \\
\end{pmatrix}
\]
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 storage mode.

Matrix $A$ is:

```
| 6  4  4 |
| .  6  1 |
| .  .  6 |
```

Matrix $B$ is:

```
| 1  0  0 |
| .  1  0 |
| .  .  1 |
```

Call Statement and Input:

```
CALL DSYGV ( 2, 'V', 'U', 3, A, 3, B, 3, W, WORK, 0, INFO)
```

Output:

```
W =
| 0.8210916 |
| 5.000000  |
| 12.178908 |
```

```
A =
| -0.737578  0.4774816  0.4774816 |
| 0.000000  -0.707107  0.7071067 |
| -0.675261  -0.521546 -0.521546  |
```

```
B =
| 1  0  0 |
| .  1  0 |
| .  .  1 |
```

INFO = 0

Example 3

This example shows how to find the eigenvalues of a real symmetric positive generalized eigenproblem of the form: $BAx = \lambda x$. Matrices $A$ and $B$ are stored in upper storage mode.

Matrix $A$ is:

```
| (10.0, 1.0)  (1.0, 0.0)  (2.0, -1.0) |
| (-1.0, 0.0)  (2.0, -1.0)  (-1.0, 0.0) |
```

INFO = 0
Matrix $B$ is:

$$
\begin{pmatrix}
(1.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (1.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (0.0, 0.0) & (1.0, 0.0)
\end{pmatrix}
$$

Call Statement and Input:

```
ITYPE JOBZ UPLO N  A LDA  B LDB  W  WORK LWORK RWORK INFO
CALL ZHEGV ( 3,   'N', 'U', 3, A, 3,  B, 3,  W, WORK, 0,  RWORK, INFO)
```

Output:

Matrix $A$ is overwritten.

$$
W = \begin{pmatrix}
0.4537610 \\
2.8973095 \\
10.648929
\end{pmatrix}
$$

$$
B = \begin{pmatrix}
(1.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (1.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (0.0, 0.0) & (1.0, 0.0)
\end{pmatrix}
$$

INFO = 0

**Example 4**

This example shows how to find the eigenvalues and eigenvectors of a complex Hermitian generalized eigenproblem of the form: $ABx = \lambda x$. Matrices $A$ and $B$ are stored in lower storage mode.

Matrix $A$ is:

$$
\begin{pmatrix}
(6.0, 0.0) & ( . ) & ( . ) \\
(4.0, 0.0) & (6.0, 0.0) & ( . ) \\
(4.0, 0.0) & (1.0, 0.0) & (6.0, 0.0)
\end{pmatrix}
$$

Matrix $B$ is:

$$
\begin{pmatrix}
(1.0, 0.0) & ( . ) & ( . ) \\
(0.0, 0.0) & (1.0, 0.0) & ( . ) \\
(0.0, 0.0) & (0.0, 0.0) & (1.0, 0.0)
\end{pmatrix}
$$

Call Statement and Input:

```
ITYPE JOBZ UPLO N  A LDA  B LDB  W  WORK LWORK RWORK INFO
CALL ZHEGV ( 2,   'V', 'L', 3, A, 3, B, 3, W, WORK, 0,  RWORK, INFO)
```
Output:

\[
W = \begin{bmatrix}
0.8210916 \\
5.0000000 \\
12.178908
\end{bmatrix}
\]

\[
A = \begin{bmatrix}
(-0.737578, 0.0) & (0.47748165, 0.0) & (0.4774816, 0.0) \\
(0.0, 0.0) & (-0.7071067, 0.0) & (0.7071067, 0.0) \\
(-0.675261, 0.0) & (-0.5215469, 0.0) & (-0.521546, 0.0)
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
(1.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (1.0, 0.0) & (0.0, 0.0) & (0.0, 0.0) \\
(0.0, 0.0) & (0.0, 0.0) & (1.0, 0.0) & (0.0, 0.0)
\end{bmatrix}
\]

INFO = 0

Example 5

This example shows how to find the eigenvalues of a real symmetric positive generalized eigenproblem of the form: \(Ax = \lambda Bx\). 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:

\[
\begin{align*}
\text{ITYPE} & = 1 \\
\text{JOBZ} & = \text{'N'} \\
\text{RANGE} & = \text{'A'} \\
\text{UPLO} & = \text{'L'} \\
\text{N} & = 4 \\
\text{AP} & = (6.0, 4.0, 4.0, 1.0, 6.0, 1.0, 4.0, 6.0, 1.0, 4.0, 6.0) \\
\text{BP} & = (1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0)
\end{align*}
\]

Output:
Example 6
This example shows how to find the eigenvalues and eigenvectors of a real symmetric positive generalised 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 $range = 'I'$.

Matrices $A$ and $B$ are the same as in Example 5.

**Note:** Because $range = 'I'$, arguments $vl$ and $vu$ are not referenced.

Call Statement and Input:

```fortran
CALL DSPGVX ( 2, 'V', 'I', 'U', 4, AP, BP, VL, VU, 1, 2, -1.0, M, W, Z, 4, WORK, IWORK, IFAIL, INFO)
```

Output:
Matrix $AP$ is overwritten.

```
BP = (1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0)
```

```
M = 4
```

```
W = (-1.0, 5.0, 5.0, 15.0)
```

```
INFO = 0
```

Example 7
This example shows how to find all eigenvalues only of a positive definite complex Hermitian generalised eigenproblem of the form: $Ax = \lambda Bx$. Matrices $A$ and $B$ are stored in lower-packed storage mode.

Matrix $A$ is:

```
| (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) |
Matrix $B$ is:

\[
\begin{pmatrix}
1.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 1.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 1.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 1.0 \\
\end{pmatrix}
\]

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:

```
CALL ZHPGVX ( 1, 'N', 'A', 'L', 4, AP, BP VL, VU, IL, IU, -1.0, M, W, Z, 4, WORK, RWORK, IWORK, IFAIL, INFO)
```

```
AP       =  ((6.0, . ), (4.0, 0.0), (4.0, 0.0), (1.0, 0.0),(6.0, . ), (1.0, 0.0), (4.0, 0.0), (6.0, . ),
(4.0, 0.0), (6.0, . ))
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))
```

Output:

Matrix $AP$ is overwritten.

```
M    =    4
```

```
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))
```

```
W    =  (-1.0, 5.0, 5.0, 15.0)
```

```
INFO = 0
```

Example 8

This example shows how to find all 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 $range = 'I'$.

Matrices $A$ and $B$ are the same as in Example 7.

Notes:
1. Because $range = 'I'$, arguments $vl$ and $vu$ are not referenced.
2. 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 ZHPGVX ( 2, 'V', 'I', 'L', 4, AP, BP, VL, VU, 1, 2, -1.0, M, W, Z, WORK, RWORK, IWORK, IFAIL, INFO)
```

| AP       | = ( (6.0, . ), (4.0, 0.0), (4.0, 0.0), (1.0, 0.0), (6.0, . ), (1.0, 0.0), (4.0, 0.0), (6.0, . ),
|          | (4.0, 0.0), (6.0, . ) )
| BP       | = ( (1.0, . ), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (1.0, . ), (0.0, 0.0), (0.0, 0.0), (1.0, . ),
|          | (0.0, 0.0), (1.0, . ) )
```

Output:

Matrix AP is overwritten.

```
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), (1.0, 0.0), (1.0, 0.0),
```

M = 2

W = (-1.0, 5.0, . , . )

```
Z = ( ( 0.500000, 0.0) ( 0.544042, 0.0) . . |
    | (0.500000, 0.0) ( 0.451683, 0.0) . . |
    | (0.500000, 0.0) ( -0.451683, 0.0) . . |
    | ( 0.500000, 0.0) ( -0.544042, 0.0) . . |
```

IFAIL = (0, 0, . , . )
INFO = 0

Example 9

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 5

**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.

Call Statement and Input:

```
CALL DSYGV ( 1, 'N', 'A', 'L', 4, A, 4, B, 4, VL, VU, IL, IU, -1.0, M, W, Z, WORK, LWORK, IWORK, IFAIL, INFO)
```

```
A = | 6.0 . . |
    | 4.0 6.0 . |
    | 4.0 1.0 6.0 |
    | 1.0 4.0 4.0 6.0 |
B = | 1.0 . . |
    | 0.0 1.0 . |
    | 0.0 0.0 1.0 |
    | 0.0 0.0 0.0 1.0 |
```

Output:
Matrix $A$ is overwritten.

<table>
<thead>
<tr>
<th>$M$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.000000  .  .  .  .</td>
</tr>
<tr>
<td>0.000000  1.000000  .  .</td>
</tr>
<tr>
<td>0.000000  0.000000  1.000000  .</td>
</tr>
<tr>
<td>0.000000  0.000000  0.000000  1.000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$W$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.000000</td>
</tr>
<tr>
<td>5.000000</td>
</tr>
<tr>
<td>5.000000</td>
</tr>
<tr>
<td>15.000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$INFO$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

**Example 10**

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 range = 'I'.

Matrices $A$ and $B$ are the same as in Example 5.

**Notes:**
1. Because range = 'I', arguments $vl$ and $vu$ are not referenced.
2. Because $lwork = 0$, the subroutine dynamically allocates WORK.

**Call Statement and Input:**

```plaintext
CALL DSYGVX ( 2, 'V', 'I', 'U', 4, A, 4, B, 4, VL, VU, 1, 2, -1.0, M, W, Z, 4, WORK, 0, IWORK, IFAIL, INFO)
```

**Output:**

Matrix $A$ is overwritten.

<table>
<thead>
<tr>
<th>$M$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.000000  0.000000  0.000000  0.000000</td>
</tr>
<tr>
<td>.  1.000000  0.000000  0.000000</td>
</tr>
<tr>
<td>.  .  1.000000  0.000000</td>
</tr>
<tr>
<td>.  .  .  1.000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$W$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.000000</td>
</tr>
<tr>
<td>5.000000</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$Z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>.  0.500000  0.543058</td>
</tr>
<tr>
<td>-0.500000  0.452866</td>
</tr>
<tr>
<td>-0.500000 -0.452866</td>
</tr>
</tbody>
</table>

948 IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and Reference
Example 11

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 5.

Notes:
1. Because range = 'V', arguments $i\ell$ and $iu$ are not referenced.
2. On output, array $A$ is overwritten.
3. Because $lwork = 0$, the subroutine dynamically allocates WORK.

Call Statement and Input:

```
ITYPE JOBZ RANGE UPLO  N  A LDA  B LDB  VL  VU  IL  IU  ABSTOL  M  W  Z  LDZ  WORK LWORK  IWORK  IFAIL  INFO
CALL DSYGVX ( 3, 'V', 'V', 'U', 4, A, 4, B, 4, 2.0, 6.0, IL, IU, -1.0, M, W, Z, 4, WORK, 0, IWORK, IFAIL, INFO)
```

```
A =  6.0  4.0  4.0  1.0
    .  6.0  1.0  4.0
    .  .  6.0  4.0

B =  1.0  0.0  0.0  0.0
    .  1.0  0.0  0.0
    .  .  1.0  0.0
    .  .  .  1.0
```

Output:
Matrix $A$ is overwritten.

```
M =  2

B =  1.000000  0.000000  0.000000  0.000000
    .  1.000000  0.000000  0.000000
    .  .  1.000000  0.000000
    .  .  .  1.000000
```

```
W =  5.000000
    5.000000
    .
    .
```

```
Z =  -0.123202 -0.696291  .  .
    0.696291 -0.123202  .  .
    -0.696291  0.123202  .  .
```

Eigensystem analysis subroutines 949
Example 12

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:

```Fortran
CALL ZHEGVX ( ITYPE, JOBZ, 'A', 'L', N, A, LDA, B, LDB, VL, VU, IL, IU, -1.0, M, W, Z, LDZ, WORK, LWORK, RWORK, IWORK, IFAIL, INFO )
```

| IFAIL = (0, 0, 0, 0) |
| INFO = 0 |

Example 13

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.

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>(6.0, 0.0)</td>
</tr>
<tr>
<td>(4.0, 0.0) (6.0, 0.0)</td>
</tr>
<tr>
<td>(1.0, 0.0) (4.0, 0.0) (6.0, 0.0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1.0, 0.0)</td>
</tr>
<tr>
<td>(0.0, 0.0) (1.0, 0.0)</td>
</tr>
<tr>
<td>(0.0, 0.0) (0.0, 0.0) (1.0, 0.0)</td>
</tr>
</tbody>
</table>

Output:

Matrix \( A \) is overwritten.

M = 4

<table>
<thead>
<tr>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1.0, 0.0)</td>
</tr>
<tr>
<td>(0.0, 0.0) (1.0, 0.0)</td>
</tr>
<tr>
<td>(0.0, 0.0) (0.0, 0.0) (1.0, 0.0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.0</td>
</tr>
<tr>
<td>5.0</td>
</tr>
<tr>
<td>5.0</td>
</tr>
<tr>
<td>15.0</td>
</tr>
</tbody>
</table>

INFO = 0
This example illustrates the use of the \texttt{il} and \texttt{iu} parameters when \texttt{range} = 'I'.

Matrices \texttt{A} and \texttt{B} are the same as in Example 3.

**Notes:**

1. Because \texttt{range} = 'I', arguments \texttt{vl} and \texttt{vu} are not referenced.
2. Because \texttt{lwork} = 0, the subroutine dynamically allocates \texttt{WORK}.
3. On input, the imaginary parts of the Hermitian matrix \texttt{A} are assumed to be zero, values. On output, they are set to zero.

**Call Statement and Input:**

```
CALL ZHEGVX ( 2, 'V', 'I', 'L', 4, A, 4, B, 4, VL, VU, 1, 2, -1.0, M, W, Z, 4, WORK, 0, RWORK, IWORK, IFAIL, INFO)
```

**Output:**

Matrix \texttt{A} is overwritten.

\begin{verbatim}
M = 2
\end{verbatim}

\begin{verbatim}
B = (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{verbatim}

\begin{verbatim}
W = -1.000000 5.000000
\end{verbatim}

\begin{verbatim}
Z = (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{verbatim}

\begin{verbatim}
IFAIL = (0, 0, ., .)
INFO = 0
\end{verbatim}
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>
<thead>
<tr>
<th>Short-Precision Subroutine</th>
<th>Long-Precision Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCFTD&lt;sup&gt;GPU&lt;/sup&gt;</td>
<td>DCFTD&lt;sup&gt;GPU&lt;/sup&gt;</td>
<td>“SCFTD and DCFTD (Multidimensional Complex Fourier Transform)” on page 962</td>
</tr>
<tr>
<td>SRCFTD&lt;sup&gt;GPU&lt;/sup&gt;</td>
<td>DRCFTD&lt;sup&gt;GPU&lt;/sup&gt;</td>
<td>“SRCFTD and DRCFTD (Multidimensional Real-to-Complex Fourier Transform)” on page 970</td>
</tr>
<tr>
<td>SCRFTD&lt;sup&gt;GPU&lt;/sup&gt;</td>
<td>DCRFTD&lt;sup&gt;GPU&lt;/sup&gt;</td>
<td>“SCRFTD and DCRFTD (Multidimensional Complex-to-Real Fourier Transform)” on page 978</td>
</tr>
<tr>
<td>SCFT&lt;sup&gt;§&lt;/sup&gt;</td>
<td>DCFT&lt;sup&gt;§&lt;/sup&gt;</td>
<td>“SCFT and DCFT (Complex Fourier Transform)” on page 986</td>
</tr>
<tr>
<td>SCFTP&lt;sup&gt;§, ND&lt;/sup&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRCFT&lt;sup&gt;§&lt;/sup&gt;</td>
<td>DRCFT&lt;sup&gt;§&lt;/sup&gt;</td>
<td>“SRCFT and DRCFT (Real-to-Complex Fourier Transform)” on page 993</td>
</tr>
<tr>
<td>SCRFT&lt;sup&gt;§&lt;/sup&gt;</td>
<td>DCRFT&lt;sup&gt;§&lt;/sup&gt;</td>
<td>“SCRFT and DCRFT (Complex-to-Real Fourier Transform)” on page 1000</td>
</tr>
<tr>
<td>SCOSF&lt;sup&gt;§&lt;/sup&gt;</td>
<td>DCOSF</td>
<td>“SCOSF and DCOSF (Cosine Transform)” on page 1008</td>
</tr>
<tr>
<td>SCOSFT&lt;sup&gt;§, ND&lt;/sup&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSINF</td>
<td>DSINF</td>
<td>“SSINF and DSINF (Sine Transform)” on page 1015</td>
</tr>
<tr>
<td>SCFT2&lt;sup&gt;§&lt;/sup&gt;</td>
<td>DCFT2&lt;sup&gt;§&lt;/sup&gt;</td>
<td>“SCFT2 and DCFT2 (Complex Fourier Transform in Two Dimensions)” on page 1021</td>
</tr>
<tr>
<td>SCFT2P&lt;sup&gt;§, ND&lt;/sup&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRCFT2&lt;sup&gt;§&lt;/sup&gt;</td>
<td>DRCFT2&lt;sup&gt;§&lt;/sup&gt;</td>
<td>“SRCFT2 and DRCFT2 (Real-to-Complex Fourier Transform in Two Dimensions)” on page 1028</td>
</tr>
<tr>
<td>SCRFT2&lt;sup&gt;§&lt;/sup&gt;</td>
<td>DCRFT2&lt;sup&gt;§&lt;/sup&gt;</td>
<td>“SCRFT2 and DCRFT2 (Complex-to-Real Fourier Transform in Two Dimensions)” on page 1034</td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 2018
Table 214: 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>SCFT3 $^§$</td>
<td>DCFT3 $^§$</td>
<td>“SCFT3 and DCFT3 (Complex Fourier Transform in Three Dimensions)” on page 1041</td>
</tr>
<tr>
<td>SCFT3P $^§$, ND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRCFT3 $^§$</td>
<td>DRCFT3 $^§$</td>
<td>“SRCFT3 and DRCFT3 (Real-to-Complex Fourier Transform in Three Dimensions)” on page 1047</td>
</tr>
<tr>
<td>SCRFT3 $^§$</td>
<td>DCRFT3 $^§$</td>
<td>“SCRFT3 and DCRFT3 (Complex-to-Real Fourier Transform in Three Dimensions)” on page 1053</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.

GPU This subroutine can offload computations to the GPUs when the ESSL SMP CUDA library is used.

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 215: 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></td>
<td>“SCON and SCOR (Convolution or Correlation of One Sequence with One or More Sequences)” on page 1060</td>
</tr>
<tr>
<td>SCOR $^§$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCOND</td>
<td></td>
<td>“SCOND and SCORD (Convolution or Correlation of One Sequence with Another Sequence Using a Direct Method)” on page 1065</td>
</tr>
<tr>
<td>SCORD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCONF</td>
<td></td>
<td>“SCONF and SCORF (Convolution or Correlation of One Sequence with One or More Sequences Using the Mixed-Radix Fourier Method)” on page 1070</td>
</tr>
<tr>
<td>SCORF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SDCON</td>
<td>DDCON</td>
<td>“SDCON, DDCON, SDCOR, and DDCOR (Convolution or Correlation with Decimated Output Using a Direct Method)” on page 1079</td>
</tr>
<tr>
<td>SDCOR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SACOR $^§$</td>
<td></td>
<td>“SACOR (Autocorrelation of One or More Sequences)” on page 1083</td>
</tr>
<tr>
<td>SACORF</td>
<td></td>
<td>“SACORF (Autocorrelation of One or More Sequences Using the Mixed-Radix Fourier Method)” on page 1086</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 216: 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 1091</td>
</tr>
<tr>
<td>SIZC</td>
<td>DIZC</td>
<td>“SIZC and DIZC (I-th Zero Crossing)” on page 1094</td>
</tr>
<tr>
<td>STREC</td>
<td>DTREC</td>
<td>“STREC and DTREC (Time-Varying Recursive Filter)” on page 1096</td>
</tr>
<tr>
<td>SQINT</td>
<td>DQINT</td>
<td>“SQINT and DQINT (Quadratic Interpolation)” on page 1099</td>
</tr>
<tr>
<td>SWLEV, CWLEV</td>
<td>DWLEV, ZWLEV</td>
<td>“SWLEV, DWLEV, CWLEV, and ZWLEV (Wiener-Levinson Filter Coefficients)” on page 1102</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 \( j = 0, 1, ..., n-1 \) rather than \( j = 1, 2, ..., n \). 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 217 on page 956, all transform lengths between 0 and 1073479680 are acceptable.
- For subroutines in Table 218 on page 956, you have two choices:
  - You can use the formula or table of values in “Acceptable Lengths for the Transforms” on page 956 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 56.
Table 217: 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 218: Fourier Transform subroutines whose lengths are limited to those in Figure 15 on page 957.

<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:

\[
n = (2^h) \cdot (3^i) \cdot (5^j) \cdot (7^k) \cdot (11^m) \quad \text{for } n \leq 37748736
\]

where:

- \( h = 1, 2, \ldots, 25 \)
- \( i = 0, 1, 2 \)
- \( j, k, m = 0, 1 \)

Figure 15 on page 957 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 \textit{aux} and \textit{aux2}, respectively. The sizes of these storage areas are specified by the calling sequence arguments \textit{aux1}, \textit{auxx1}, and \textit{aux2}, respectively.

\textbf{AUX1}

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.

**Figure 15: Table of Acceptable Lengths for the Transforms**

<table>
<thead>
<tr>
<th>\textbf{12}</th>
<th>\textbf{13}</th>
<th>\textbf{14}</th>
<th>\textbf{15}</th>
<th>\textbf{16}</th>
<th>\textbf{17}</th>
<th>\textbf{18}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{2}</td>
<td>\textbf{3}</td>
<td>\textbf{4}</td>
<td>\textbf{5}</td>
<td>\textbf{6}</td>
<td>\textbf{7}</td>
<td>\textbf{8}</td>
</tr>
<tr>
<td>\textbf{9}</td>
<td>\textbf{10}</td>
<td>\textbf{11}</td>
<td>\textbf{12}</td>
<td>\textbf{13}</td>
<td>\textbf{14}</td>
<td>\textbf{15}</td>
</tr>
<tr>
<td>\textbf{16}</td>
<td>\textbf{17}</td>
<td>\textbf{18}</td>
<td>\textbf{19}</td>
<td>\textbf{20}</td>
<td>\textbf{21}</td>
<td>\textbf{22}</td>
</tr>
<tr>
<td>\textbf{23}</td>
<td>\textbf{24}</td>
<td>\textbf{25}</td>
<td>\textbf{26}</td>
<td>\textbf{27}</td>
<td>\textbf{28}</td>
<td>\textbf{29}</td>
</tr>
</tbody>
</table>

**Fourier Transforms, Convolutions and Correlations, and Related Computations**

957
The aux3 argument is provided for migration purposes only and is ignored.

Initializing Auxiliary Working Storage

In many of those subroutines requiring aux1 auxiliary working storage, two invocations of the subroutines are necessary. The first invocation initializes the working storage in aux1 for the subroutine, and the second performs the computations. (For an explanation of auxiliary working storage, see “Understanding Auxiliary Working Storage Requirements” on page 957.) As a result, the working storage in aux1 should not be used by the calling program between the two calls to the subroutine. However, it can be reused after intervening calls to the subroutine with different arguments.

If you plan to repeat a computation many times using the same set of arguments, you only need to do one initialization of the aux1 array; that is, the initialized aux1 array can be saved and reused as many times as needed for the computation.

If you plan to perform different computations, with different sets of arguments (except for input argument x), you need to do an initialization for each different computation; that is, you initialize the various aux1 arrays for use with the different computations, saving and reusing them until they are not needed any more.

Determining the Amount of Auxiliary Working Storage That You Need

To determine the size of auxiliary storage, you have several choices. First, you can use the formulas provided in each subroutine description. Second, ESSL’s input-argument error recovery provides a means of determining the minimum size you need for auxiliary storage. It uses the optionally-recoverable error 2015. For details, see “Using Auxiliary Storage in ESSL” on page 49. Third, you can have ESSL dynamically allocate aux and aux2. For details, see “Dynamic Allocation of Auxiliary Storage” on page 51.

When you use the ESSL SMP CUDA library subroutines, SCFTD, DCFTD, SRCFTD, DRCFTD, SCRFTD, and DCRFTD, use a small auxiliary working storage in host memory if the computation is offloaded to the GPUs. You can save significant memory by using the error handling facility to determine the size of auxiliary storage when you use the ESSL SMP CUDA library.

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 Chapter 2, “Planning for ESSL,” on page 33.

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 61.

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, which might prevent the ESSL SMP CUDA library subroutines SCFTD, DCFTD, SRCFTD, DRCFTD, SCRFTD, and DCRFTD from running on the GPUs. 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, for best performance, you must 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 1195.

To obtain the best performance in the three-dimensional Fourier transform subroutines, you should use strides, $inc2$ and $inc3$, provided by the STRIDE subroutine and declare your three-dimensional data structure as a one-dimensional array. The three-dimensional Fourier transform subroutines assume that $inc1$ for the array is 1. Therefore, each element $x_{ijk}$ for $i = 0, 1, ..., n1-1$, $j = 0, 1, ..., n2-1$, and $k = 0, 1, ..., n3-1$ of the three-dimensional data structure of dimensions $n1$ by $n2$ by $n3$ is stored in a one-dimensional array $X(0:L)$ at location $X(l)$, where $l = i + inc2(j) + inc3(k)$. The minimum required value of $L$ is calculated by inserting the maximum values for $i$, $j$, and $k$ in the above equation, giving $L = (n1-1) + inc2(n2-1) + inc3(n3-1)$. The minimum total size of array $X$ is $L+1$. To ensure that this mapping is unique so no two elements $x_{ijk}$ occupy the same array element, $X(l)$, the subroutines have the following restriction: $inc2 \geq n1$ and $inc3 \geq (inc2)(n2)$. This arrangement of array data in storage leaves some blank space between successive planes of the array $X$. By determining the best size for this space, specifying an optimum $inc3$ stride, the third dimension of the array does not create conflicts in the 3090 storage hierarchy.

If the $inc3$ stride value returned by the STRIDE subroutine turns out to be a multiple of $inc2$, the array $X$ can be declared as a three-dimensional array as $X(inc2, inc3/inc2, n3)$; otherwise, it can be declared as either a one-dimensional array, $X(0:L)$, as described above, or a two-dimensional array $X(0:inc3-1, 0:n3-1)$, where $x_{ijk}$ is stored in $X(l,k)$ where $l = i + (inc2)(j)$. In such case, the transform cannot be offloaded to the GPUs when you use the ESSL SMP CUDA library.

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.

Using GPUs

Performance may improve by offloading computations to the GPUs for some large transform sizes. To achieve optimal performance, experimentation is necessary by comparing performance results from the ESSL SMP library that uses data with strides provided by the STRIDE subroutine, and the ESSL SMP CUDA library that uses contiguous data.
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:
  \[ N = N_1 N_2 N_3 ... N_p \]
  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 1253], [“6” on page 1253], and [“8” on page 1253]).

- 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.

- For systems with NVIDIA GPUs, if the ESSL SMP CUDA library is used, the SCFTD, DCFTD, SRCFTD, DRCFTD, SCRFTD, and DCRFTD subroutines can offload large transforms to GPUs if the sequences length and data layout requirements are satisfied. For details, see the subroutine descriptions.

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 \( 2^{56} - 1 \) for long-precision computations), the results are exact.

However, when short-precision subroutines use the 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 [“117” on page 1259] and [“105” on page 1258] 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 61.
Fourier Transform Subroutines

This contains the Fourier transform subroutine descriptions.

SCFTD and DCFTD (Multidimensional Complex Fourier Transform)

Purpose

These subroutines compute a set of $m \times d$-dimensional discrete Fourier transforms of complex data.

Table 219: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y scale</td>
<td></td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>SCFTD</td>
</tr>
<tr>
<td>Long-precision complex</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 34.
3. On certain platforms that have NVIDIA GPUs, transform might be offloaded to GPUs if problem size and data layout requirements are met when ESSL SMP CUDA library is used.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SCFTD</td>
</tr>
<tr>
<td>C and C++</td>
<td>scftd</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 33). The contents of $x$ and $y$ are not used or changed.

If $init = 0$, the discrete Fourier transforms of the given array is 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, \( x_{j1,j2,...,jd,mm} \) is stored in location \( j1(incx_1) + j2(incx_2) + ... + jd(incx_d) + mm(incmx) \) of the array \( X \).

Specified as: an array of (at least) length \( 1 + incx_1(n_1-1) + ... + incx_d(n_d-1) + incmx(m-1) \), containing numbers of the data type indicated in Table 219 on page 962.

\( incx \)

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; \( 0 \leq n_{1:d} \leq 1073479680 \).

\( m \)

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 \( W_{n1}, W_{n2}, ..., W_{nd} \), 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; \( isign_{1:d} \neq 0 \).

\( scale \)

is the scaling constant by which the transforms are multiplied. See “Function” on page 966 for its usage.

Specified as: a number of the data type indicated in Table 219 on page 962, where \( scale \neq 0.0 \).

\( aux1 \)

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 \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} > 7(d+1)+1 and \textit{naux1} ≥ (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 49.

\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} ≥ (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 49.

\textbf{On Return}

\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 \textit{m d}-dimensional discrete Fourier transforms. Using zero-based indexing, \(y_{k_1,k_2,\ldots,k_d,m}\) is stored in location \(k_1(incy_1) + k_2(incy_2) + \ldots + k_d(incy_d) + mm(incmy)\) of the array \textit{Y}.

Returned as: an array of (at least) length \(1 + incy_1(n_1-1) + \ldots + incy_d(n_d-1) + incmy(m-1)\), containing numbers of the data type indicated in Table 219 on page 962.

\textit{aux1}

is the working storage for this subroutine, where:

If \textit{init} > 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If \textit{init} = 0, its contents are unchanged.

Returned as: the contents are not relevant.

\textit{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 \textit{naux1} that is at least 7(d+1)+1 (but insufficient for the problem).
- There were no other errors.

Otherwise, it remains unchanged.
***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 \( \text{init} > 0 \) 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. 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 \( \text{incx}_i \) and \( \text{incy}_i \) for \( i = 1, \ldots, d \) must be equal, and \( \text{incmx} \) and \( \text{inmy} \) must be equal. 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 “Vector concepts” on page 71.

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 49.

The following notes apply to the ESSL SMP CUDA library:

1. Both initialization and computation steps are performed in the same way for the ESSL SMP CUDA library subroutines. The subroutines check whether the transforms can be done on GPUs. For GPU-supported transforms, if ESSL_CUDA_HYBRID=no, subroutines offload computations to GPUs. If ESSL_CUDA_HYBRID=yes, subroutines run computations on either CPUs or GPUs based on predicted performance. Performance might be affected if transforms are forced to run on GPUs.

2. Subroutines use the cuFFT library to offload computations to GPUs. For more information about cuFFT library, see the cuFFT library user guide.

3. Additional data layout requirements must be met to offload transform to GPUs. Sequence must be stored contiguously, so either the first or the last dimension strides must be 1 for both input and output arrays. Sequences cannot overlap in memory (for example, no row transforms). For three-dimensional transforms, data must be stored as a three-dimensional array.

4. Subroutines determine whether to use GPUs by predicting the relative performances of CPUs and GPUs based on the size of transforms, dimensions of transforms, number of transforms, number of CPU cores and their versions, GPU versions, and the pinning status of host memory. Only large transforms are offloaded to GPUs, if ESSL_CUDA_HYBRID=yes.

5. Available GPU memory can limit the size of transform that can be offloaded to GPUs.

6. For systems with more than 1 GPU, subroutines determine the number of GPUs that can be used to achieve better performance. Subroutines can use the cuFFT XT APIs to perform a single transform on multiple GPUs. The cuFFT XT APIs can be performed only on two or four GPUs for one-dimensional transforms.

7. If the transform is offloaded to GPUs, the only requirement for auxiliary storage is the minimum naux1 = 7*(d+1)+1. If the auxiliary storage is allocated by using the Processor-Independent Formulas for naux1 and naux2, subroutines can run successfully regardless of whether GPUs are used. To save memory, if GPUs are used, you can set naux1= 7*(d+1)+1 and naux2=0, and let error handling return...
the auxiliary storage to your program. Unchanged value of the naux1 parameter means that computations will be offloaded to GPUs and additional auxiliary storage is not required any more.

8. When you use the ESSL SMP CUDA library, the number of threads that is specified must not be changed between calls to this subroutine with init > 0 and init = 0.

Formulas

Processor-Independent Formulas for SCFTD for NAUX1 and NAUX2:

**NAUX1 Formulas**

If \( \max(n_1, n_2, \ldots, n_d) \leq 2048 \), \( naux1 = 30000d \).

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 DCFTD for NAUX1 and NAUX2:

**NAUX1 Formulas**

If \( \max(n_1, n_2, \ldots, n_d) \leq 1024 \), \( naux1 = 30000d \).

If \( \max(n_1, n_2, \ldots, n_d) > 1024 \), \( naux1 = 60000d + 28.24(n_1 + \ldots + n_d) \).

**NAUX2 Formulas**

If \( \max(n_1, n_2, \ldots, n_d) < 252 \), \( naux2 = 20000 \).

If \( \max(n_1, n_2, \ldots, n_d) \geq 252 \), \( naux2 = 20000 + (2r + 256)(s + 17.12) \).

where:

\( r = \max(n_1, n_2, \ldots, n_d) \) and

\( s = \min(64, r) \)

Function

The set of \( m \) \( d \)-dimensional discrete Fourier transforms of complex data in array \( X \), with results going into array \( Y \), is expressed as follows:

\[
Y_{k_1, k_2, \ldots, k_d, l} = scale \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \ldots \sum_{j_d=0}^{n_d-1} X_{j_1, j_2, \ldots, j_d, l} W_{n_1}^{(i, 0, \ldots, 0)} W_{n_2}^{(i, 1, 0, \ldots, 0)} \ldots W_{n_d}^{(i, 0, \ldots, 1)}
\]

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:

$$W_{n_l} = e^{-2\pi i (\sqrt{-1}) / n_l}$$

for:

\(l = 1, \ldots, d\)

and where:

\(x_{k_1, k_2, \ldots, k_d, mm}\) are elements of the \(d\)-dimensional sequences in array \(X\).

\(y_{k_1, k_2, \ldots, k_d, mm}\) 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[“6” on page 1253], [“8” on page 1253], [“13” on page 1254], and [“32” on page 1254].

Two invocations of this subroutine are necessary:

1. With \(\text{init} > 0\), the subroutine tests and initializes arguments of the program, setting up the \(\text{aux1}\) working storage.
2. With \(\text{init} = 0\), the subroutine checks that the initialization arguments in the \(\text{aux1}\) working storage correspond to the present arguments, and if so, performs the calculation of the Fourier transforms.

If \(n_l = 0\) 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, \(\text{naux2} = 0\), and unable to allocate work area.

**Computational Errors**

None

**Input-Argument Errors**

1. \(\text{init} < 0\) or \(\text{init} > 2\)
2. \(d < 1\) or \(d > 3\)
3. \(\text{incx}_i \leq 0\) (\(i = 1, \ldots, d\))
4. \(\text{incmx} \leq 0\)
5. \(\text{incy}_i \leq 0\) (\(i = 1, \ldots, d\))
6. \(\text{incmy} \leq 0\)
7. \(n_i < 0\) or \(n_i > 1073479680\) (\(i = 1, \ldots, d\))
8. \(m < 0\)
9. \(\text{isign}_i = 0\) (\(i = 1, \ldots, d\))
10. \(\text{scale} = 0.0\)
11. \(\text{naux1} < 7(d+1)+1\)
12. \(\text{naux1}\) 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 \(\text{naux2} \neq 0\), and \(\text{naux2}\) is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
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:

\[ 2\pi(\sqrt{-1}) \frac{jk}{n} \]

for \( j = 0, 1, \ldots, n-1 \) with \( n = 8 \), and the single frequencies \( k = 0, 1, 2, \) and 3.

\textbf{Note:} X is the same input array used in Example 1.

The arrays are declared as follows:

\begin{verbatim}
COMPLEX*16  X(0:31),Y(0:31)
REAL*8      AUX1(10000),AUX2(1)
\end{verbatim}

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.

\textbf{Note:} Because NAUX2 = 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

\begin{verbatim}
INIT  D  X  INCX INCMX Y  INCY INCMY N  M  ISIGN SCALE AUX1  NAUX1  AUX2 NAUX2
\end{verbatim}

\begin{verbatim}
CALL DCFTD(INIT, 1, X, INCX, 8,  Y, INCY, 8,  N, 4, ISIGN, 1.0, AUX1, 10000, AUX2, 0)
\end{verbatim}

\textbf{Call:} DCFTD(INIT, 1, X, INCX, 8, Y, INCY, 8, N, 4, ISIGN, 1.0, AUX1, 10000, AUX2, 0)

\textbf{Call:} DCFTD(INIT, 0, X, INCX, 8, Y, INCY, 8, N, 4, ISIGN, 1.0, AUX1, 10000, AUX2, 0)

\textbf{Output:}

\begin{verbatim}
Y contains the following four sequences:
\end{verbatim}

\begin{verbatim}
(8.0000, 0.0000) (0.0000, 0.0000) (0.0000, 0.0000) (0.0000, 0.0000)
(0.0000, 0.0000) (8.0000, 0.0000) (0.0000, 0.0000) (0.0000, 0.0000)
(0.0000, 0.0000) (0.0000, 0.0000) (8.0000, 0.0000) (0.0000, 0.0000)
(0.0000, 0.0000) (0.0000, 0.0000) (0.0000, 0.0000) (8.0000, 0.0000)
(0.0000, 0.0000) (0.0000, 0.0000) (0.0000, 0.0000) (0.0000, 0.0000)
(0.0000, 0.0000) (0.0000, 0.0000) (0.0000, 0.0000) (0.0000, 0.0000)
(0.0000, 0.0000) (0.0000, 0.0000) (0.0000, 0.0000) (0.0000, 0.0000)
(0.0000, 0.0000) (0.0000, 0.0000) (0.0000, 0.0000) (0.0000, 0.0000)
\end{verbatim}
Example 2

This example shows how to compute a three-dimensional transform. In this example, $\text{INCX} \geq \text{INCY}$, 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, $Y(\text{INCY}(3),N(3))$.

For equivalence, it is required that $\text{INCX}(2) \geq \text{INCY}(2)$ and $\text{INCX}(3) \geq \text{INCY}(3)$. Therefore, INCX(2) and 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, INCX(3) must be a multiple of INCX(2). Therefore, its value is set as $\text{INCX}(3) = 65(\text{INCX}(2)) = 2080$.

The arrays are declared as follows:

```
COMPLEX*8     X(32,65,40),Y(2056,40)
REAL*8        AUX1(90000),AUX2(1),SCALE
```

Arrays $X$ and $Y$ are made equivalent by the following statement, making them occupy the same storage:

```
EQUIVALENCE (X,Y)
```

**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>INCX</th>
<th>INCMX</th>
<th>Y</th>
<th>INCY</th>
<th>INCMY</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>
<tbody>
<tr>
<td>CALL SCFTD(INIT, 3, X, INCX, 0, Y, INCY, 0, N, 1, ISIGN, 1.0, AUX1, 90000, 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>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

| INIT | = 1 (for initialization) |
| INIT | = 0 (for computation) |
| INCX is an array of length $d$. |
| INCX(1) = 1 |
| INCX(2) = 32 |
| INCX(3) = 2080 |
| INCY is an array of length $d$. |
| INCY(1) = 1 |
| INCY(2) = 32 |
| INCY(3) = 2056 |
| N is an array of length $d$. |
| N(1) = 32 |
| 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 $(1.0,2.0)$ in location $X(1,1,1)$ and $(0.0,0.0,0.0)$ 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>
<thead>
<tr>
<th>Table 220: Data Types.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Types</strong></td>
</tr>
<tr>
<td>( X, ) scale ( 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>
<tr>
<td><strong>Subroutine</strong></td>
</tr>
<tr>
<td>SRCFTD</td>
</tr>
<tr>
<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 34.
3. On certain platforms with NVIDIA GPUs, transforms can be offloaded to GPUs if problems size and data layout requirements are fulfilled when ESSL SMP CUDA library is used.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SRCFTD</td>
</tr>
<tr>
<td>C and C++</td>
<td>srcftd</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 33). 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 aux2 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 real arrays to be transformed. Using zero-based indexing, $X_{j_1,j_2,...,j_d,mm}$ is stored in location $j_1(\text{incx}_1) + j_2(\text{incx}_2) + ... + j_d(\text{incx}_d) + mm(\text{incmx})$ of the array X.

Specified as: an array of (at least) length $1 + \text{incx}_1(n_1-1) + ... + \text{incx}_d(n_d-1) + \text{incmx}(m-1)$, containing numbers of the data type indicated in Table 220 on page 970.

**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; $\text{incx}_{1:d} > 0$.

**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.

**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; $\text{incy}_{1:d} > 0$.

**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; $m > 0$.

**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 $W_{n_1}, W_{n_2}, ..., W_{n_d}$, 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; $\text{isign}_{1:d} \neq 0$.

**scale** is the scaling constant by which the transforms are multiplied. See “Function” on page 974 for its usage.

Specified as: a number of the data type indicated in Table 220 on page 970, where scale $\neq 0.0$.

**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 aux1 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 974). 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 49.

$aux2$
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 49.

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, $y_{k_1,k_2,...,kd,mm}$ is stored in location $k_1(incy_1) + k_2(incy_2) + ... + kd(incy_d) + mm(incmy)$ 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 $k_1 = 0, 1, ..., n_1/2$.
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 220 on page 970.

$aux1$
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 “Vector concepts” on page 71.

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 49.

The following notes apply to the ESSL SMP CUDA library:

1. Both initialization and computation steps are performed in the same way for the ESSL SMP CUDA library subroutines. The subroutines check whether the transforms can be done on GPUs. For GPU-supported transforms, if \(ESSL\_CUDA\_HYBRID=no\), subroutines offload computations to GPUs. If \(ESSL\_CUDA\_HYBRID=yes\), subroutines run computations on either CPUs or GPUs based on predicted performance. Performance might be affected if transforms are forced to run on GPUs.

2. Subroutines use the cuFFT library to offload computations to GPUs. For more information about cuFFT library, see the cuFFT library user guide.

3. Only forward-transform computations (time to frequency) can be offloaded to GPUs.

4. Additional data layout requirements must be met to offload transform to GPUs. Sequence must be stored contiguously, so either the first or the last dimension strides must be 1 for both input and output arrays. Sequences cannot overlap in memory (for example, no row transforms). For three-dimensional transforms, data must be stored as a three-dimensional array.

5. Subroutines determine whether to use GPUs by predicting the relative performances of CPUs and GPUs based on the size of transforms, dimensions of transforms, number of transforms, number of CPU cores and their versions, GPU versions, and the pinning status of host memory. Only large transforms are offloaded to GPUs, if \(ESSL\_CUDA\_HYBRID=yes\).

6. Available GPU memory can limit the size of transform that can be offloaded to GPUs.

7. For systems with more than 1 GPU, subroutines determine the number of GPUs that can be used to achieve better performance. However, a single transform can be offloaded to only one GPU.

8. If the transform is offloaded to GPUs, the only requirement for auxiliary storage is the minimum \(naux1 = 4*d+11\). If the auxiliary storage is allocated by using the Processor-Independent Formulas for \(naux1\) and \(naux2\), subroutines can run successfully regardless of whether GPUs are used. To save memory, if GPUs are used, you can set \(naux1 = 4*d+11\) and \(naux2 = 0\), and let error handling return the auxiliary Fourier Transforms, Convolutions and Correlations, and Related Computations 973
storage to your program. Unchanged value of the naux1 parameter means that computations will be
offloaded to GPUs and additional auxiliary storage is not required any more.

9. When you use the ESSL SMP CUDA library, the number of threads that is specified must not be
changed between calls to this subroutine with init > 0 and init = 0.

Formulas

Processor-Independent Formulas for SRCFTD for NAUX1 and NAUX2:

NAUX1 Formulas

If max(n₁, n₂, ..., n₉) ≤ 2048, naux1 = 60000d.
If max(n₁, n₂, ..., n₉) > 2048, naux1 = 60000d+14.12(n₁+...+n₉).

NAUX2 Formulas

If max(n₁, n₂, ..., n₉) < 252, naux2 = 20000.
If max(n₁, n₂, ..., n₉) ≥ 252, naux2 = 20000+(r+256)(s+8.56).
where:
   r = max(n₁, n₂, ..., n₉) and
   s = min(64,r)

Processor-Independent Formulas for DRCFTD for NAUX1 and NAUX2:

NAUX1 Formulas

If max(n₁, n₂, ..., n₉) ≤ 1024, naux1 = 60000d.
If max(n₁, n₂, ..., n₉) > 1024, naux1 = 60000d+28.24(n₁+...+n₉).

NAUX2 Formulas

If max(n₁, n₂, ..., n₉) < 252, naux2 = 20000.
If max(n₁, n₂, ..., n₉) ≥ 252, naux2 = 20000+(2r+256)(s+17.12).
where:
   r = max(n₁, n₂, ..., n₉) 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:

\[ y_{k_1,k_2,...,k_d} = scale \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_d=0}^{n_d-1} x_{j_1,j_2,...,j_d} W_{n_1}^{(isgn_1)(j_1)(k_1)} W_{n_2}^{(isgn_2)(j_2)(k_2)} \cdots W_{n_d}^{(isgn_d)(j_d)(k_d)} \]

for:
   k₁ = 0,...,n₁-1
   k₂ = 0,...,n₂-1
   .
   .
   .

IBM Engineering and Scientific Subroutine Library for Linux on POWER   Version 6.1: ESSL Guide and Reference
where:

$$W_{n_i} = e^{-2\pi(i-1)/n_i}$$

for:

$l = 1,\ldots,d$

and where:

$x_{i1,i2,\ldots,idd}$ are elements of the $d$-dimensional sequences in array $X$.

$y_{k1,k2,\ldots,kdd}$ 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[“6” on page 1253], [“8” on page 1253], [“13” on page 1254], and [“32” on page 1254].

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 $n_i > 1073497760$ ($i = 1,\ldots,d$)
8. $m < 0$
9. $isign_i = 0$ ($i = 1,\ldots,d$)
10. $scale = 0.0$
11. $naux1 < 4d+11$
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:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL*4</td>
<td>X</td>
<td>X(0:100)</td>
</tr>
<tr>
<td>COMPLEX*8</td>
<td>Y</td>
<td>Y(0:50)</td>
</tr>
<tr>
<td>REAL*8</td>
<td>AUX1</td>
<td>AUX1(1000), AUX2</td>
</tr>
</tbody>
</table>

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:**

```
AUX2       INIT  D   X    INCX  INCMX  Y    INCY  INCMY  N    M    ISIGN   SCALE   AUX1   NAUX1
|     | |   |    |    | |    |    |   |    |      |      |      |
|      | |
| CALL SRCFTD(INIT, 1  X,  INCX, 16,   Y, INCY, 9,    N,  4, ISIGN,  SCALE, AUX1,  1000, |
| AUX2,    0  ) |
```

**INIT** = 1 (for initialization)
**INIT** = 0 (for computation)
**INCX** is an array of length \( d \).
**INCY** is an array of length \( d \).
**N** is an array of length \( d \).
**ISIGN** is an array of length \( d \).
**SCALE** = \( 1.0 / 16 \)

X contains the following four sequences:

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0000</td>
<td>0.9239</td>
<td>0.7071</td>
<td>0.3827</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0000</td>
<td>0.7071</td>
<td>0.0000</td>
<td>-0.7071</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0000</td>
<td>0.3827</td>
<td>-0.7071</td>
<td>-0.9239</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0000</td>
<td>0.0000</td>
<td>-1.0000</td>
<td>0.0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0000</td>
<td>-0.3827</td>
<td>-0.7071</td>
<td>0.9239</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0000</td>
<td>-0.7071</td>
<td>0.0000</td>
<td>0.7071</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output:

Y contains the following four sequences:

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(1.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

976 IBM Engineering and Scientific Subroutine Library for Linux on POWER  Version 6.1: ESSL Guide and Reference
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 INCX.) Using the transform lengths \((N(1) = 33, N(2) = 64, \text{and } N(3) = 40)\) 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 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), INCY(2), INCX(3) and INCY(3) are set as follows:

<table>
<thead>
<tr>
<th>INCY(2)</th>
<th>INCX(2)</th>
<th>INCY(3)</th>
<th>INCX(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>36</td>
<td>1160</td>
<td>2320</td>
</tr>
</tbody>
</table>

The arrays are declared as follows:

```
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:

```
EQUIVALENCE (X,Y)
```

**Note:** Because \(NAUX2=0\), this subroutine dynamically allocates the \(AUX2\) working storage.

Call Statement and Input:

```
AUX2   NAUX2   INIT   D   X   INCX   INCX   Y   INCX   INCMX   Y   INCMY   N   M   ISIGN   SCALE   AUX1   NAUX1
|      |   |      |      |      |  | |      | | | | | | | | | | | |
|      |   | CALL SRCFTD(INIT, 3, X, INCX, 0, Y, INCY, 0 N, 1, ISIGN, 1.0, AUX1, 5000, |
```

| INIT = 1 (for initialization) |
| INIT = 0 (for computation) |
| INCX is an array of length \(d\). |
| INCX(1) = 1 |
| INCX(2) = 36 |
| INCX(3) = 2320 |
| INCY is an array of length \(d\). |
| INCY(1) = 1 |
| INCY(2) = 18 |
| INCY(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) 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>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<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>

**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 34.
3. On certain platforms that have NVIDIA GPUs, transform might be offloaded to GPUs if problem size and data layout requirements are met when ESSL SMP CUDA library is used.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SCRFTD</td>
</tr>
<tr>
<td>C and C++</td>
<td>scrftd</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 33). The contents of \( x \) and \( y \) are not used or changed.
If \( \text{init} = 0 \), the discrete Fourier transforms of the given array are computed. The only arguments that may change after initialization are \( x \), \( y \), and \( \text{aux2} \). The arguments \( d, \text{incx}, \text{incmx}, \text{incy}, \text{incmy}, n, m, \text{isign}, \text{scale}, \text{aux1}, \text{naux1}, \) and \( \text{naux2} \) 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, \( x_{i_1,i_2,...,i_d,m} \) is stored in location \( j (\text{incx}_1) + j_2 (\text{incx}_2) + ... + j_d (\text{incx}_d) + m (\text{incmx}) \) of the array \( X \). Due to complex conjugate symmetry, the input consists of only the first \( n_1/2+1 \) values along the first dimension of the array, for \( j_1 = 0, 1, ..., n_1/2 \).

Specified as: an array of (at least) length \( 1 + \text{incx}_1 (n_1-1) + ... + \text{incx}_d (n_d-1) + \text{incmx} (m-1) \), containing numbers of the data type indicated in Table 221 on page 978.

\( \text{incx} \)

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; \( \text{incx}_{1:d} > 0 \).

\( \text{incmx} \)

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.

\( \text{incy} \)

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; \( \text{incy}_{1:d} > 0 \).

\( \text{incmy} \)

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; \( m > 0 \).

\( \text{isign} \)

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 \( W_{n_1}, W_{n_2}, ..., W_{n_d} \), 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; \( \text{isign}_{1:d} \neq 0 \).

\( \text{scale} \)

is the scaling constant by which the transforms are multiplied. See “Function” on page 983 for its usage.
Specified as: a number of the data type indicated in Table 221 on page 978, 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* ≥ (minimum value required for successful processing). To determine a sufficient value, use the processor-independent formulas (see “Formulas” on page 982). 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 49.

**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* ≥ (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 49.

**On Return**

**x**

has the following meaning, where:
If *init* > 0, this argument is not used, and its contents remain unchanged.
If *init* = 0, this argument is not used, and its contents remain unchanged if one of the following is true:
- *d* = 1
- *incx1* = 1 and *incy1* = 1
Otherwise, *x* is overwritten; that is, the original input is not preserved.

**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 *md*-dimensional discrete Fourier transforms of complex conjugate even data. Using zero-based indexing, \( y_{k1, k2, \ldots, kd, mm} \) is stored in location \( k1 \cdot \text{incy1} + k2 \cdot \text{incy2} + \ldots + kd \cdot \text{incyd} + mm \cdot \text{incmy} \) of the array *Y*.

Returned as: an array of (at least) length \( 1 + \text{incy1} \cdot (n1-1) + \ldots + \text{incyd} \cdot (nd-1) + \text{incmy} \cdot (m-1) \), containing numbers of the data type indicated in Table 221 on page 978.
**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 \( i = 2,\ldots,d \)
   - \( \min(\text{incmy}, \text{incy}_2,\ldots,\text{incy}_d) \geq 2(n_1/2+1) \)

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 \( i = 2,\ldots,d \)
   - \( \text{incmy} \) must be equal to \( 2(\text{incmx}) \) 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 “Vector concepts” on page 71.

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 49.

The following notes apply to the ESSL SMP CUDA library:

1. Both initialization and computation steps are performed in the same way for the ESSL SMP CUDA library subroutines. The subroutines check whether the transforms can be done on GPUs. For GPU-supported transforms, if ESSL_CUDA_HYBRID=no, subroutines offload computations to GPUs. If
ESSL_CUDA_HYBRID=yes, subroutines run computations on either CPUs or GPUs based on predicted performance. Performance might be affected if transforms are forced to run on GPUs.

2. Subroutines use the cuFFT library to offload computations to GPUs. For more information about cuFFT library, see the cuFFT library user guide.

3. Only backward-transform computations (frequency to time) can be offloaded to GPUs.

4. Additional data layout requirements must be met to offload transform to GPUs. Sequence must be stored contiguously, so either the first or the last dimension strides must be 1 for both input and output arrays. Sequences cannot overlap in memory (for example, no row transforms). For three-dimensional transforms, data must be stored as a three-dimensional array.

5. Subroutines determine whether to use GPUs by predicting the relative performances of CPUs and GPUs based on the size of transforms, dimensions of transforms, number of transforms, number of CPU cores and their versions, GPU versions, and the pinning status of host memory. Only large transforms are offloaded to GPUs, if ESSL_CUDA_HYBRID=yes.

6. Available GPU memory can limit the size of transform that can be offloaded to GPUs.

7. For systems with more than 1 GPU, subroutines determine the number of GPUs that can be used to achieve better performance. However, a single transform can be offloaded to only one GPU.

8. If the transform is offloaded to GPUs, the only requirement for auxiliary storage is the minimum naux1 = \(4^d+11\). If the auxiliary storage is allocated by using the Processor-Independent Formulas for naux1 and naux2, subroutines can run successfully regardless of whether GPUs are used. To save memory, if GPUs are used, you can set naux1 = \(4^d+11\) and NAUX2=0, and let error handling return the auxiliary storage to your program. Unchanged value of the naux1 parameter means that computations will be offloaded to GPUs and additional auxiliary storage is not required any more.

9. When you use the ESSL SMP CUDA library, the number of threads that is specified must not be changed between calls to this subroutine with init > 0 and init = 0.

Formulas

Processor-Independent Formulas for SCRFTD for NAUX1 and NAUX2:

**NAUX1 Formulas**

- If \(\max(n_1,n_2,...,n_d) \leq 2048\), \(naux1 = 60000d\).
- If \(\max(n_1,n_2,...,n_d) > 2048\), \(naux1 = 60000d+14.12(n_1+...+n_d)\).

**NAUX2 Formulas**

- If \(\max(n_1,n_2,...,n_d) < 252\), \(naux2 = 20000\).
- If \(\max(n_1,n_2,...,n_d) \geq 252\), \(naux2 = 20000+(r+256)(s+8.56)\).

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 \(\max(n_1,n_2,...,n_d) \leq 1024\), \(naux1 = 60000d\).
- If \(\max(n_1,n_2,...,n_d) > 1024\), \(naux1 = 60000d+28.24(n_1+...+n_d)\).

**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:

\[ r = \max(n_1, n_2, \ldots, n_d) \quad \text{and} \quad s = \min(64, r) \]

**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:

\[
y_{k_1, k_2, \ldots, k_d, i} = scale \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_d=0}^{n_d-1} x_{j_1, j_2, \ldots, j_d, i} W_{n_1}^{(isign_1)(j_1)(k_1)} W_{n_2}^{(isign_2)(j_2)(k_2)} \cdots W_{n_d}^{(isign_d)(j_d)(k_d)}
\]

for:

\[ k_1 = 0, \ldots, n_1 - 1 \]
\[ k_2 = 0, \ldots, n_2 - 1 \]
\[ \vdots \]
\[ k_d = 0, \ldots, n_d - 1 \]
\[ i = 0, \ldots, m - 1 \]

where:

\[
W_{n_l} = e^{-2\pi(\sqrt{-1}) / n_l}
\]

for:

\[ l = 1, \ldots, d \]

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 \( scale = 1.0 \) 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 ["6" on page 1253], ["8" on page 1253], ["13" on page 1254], and ["32" on page 1254].

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. $\text{init} < 0$ or $\text{init} > 2$
2. $d < 1$ or $d > 3$
3. $\text{incx}_i \leq 0$ ($i = 1, ..., d$)
4. $\text{incmx} \leq 0$
5. $\text{incy}_i \leq 0$ ($i = 1, ..., d$)
6. $\text{incmy} \leq 0$
7. $n_i < 0$ or $n_i > 1073479680$ ($i = 1, ..., d$)
8. $m \leq 0$
9. $\text{isign}_i = 0$ ($i = 1, ..., d$)
10. $\text{scale} = 0.0$
11. $\text{naux1} < 4d+11$.
12. $\text{naux1}$ 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 $\text{naux2} \neq 0$, and $\text{naux2}$ is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
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 $\text{AUX1}$ 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:

```plaintext
CALL SCRTDF( INIT, 1, X, INCX, 7, Y, INCY, 12, N, 1, ISIGN, 1.0, AUX1, 100, NAUX1 )
```

**INIT** = 1 (for initialization)
**INIT** = 0 (for computation)
INCX is an array of length $d$.
INCY is an array of length $d$.
N is an array of length $d$.
ISIGN is an array of length $d$.
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:

```
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:

```
<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></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>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<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>
<tr>
<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>
<tr>
<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>
<tr>
<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>
<tr>
<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>
<tr>
<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>
<tr>
<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>
<tr>
<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>
<tr>
<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>
<tr>
<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>
<tr>
<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>

INIT = 1 (for initialization)
INIT = 0 (for computation)

INCX is an array of length d.

INCX(1) = 1
INCX(2) = 4
INCX(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>
<thead>
<tr>
<th>Table 222: Data Types.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Types</strong></td>
</tr>
<tr>
<td>X, Y scale</td>
</tr>
<tr>
<td>Short-precision complex</td>
</tr>
<tr>
<td>Long-precision complex</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 34.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SCFT</td>
</tr>
<tr>
<td>C and C++</td>
<td>scft</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 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 $n$.
  - Specified as: an array of (at least) length $1+(n-1)inc1x+(m-1)inc2x$, containing numbers of the data type indicated in Table 222 on page 986.
**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 each sequence in array Y. (If m = 1, this argument is ignored.) Specified as: an integer; inc2y > 0.

**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 956. 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 56.

**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<sub>n</sub>, 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 989 for its usage.  
Specified as: a number of the data type indicated in Table 222 on page 986, 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 > 7 (32-bit integer arguments) or 13 (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 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 49.

**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.

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 49.

**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, each of length $n$.

Returned as: an array of (at least) length $1+(n-1)inc1y+(m-1)inc2y$, containing numbers of the data type indicated in Table 222 on page 986.

$aux1$ 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. The preferred value for $inc2x$ and $inc2y$ is $n$. This implies that sequences are stored one after another without any gap.

   It is possible to specify sequences in the transposed form—that is, as rows of a two-dimensional array. In this case, $inc2x$ (or $inc2y$) = 1 and $inc1x$ (or $inc1y$) is equal to the leading dimension of the array. One can specify either input, output, or both in the transposed form by specifying appropriate values for the stride parameters. For selecting optimal values of $inc1x$ and $inc1y$ for _CFT, you should use “STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)” on page 1195. Example 1 in the STRIDE subroutine description explains how it is used for _CFT.

   If you specify the same array for $X$ and $Y$, then $inc1x$ and $inc1y$ must be equal, and $inc2x$ and $inc2y$ must be equal. In this case, output overwrites input. If $m = 1$, the $inc2x$ and $inc2y$ values are not used by the subroutine. If you specify different arrays for $X$ and $Y$, they must have no common elements; otherwise, results are unpredictable. See “Vector concepts” on page 71.

**Formulas**

Processor-Independent Formulas for SCFT for NAUX1 and NAUX2:

**NAUX1 Formulas**

**For 32-bit integer arguments:**

If $n \leq 8192$, use $naux1 = 20000$.

If $n > 8192$, use $naux1 = 20000+1.14n$. 

988 IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and Reference
For 64-bit integer arguments:

If \( n \leq 8192 \), use \( naux1 = 30000 \).
If \( n > 8192 \), use \( naux1 = 30000 + 1.14n \).

**NAUX2 Formulas**

If \( n \leq 8192 \), use \( naux2 = 20000 \).
If \( n > 8192 \), use \( naux2 = 20000 + 1.14n \).

For the transposed case, where \( inc2x = 1 \) or \( inc2y = 1 \), and where \( n \geq 252 \), add the following to the above storage requirements:

\[(n+256)(\min(64, m))\]

**Processor-Independent Formulas for DCFT for NAUX1 and NAUX2:**

**NAUX1 Formulas**

**For 32-bit integer arguments:**

If \( n \leq 2048 \), use \( naux1 = 20000 \).
If \( n > 2048 \), use \( naux1 = 20000 + 2.28n \).

**For 64-bit integer arguments:**

If \( n \leq 2048 \), use \( naux1 = 30000 \).
If \( n > 2048 \), use \( naux1 = 30000 + 2.28n \).

**NAUX2 Formulas**

If \( n \leq 2048 \), use \( naux2 = 20000 \).
If \( n > 2048 \), use \( naux2 = 20000 + 2.28n \).

For the transposed case, where \( inc2x = 1 \) or \( inc2y = 1 \), 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:

\[y_{ki} = \text{scale} \sum_{j=0}^{n-1} x_{ji} W_n^{(\text{isign})jk}\]

for:

\( k = 0, 1, ..., n-1 \)
\( i = 1, 2, ..., m \)

where:

\[W_n = e^{-2\pi(\sqrt{-1})i/n}\]

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.

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 1253], [“4” on page 1253], [“5” on page 1253], [“27” on page 1254], and [“28” on page 1254].

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. $inc1x, inc2x, inc1y, or inc2y \leq 0$  
3. $m \leq 0$  
4. $isign = 0$  
5. $scale = 0.0$  
6. The subroutine has not been initialized with the present arguments.  
7. The length of the transform in $n$ is not an allowable value. Return code 1 is returned if error 2030 is recoverable.  
8. $naux1 \leq 7$  
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:

$$2\pi \left( \sqrt{-1} \right)^jk/n$$

for $j = 0, 1, …, n-1$ with $n = 8$, and the single frequencies $k = 0, 1, 2, and 3$. The arrays are declared as follows:

<table>
<thead>
<tr>
<th>Complex*8</th>
<th>X(0:1023)</th>
<th>Y(0:1023)</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL*8</td>
<td>AUX1(1693)</td>
<td>AUX2(1)</td>
</tr>
</tbody>
</table>

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>X</th>
<th>INC1X</th>
<th>INC2X</th>
<th>Y</th>
<th>INC1Y</th>
<th>INC2Y</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>
<tbody>
<tr>
<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></td>
<td></td>
</tr>
</tbody>
</table>

CALL SCFT(INIT, X, 1, 8, Y, 1, 8, 8, 4, 1, SCALE, AUX1, 1693, AUX2, 0)

INIT = 1 (for initialization)
INIT = 0 (for computation)
SCALE = 1.0

X contains the following four sequences:

\[
\begin{align*}
(1.0000, 0.0000) & \quad (1.0000, 1.0000) \\
(0.7071, 0.7071) & \quad (0.0000, 0.0000) \\
(1.0000, 0.0000) & \quad (-1.0000, 0.0000) \\
(0.0000, 1.0000) & \quad (0.0000, -1.0000) \\
(1.0000, 0.0000) & \quad (0.0000, -1.0000) \\
(0.7071, -0.7071) & \quad (0.0000, 0.0000) \\
\end{align*}
\]

Output:

Y contains the following four sequences:

\[
\begin{align*}
(8.0000, 0.0000) & \quad (0.0000, 0.0000) \\
(0.0000, 0.0000) & \quad (0.0000, 0.0000) \\
(0.0000, 0.0000) & \quad (0.0000, 0.0000) \\
(0.0000, 0.0000) & \quad (0.0000, 0.0000) \\
(0.0000, 0.0000) & \quad (0.0000, 0.0000) \\
(0.0000, 0.0000) & \quad (0.0000, 0.0000) \\
\end{align*}
\]

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:

<table>
<thead>
<tr>
<th>INIT</th>
<th>X</th>
<th>INC1X</th>
<th>INC2X</th>
<th>Y</th>
<th>INC1Y</th>
<th>INC2Y</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>
<tbody>
<tr>
<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></td>
<td></td>
</tr>
</tbody>
</table>

CALL SCFT(INIT, X, 1, 8, Y, 1, 8, 8, 4, -1, SCALE, AUX1, 1693, AUX2, 0)

INIT = 1 (for initialization)
INIT = 0 (for computation)
SCALE = 0.125
X = (same as output \( Y \) in Example 1)

Output:

Y = (same as input \( X \) in Example 1)

Example 3

This example shows an input array \( X \) with a set of four short-precision complex sequences

\[
\hat{E} = 2\pi \left( \sqrt{-1} \right)^{jk/n}
\]
for \( j = 0, 1, \ldots, n-1 \) with \( n = 12 \), and the single frequencies \( k = 0, 1, 2, \) and \( 3 \). Also, \( \text{inc1}x = \text{inc1}y = m \) and \( \text{inc2}x = \text{inc2}y = 1 \) to show how the input and output arrays can be stored in the transposed form. The arrays are declared as follows:

<table>
<thead>
<tr>
<th>COMPLEX*8</th>
<th>X (4,0:11), Y (4,0:11)</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL*8</td>
<td>AUX1(10000), AUX2(1)</td>
</tr>
</tbody>
</table>

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 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) |

INIT = 1 (for initialization)
INIT = 0 (for computation)
SCALE = 1.0

X contains the following four sequences:

\[
\begin{align*}
(1.0000, 0.0000) & \quad (1.0000, 0.0000) \\
(1.0000, 0.0000) & \quad (0.8660, 0.5000) \\
(1.0000, 0.0000) & \quad (0.5000, 0.8660) \\
(1.0000, 0.0000) & \quad (0.0000, 1.0000) \\
(1.0000, 0.0000) & \quad (-0.5000, 0.8660) \\
(1.0000, 0.0000) & \quad (-0.5000, 0.8660) \\
(1.0000, 0.0000) & \quad (1.0000, 0.0000) \\
(1.0000, 0.0000) & \quad (0.0000, 1.0000) \\
\end{align*}
\]

Output:

Y contains the following four sequences:

\[
\begin{align*}
(12.0000, 0.0000) & \quad (0.0000, 0.0000) \\
(0.0000, 0.0000) & \quad (12.0000, 0.0000) \\
(0.0000, 0.0000) & \quad (0.0000, 0.0000) \\
(0.0000, 0.0000) & \quad (0.0000, 0.0000) \\
(0.0000, 0.0000) & \quad (0.0000, 0.0000) \\
(0.0000, 0.0000) & \quad (0.0000, 0.0000) \\
(0.0000, 0.0000) & \quad (0.0000, 0.0000) \\
(0.0000, 0.0000) & \quad (0.0000, 0.0000) \\
\end{align*}
\]

**Example 4**

This example shows an input array X with a set of four input spike sequences exactly equal to the output of Example 3. This shows how you can compute the inverse of the transform in Example 3 by using a negative isign, giving as output the four sequences listed in the input for Example 3. First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

**Note:** Because NAUX2 = 0, this subroutine dynamically allocates the 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) |

992 IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and Reference
Example 5

This example shows how to compute a transform of a single long-precision complex sequence. It uses \( isign = 1 \) and \( scale = 1.0 \). The arrays are declared as follows:

\[
\begin{align*}
\text{COMPLEX*16} & \quad X(0:7), Y(0:7) \\
\text{REAL*8} & \quad \text{AUX1}(26), \text{AUX2}(1)
\end{align*}
\]

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:**

<table>
<thead>
<tr>
<th>INIT</th>
<th>X INC1X INC2X</th>
<th>Y INC1Y INC2Y</th>
<th>N</th>
<th>M ISIGN SCALE</th>
<th>AUX1</th>
<th>NAUX1</th>
<th>AUX2</th>
<th>NAUX2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{CALL DCFT(INIT, X , 1  ,  0 , Y , 1  ,  0 , 8 , 1 , 1 , SCALE , AUX1 , 26  , AUX2 , 0)}
\]

**INIT = 1** (for initialization)

**INIT = 0** (for computation)

**SCALE = 1.0**

\( X \) contains the following sequence:

\[
\begin{align*}
(1.0000, & \ 0.0000) \\
(0.0000, & \ 0.0000) \\
(0.0000, & \ 0.0000) \\
(0.0000, & \ 0.0000) \\
(0.0000, & \ 0.0000) \\
(0.0000, & \ 0.0000) \\
(0.0000, & \ 0.0000) \\
(0.0000, & \ 0.0000)
\end{align*}
\]

**Output:**

\[
\begin{align*}
(1.0000, & \ 0.0000) \\
(1.0000, & \ 0.0000) \\
(1.0000, & \ 0.0000) \\
(1.0000, & \ 0.0000) \\
(1.0000, & \ 0.0000) \\
(1.0000, & \ 0.0000) \\
(1.0000, & \ 0.0000) \\
(1.0000, & \ 0.0000)
\end{align*}
\]

**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>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X, scale )</td>
<td>( Y )</td>
</tr>
</tbody>
</table>
Table 223: Data Types. (continued)

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Short-precision complex</th>
<th>SRCFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>Short-precision complex</td>
<td>SRCFT</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision complex</td>
<td>DRCFT</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 34.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
</table>
| Fortran  | CALL SRCFT (init, x, inc2x, y, inc2y, n, m, isign, scale, aux1, naux1, aux2, naux2, aux3, naux3)  
           CALL DRCFT (init, x, inc2x, y, inc2y, n, m, isign, scale, aux1, naux1, aux2, naux2) |
| C and C++ | srcft (init, x, inc2x, y, inc2y, n, m, isign, scale, aux1, naux1, aux2, naux2, aux3, naux3);  
              drcft (init, x, inc2x, y, inc2y, n, m, isign, scale, aux1, naux1, aux2, naux2); |

**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, which are to be transformed. The sequences are assumed to be stored with stride 1.

  Specified as: an array of (at least) length n+(m-1)inc2x, containing numbers of the data type indicated in Table 223 on page 993. See “Notes” on page 996 for more details. (It can be declared as X(inc2x,m).)

- **inc2x**
  
is the stride between the first elements of the sequences in array X. (If m = 1, this argument is ignored.) Specified as: an integer; inc2x ≥ n.

- **y**
  
  See On Return.

- **inc2y**
  
is the stride between the first elements of the sequences in array Y. (If m = 1, this argument is ignored.) Specified as: an integer; inc2y ≥ (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 956. 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 56.

\( m \)

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; \( isign \geq 0 \) or \( isign < 0 \).

\( scale \)

is the scaling constant \( scale \). See “Function” on page 997 for its usage.

Specified as: a number of the data type indicated in Table 223 on page 993, 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; \( naux1 \geq 14 \) (32-bit integer arguments) or 27 (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 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 49.

\( aux2 \)

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, SRCFT and DRCFT dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program. Otherwise, \( naux2 \geq \) (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 49.

\( aux3 \)

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 \( \text{init} \neq 0 \), 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, \( y_{ki}, k = 0, 1, ..., n/2, i = 1, 2, ..., m \).

Returned as: an array of (at least) length \( n/2+1+(m-1)\text{inc2y} \), containing numbers of the data type indicated in Table 223 on page 993. 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. **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:

\[
\min(\text{inc2x}) = n \\
\min(\text{inc2y}) = n/2+1
\]

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 \( m = 1 \), 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 “Vector concepts” on page 71.

**Formulas**

Processor-Independent Formulas for SRCFT for NAUX1 and NAUX2:

**NAUX1 Formulas**

For 32-bit integer arguments:

- If \( n \leq 16384 \), use \( \text{naux1} = 25000 \).
- If \( n > 16384 \), use \( \text{naux1} = 20000+0.82n \).
For 64-bit integer arguments:

If \( n \leq 16384 \), use \( n_{aux1} = 35000 \).
If \( n > 16384 \), use \( n_{aux1} = 30000 + 0.82n \).

**NAUX2 Formulas**

If \( n \leq 16384 \), use \( n_{aux2} = 20000 \).
If \( n > 16384 \), use \( n_{aux2} = 20000 + 0.57n \).

Processor-Independent Formulas for DRCFT for NAUX1 and NAUX2:

**NAUX1 Formulas**

For 32-bit integer arguments:

If \( n \leq 4096 \), use \( n_{aux1} = 22000 \).
If \( n > 4096 \), use \( n_{aux1} = 20000 + 1.64n \).

For 64-bit integer arguments:

If \( n \leq 4096 \), use \( n_{aux1} = 32000 \).
If \( n > 4096 \), use \( n_{aux1} = 30000 + 1.64n \).

**NAUX2 Formulas**

If \( n \leq 4096 \), use \( n_{aux2} = 20000 \).
If \( n > 4096 \), use \( n_{aux2} = 20000 + 1.14n \).

**Function**

The set of \( m \) complex conjugate even discrete \( n \)-point Fourier transforms of real data in array \( X \), with results going into array \( Y \), is expressed as follows:

\[
y_{ki} = scale \sum_{j=0}^{n-1} x_{ji} W_n^{(Isign)jk}
\]

for:

\( k = 0, 1, ..., n-1 \)
\( i = 1, 2, ..., m \)

where:

\[
W_n = e^{-2\pi(j-1)/n}
\]

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 1253], [“5” on page 1253], [“27” on page 1254], and [“28” on page 1254].
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 \( \text{aux1} \) working storage.
2. With \( \text{init} = 0 \), the subroutine checks that the initialization arguments in the \( \text{aux1} \) 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, \( \text{naux2} = 0 \), and unable to allocate work area.

**Computational Errors**
- None

**Input-Argument Errors**
1. \( n > 37748736 \)
2. \( m \leq 0 \)
3. \( \text{inc2x} < n \)
4. \( \text{inc2y} < n/2+1 \)
5. \( \text{isign} = 0 \)
6. \( \text{scale} = 0.0 \)
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. \( \text{naux1} \leq 14 \)
10. \( \text{naux1} \) 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 \( \text{naux2} \neq 0 \), 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 an input array \( X \) with a set of \( m \) cosine sequences \( \cos(2\pi jk/n) \), \( j = 0, 1, ..., 15 \) with the single frequencies \( k = 0, 1, 2, 3 \). The Fourier transform of the cosine sequence with frequency \( k = 0 \) or \( n/2 \) has 1.0 in the 0 or \( n/2 \) position, respectively, and zeros elsewhere. For all other \( k \), the Fourier transform has 0.5 in the \( k \) position and zeros elsewhere. The arrays are declared as follows:

| REAL*4 | X(0:65535) |
| COMPLEX*8 | Y(0:32768) |
| REAL*8 | AUX1(41928), AUX2(1), AUX3(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:**

```bash
<table>
<thead>
<tr>
<th>NAUX3</th>
<th>INIT</th>
<th>X</th>
<th>IN2X</th>
<th>Y</th>
<th>IN2Y</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>
<th>AUX3</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALL SRCFT(INIT, X , 16 , Y , 9 , 16 , 4 , 1 , SCALE, AUX1 , 41928 , AUX2 , 0 , AUX3 , 0 )</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>
<td></td>
</tr>
</tbody>
</table>
```
INIT = 1 (for initialization)
INIT = 0 (for computation)
SCALE = 1.0/16

X contains the following four sequences:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>1.0000</td>
<td>0.9239</td>
<td>0.7071</td>
<td>0.3827</td>
</tr>
<tr>
<td>1.0000</td>
<td>0.7071</td>
<td>0.0000</td>
<td>-0.7071</td>
</tr>
<tr>
<td>1.0000</td>
<td>0.3827</td>
<td>-0.7071</td>
<td>-0.9239</td>
</tr>
<tr>
<td>1.0000</td>
<td>0.0000</td>
<td>1.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>1.0000</td>
<td>-0.9239</td>
<td>0.7071</td>
<td>-0.3827</td>
</tr>
<tr>
<td>1.0000</td>
<td>-1.0000</td>
<td>1.0000</td>
<td>-1.0000</td>
</tr>
<tr>
<td>1.0000</td>
<td>-0.9239</td>
<td>0.7071</td>
<td>-0.3827</td>
</tr>
<tr>
<td>1.0000</td>
<td>-0.7071</td>
<td>0.0000</td>
<td>0.7071</td>
</tr>
<tr>
<td>1.0000</td>
<td>-0.3827</td>
<td>0.7071</td>
<td>0.9239</td>
</tr>
</tbody>
</table>

Output:

Y contains the following four sequences:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(1.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
</tr>
<tr>
<td>(0.0000, 0.0000)</td>
<td>(0.5000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
</tr>
<tr>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
</tr>
<tr>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
</tr>
<tr>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
</tr>
<tr>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
</tr>
<tr>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
</tr>
<tr>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
<td>(0.0000, 0.0000)</td>
</tr>
</tbody>
</table>

Example 2

This example shows another transform computation with different data using the same initialized array AUX1 as in Example 1. The input is also a set of four cosine sequences \( \cos(2\pi jk/n) \), \( j = 0, 1, ..., 15 \) with the single frequencies \( k = 8, 9, 10, 11 \), thus including the middle frequency \( k = 8 \). The middle frequency has the value 1.0. For other frequencies, the transform has zeros, except for frequencies \( k \) and \( n-k \). Only the values for \( j = n-k \) are given in the output.

**Note:** Because NAUX2 = 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```plaintext
CALL SRCFT( 0 , X , 16 , Y , 9 , 16 , 4 , 1 , SCALE, AUX1 , 41928 , AUX2 , 0 , AUX3 , 0 )
```

SCALE = 1.0/16

X contains the following four sequences:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>-1.0000</td>
<td>-0.9239</td>
<td>-0.7071</td>
<td>-0.3827</td>
</tr>
<tr>
<td>1.0000</td>
<td>0.7071</td>
<td>0.0000</td>
<td>-0.7071</td>
</tr>
<tr>
<td>-1.0000</td>
<td>-0.3827</td>
<td>0.7071</td>
<td>0.9239</td>
</tr>
<tr>
<td>1.0000</td>
<td>-0.7071</td>
<td>0.0000</td>
<td>0.7071</td>
</tr>
<tr>
<td>-1.0000</td>
<td>0.3827</td>
<td>-0.7071</td>
<td>-0.9239</td>
</tr>
<tr>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>-1.0000</td>
</tr>
<tr>
<td>-1.0000</td>
<td>0.9239</td>
<td>-0.7071</td>
<td>0.3827</td>
</tr>
<tr>
<td>1.0000</td>
<td>-0.7071</td>
<td>0.0000</td>
<td>0.7071</td>
</tr>
<tr>
<td>-1.0000</td>
<td>0.3827</td>
<td>-0.7071</td>
<td>-0.9239</td>
</tr>
<tr>
<td>1.0000</td>
<td>0.0000</td>
<td>-1.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>-1.0000</td>
<td>-0.3827</td>
<td>0.7071</td>
<td>0.9239</td>
</tr>
</tbody>
</table>
```

Fourier Transforms, Convolutions and Correlations, and Related Computations 999
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 224: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Y, 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 34.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SCRFT (init, x, inc2x, y, inc2y, n, m, isign, scale, aux1, naux1, aux2, naux2, aux3, naux3) \nCALL DCRFT (init, x, inc2x, y, inc2y, n, m, isign, scale, aux1, naux1, aux2, naux2, naux3)</td>
</tr>
<tr>
<td>C and C++</td>
<td>scrft (init, x, inc2x, y, inc2y, n, m, isign, scale, aux1, naux1, aux2, naux2, aux3, naux3); \ndcrft (init, x, inc2x, y, inc2y, n, m, isign, scale, aux1, naux1, aux2, naux2, 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 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. Due to complex conjugate symmetry, the input consists of only the first (n/2)+1 elements of each sequence; that is, \[ x_{ji}, j = 0, 1, \ldots, \frac{n}{2}, i = 1, 2, \ldots, m \]. The sequences are assumed to be stored with stride 1.

Specified as: an array of (at least) length n/2+1+(m-1)inc2x, containing numbers of the data type indicated in Table 224 on page 1001. This array can be declared as X(inc2x,m).

*inc2x*

is the stride between the first elements of the sequences in array *X*. (If *m* = 1, this argument is ignored.) Specified as: an integer; inc2x ≥ (n/2)+1.

*y*

See On Return.

*inc2y*

is the stride between the first elements of the sequences in array *Y*. (If *m* = 1, this argument is ignored.) Specified as: an integer; inc2y ≥ n.

*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 956. 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 56.

\( m \)

is the number of sequences to be transformed.

Specified as: an integer; \( m > 0 \).

\( \text{isign} \)

controls the direction of the transform, determining the sign \( \text{Isign} \) 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 \( \text{isign} < 0 \).

\( \text{scale} \)

is the scaling constant \( \text{scale} \). See “Function” on page 1004 for its usage.

Specified as: a number of the data type indicated in Table 224 on page 1001, 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; \( \text{naux1} > 13 \) (32-bit integer arguments) or 25 (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 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 49.

\( \text{aux2} \)

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 that 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, 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, \( \text{naux2} \geq \) (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 49.

\( \text{aux3} \)

d 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* ≠ 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 224 on page 1001. See "Notes" on page 1003 for more details. (It can be declared as *Y*(inc2y,*m*).)

*aux1*

is the working storage for this subroutine, where:

If *init* ≠ 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If *init* = 0, its contents are unchanged.

Returned as: the contents are not relevant.

**Notes**

1. *aux1* should **not** be used by the calling program between calls to this subroutine with *init* ≠ 0 and *init* = 0. However, it can be reused after intervening calls to this subroutine with different arguments.

2. 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. 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 "Vector concepts" on page 71.

**Formulas**

Processor-Independent Formulas for SCRFT for NAUX1 and NAUX2:

**NAUX1 Formulas**

For **32-bit integer arguments**:

- If *n* ≤ 16384, use *naux1* = 25000.
- If *n* > 16384, use *naux1* = 20000+0.82*n*.

For **64-bit integer arguments**:

- If *n* ≤ 16384, use *naux1* = 35000.
- If *n* > 16384, use *naux1* = 30000+0.82*n*.
NAUX2 Formulas

If \( n \leq 16384 \), use \( \text{naux2} = 20000 \).
If \( n > 16384 \), use \( \text{naux2} = 20000 + 0.57n \).

Processor-Independent Formulas for DCRFT for NAUX1 and NAUX2:

**NAUX1 Formulas**

For 32-bit integer arguments:

If \( n \leq 4096 \), use \( \text{naux1} = 22000 \).
If \( n > 4096 \), use \( \text{naux1} = 20000 + 1.64n \).

For 64-bit integer arguments:

If \( n \leq 4096 \), use \( \text{naux1} = 32000 \).
If \( n > 4096 \), use \( \text{naux1} = 30000 + 1.64n \).

**NAUX2 Formulas**

If \( n \leq 4096 \), use \( \text{naux2} = 20000 \).
If \( n > 4096 \), use \( \text{naux2} = 20000 + 1.14n \).

**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:

\[
y_{ki} = \text{scale} \sum_{j=0}^{n-1} x_{ji} W_n^{(i \text{sign}) jk}
\]

for:

\( k = 0, 1, ..., n-1 \)
\( i = 1, 2, ..., m \)

where:

\[
W_n = e^{-2\pi(i/j)/n}
\]

and where:

\( x_{ji} \) are elements of the sequences in array \( X \).
\( y_{ki} \) are elements of the sequences in array \( Y \).
\( i \text{sign} \) is + or - (determined by argument \( isign \)).
\( \text{scale} \) is a scalar value.

Because of the symmetry, \( Y \) has real data. For \( \text{scale} = 1.0 \) and \( i \text{sign} \) 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 \( i \text{sign} \) being negative. See references [“1” on page 1253], [“5” on page 1253], [“27” on page 1254], and [“28” on page 1254].

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 \( \text{aux1} \) 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, \( \text{naux2} = 0 \), and unable to allocate work area.

#### Computational Errors

- None

#### Input-Argument Errors

1. \( n > 37748736 \)
2. \( m \leq 0 \)
3. \( \text{inc2x} < n/2+1 \)
4. \( \text{inc2y} < n \)
5. \( \text{scale} = 0.0 \)
6. \( \text{isign} = 0 \)
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. \( \text{naux1} \leq 13 \)
10. \( \text{naux1} \) 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 \( \text{naux2} \neq 0 \), 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 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 \( \text{AUX1} \) 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:

1. \( X \) shows the \( n/2+1 = 7 \) elements used in the computation.
2. Because \( \text{NAUX2} = 0 \), this subroutine dynamically allocates the \( \text{AUX2} \) 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  )
```

\( \text{INIT} = 1 \) (for initialization)
\( \text{INIT} = 0 \) (for computation)
\( \text{SCALE} = 1.0 \)

\( X \) contains the following sequence:

\((1.0, 0.0)\)
\((0.0, 0.0)\)
Output:

\[
Y = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
\]

**Example 2**

This example shows another transform computation with different data using the same initialized array `AUX1` as in Example 1.

**Note:** Because `NAUX2 = 0`, this subroutine dynamically allocates the `AUX2` working storage.

**Call Statement and Input:**

<table>
<thead>
<tr>
<th>INIT</th>
<th>X INC2X</th>
<th>Y INC2Y</th>
<th>N</th>
<th>M ISIGN</th>
<th>SCALE</th>
<th>AUX1 NAUX1</th>
<th>AUX2 NAUX2</th>
<th>AUX3 NAUX3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALL SCRFT(0, X, Y, 12, 1, SCALE, AUX1, 50, AUX2, AUX3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SCALE** = 1.0

`X` contains the following sequence:

\[
(1.0, 0.0) \\
(1.0, 0.0) \\
(1.0, 0.0) \\
(1.0, 0.0) \\
(1.0, 0.0) \\
(1.0, 0.0) \\
(1.0, 0.0)
\]

**Output:**

\[
Y = (12.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
\]

**Example 3**

This example shows how to compute many transforms simultaneously. The arrays are declared as follows:

```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:**

<table>
<thead>
<tr>
<th>INIT</th>
<th>X INC2X</th>
<th>Y INC2Y</th>
<th>N</th>
<th>M ISIGN</th>
<th>SCALE</th>
<th>AUX1 NAUX1</th>
<th>AUX2 NAUX2</th>
<th>AUX3 NAUX3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALL SCRFT(INIT, X, 9, Y, 16, 2, SCALE, AUX1, 50, AUX2, AUX3)</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 two sequences:

\[
(1.0, 0.0) (0.0, 0.0) \\
(1.0, 0.0) (0.0, 0.0) \\
(1.0, 0.0) (0.0, 0.0)
\]
Output:

Y contains the following two sequences:

\[
\begin{bmatrix}
16.0 & 1.0 \\
0.0 & -1.0 \\
0.0 & 1.0 \\
0.0 & -1.0 \\
0.0 & 1.0 \\
0.0 & -1.0 \\
0.0 & 1.0 \\
0.0 & -1.0 \\
0.0 & 1.0 \\
0.0 & -1.0 \\
0.0 & 1.0 \\
0.0 & -1.0 \\
0.0 & 1.0 \\
0.0 & -1.0 \\
0.0 & 1.0 \\
0.0 & -1.0 \\
0.0 & 1.0 \\
\end{bmatrix}
\]

Example 4

This example shows the same array being used for input and output. The arrays are declared as follows:

```fortran
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:

```fortran
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:

```fortran
CALL DCRFT(INIT, X , 9 , Y , 18 , 16 , 2 , -1 , SCALE, AUX1 , 50 , AUX2 , 0)
```

INIT = 1 (for initialization)
INIT = 0 (for computation)
SCALE = 0.0625

X contains the following two sequences:

\[
\begin{bmatrix}
(1.0, 0.0) & (1.0, 0.0) \\
(0.0, 1.0) & (0.0, -1.0) \\
(-1.0, 0.0) & (-1.0, 0.0) \\
(0.0, -1.0) & (0.0, 1.0) \\
(1.0, 0.0) & (1.0, 0.0) \\
(0.0, 1.0) & (0.0, -1.0) \\
(-1.0, 0.0) & (-1.0, 0.0) \\
(0.0, -1.0) & (0.0, 1.0) \\
(1.0, 0.0) & (1.0, 0.0) \\
\end{bmatrix}
\]

Output:
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>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y, scale</td>
<td>SCOSF</td>
</tr>
<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 34.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SCOSF</td>
</tr>
<tr>
<td>C and C++</td>
<td>scosf</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 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 225 on page 1008.

\( inc1x \)

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 \( n/2+1 \) 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 956. 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 56.

\( m \)

is the number of sequences to be transformed.
Specified as: an integer; \( m > 0 \).

\( scale \)

is the scaling constant \( scale \). See “Function” on page 1011 for its usage.
Specified as: a number of the data type indicated in Table 225 on page 1008, 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 49.

\( aux2 \)

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 \( n_{aux2} \) long-precision real numbers. On output, the contents are overwritten.

**\( n_{aux2} \)**

is the number of doublewords in the working storage specified in \( aux2 \).

Specified as: an integer, where:

- If \( n_{aux2} = 0 \) 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, \( n_{aux2} \geq (\text{minimum value required for successful processing}) \). 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 49.

**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 \( n/2 + 1 \) values are given in the output—that is, \( y_{ki} k = 0, 1, \ldots, n/2 \) for each \( i = 1, 2, \ldots, m \).

Returned as: an array of (at least) length \( 1 + (n/2)inc1y + (m-1)inc2y \), containing numbers of the data type indicated in Table 225 on page 1008.

**\( aux1 \)**

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. 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 \( n/2 + 1 \).

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 \_COSF, you should use “STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)” on page 1195. 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 \( \text{inc1x} \) 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 “Vector concepts” on page 71.
Formulas
Processor-Independent Formulas for SCOSF for NAUX1 and NAUX2:

**NAUX1 Formulas**

**For 32-bit integer arguments:**
- If \( n \leq 16384 \), use \( naux1 = 40000 \).
- If \( n > 16384 \), use \( naux1 = 20000 + .30n \).

**For 64-bit integer arguments:**
- If \( n \leq 16384 \), use \( naux1 = 50000 \).
- 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:

\[
(n/4+257)(\min(128, m))
\]

Processor-Independent Formulas for DCOSF for NAUX1 and NAUX2:

**NAUX1 Formulas**

**For 32-bit integer arguments:**
- If \( n \leq 16384 \), use \( naux1 = 35000 \).
- If \( n > 16384 \), use \( naux1 = 20000 + .60n \).

**For 64-bit integer arguments:**
- If \( n \leq 16384 \), use \( naux1 = 45000 \).
- 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:

\[
(n/2+257)(\min(128, m))
\]

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:

\[
y_{k,i} = scale \left( 0.5x_{0,i} + 0.5(-1)^{\frac{k}{2}}x_{n/2,i} + \sum_{j=1}^{n/2-1} x_{j,i} \cos\left( jk \left( \frac{2\pi}{n} \right) \right) \right)
\]

for:
where:

\( x_{ji} \) are elements of the sequences in array \( X \), where each sequence contains the \( n/2+1 \) real nonredundant data \( x_{jiv} \), \( j = 0, 1, \ldots, n/2 \).

\( y_{ki} \) are elements of the sequences in array \( Y \), where each sequence contains the \( n/2+1 \) real nonredundant data \( y_{kiv} \), \( k = 0, 1, \ldots, n/2 \).

\( \text{scale} \) is a scalar value.

You can obtain the inverse cosine transform by specifying \( \text{scale} = 4.0/n \). Thus, if an \( X \) input is used with \( \text{scale} = 1.0 \), and its output is used as input on a subsequent call with \( \text{scale} = 4.0/n \), the original \( X \) is obtained. See references [“1” on page 1253], [“5” on page 1253], [“27” on page 1254], and [“28” on page 1254].

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 \( \text{aux1} \) working storage.
2. With \( \text{init} = 0 \), the subroutine checks that the initialization arguments in the \( \text{aux1} \) 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, \( \text{naux2} = 0 \), and unable to allocate work area.

**Computational Errors**
- None

**Input-Argument Errors**
1. \( n > 37748736 \)
2. \( \text{inc1x or inc1y} \leq 0 \)
3. \( \text{inc2x or inc2y} \leq 0 \)
4. \( m \leq 0 \)
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. \( \text{naux1} \) 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 \( \text{naux2} \neq 0 \), 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 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:**

```
INIT  X INC1X INC2X Y INC1Y INC2Y  N   M   SCALE  AUX1  NAUX1  AUX2  NAUX2
|    |   |     |   |   |     |    |   |    |      |      |     |      |
CALL SCOSF(INIT, X , 1  , 18 , Y , 1  , 18 , 32 , 4 , SCALE, AUX1 , 414 , AUX2 ,  0)
```

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  -0.1951  -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   0.0000  0.0000  
0.0000   0.0000   0.0000  0.0000  
0.0000   0.0000   0.0000  0.0000  
0.0000   0.0000   0.0000  0.0000  
0.0000   0.0000   0.0000  0.0000  
0.0000   0.0000   0.0000  0.0000  
0.0000   0.0000   0.0000  0.0000  
0.0000   0.0000   0.0000  0.0000  
0.0000   0.0000   0.0000  0.0000  
0.0000   0.0000   0.0000  0.0000  
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:

```
 INIT    X INC1X INC2X Y INC1Y INC2Y  N   M   SCALE  AUX1  NAUX1  AUX2  NAUX2
                      |   |   |     |   |   |     |    |   |     |     |      |     |      |
CALL SCOSF(INIT, X, 1, 18, Y, 1, 18, 32, 4, SCALE, AUX1, 414, AUX2, 0)
```

INIT = 1 (for initialization)  
INIT = 0 (for computation)  
SCALE = 4.0/32  
X = (same sequences as in output Y in Example 1)

Output:  
Y = (same sequences as in output X in Example 1)

**Example 3**

This example shows another computation using the same arguments initialized in Example 1 and using different input sequence data. The data for this example has frequencies 1, 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:

```
 INIT    X INC1X INC2X Y INC1Y INC2Y  N   M   SCALE  AUX1  NAUX1  AUX2  NAUX2
                      |   |   |     |   |   |     |    |   |     |     |      |     |      |
CALL SCOSF(0, X, 1, 18, Y, 1, 18, 32, 4, SCALE, AUX1, 414, AUX2, 0)
```

SCALE = 1.0

X contains the following four sequences:

<table>
<thead>
<tr>
<th>1.0000</th>
<th>1.0000</th>
<th>1.0000</th>
<th>1.0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.9239</td>
<td>-0.9808</td>
<td>-1.0000</td>
<td>-0.9808</td>
</tr>
<tr>
<td>0.7071</td>
<td>0.9239</td>
<td>1.0000</td>
<td>0.9239</td>
</tr>
<tr>
<td>-0.3827</td>
<td>-0.8315</td>
<td>-1.0000</td>
<td>-0.8315</td>
</tr>
<tr>
<td>0.3827</td>
<td>-0.5556</td>
<td>-1.0000</td>
<td>-0.5556</td>
</tr>
<tr>
<td>-0.7071</td>
<td>0.3827</td>
<td>1.0000</td>
<td>0.3827</td>
</tr>
<tr>
<td>0.9239</td>
<td>-0.1951</td>
<td>-1.0000</td>
<td>-0.1951</td>
</tr>
<tr>
<td>-1.0000</td>
<td>0.0000</td>
<td>1.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>0.9239</td>
<td>0.1951</td>
<td>1.0000</td>
<td>0.1951</td>
</tr>
<tr>
<td>-0.7071</td>
<td>-0.3827</td>
<td>1.0000</td>
<td>-0.3827</td>
</tr>
<tr>
<td>0.3827</td>
<td>-0.5556</td>
<td>-1.0000</td>
<td>0.5556</td>
</tr>
<tr>
<td>0.0000</td>
<td>-0.7071</td>
<td>1.0000</td>
<td>-0.7071</td>
</tr>
<tr>
<td>-0.3827</td>
<td>0.8315</td>
<td>-1.0000</td>
<td>0.8315</td>
</tr>
<tr>
<td>0.7071</td>
<td>-0.9239</td>
<td>1.0000</td>
<td>-0.9239</td>
</tr>
<tr>
<td>-0.9239</td>
<td>0.9808</td>
<td>-1.0000</td>
<td>0.9808</td>
</tr>
<tr>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>-1.0000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Output:

Y contains the following four sequences:

<table>
<thead>
<tr>
<th>0.0000</th>
<th>0.0000</th>
<th>0.0000</th>
<th>0.0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>8.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
</tbody>
</table>
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 226: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X, Y, \text{scale}$</td>
<td></td>
</tr>
<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 34.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SSINF</td>
</tr>
<tr>
<td>C and C++</td>
<td>ssinf</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 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 $1+(n / 2-1)inc1x+(m-1)inc2x$, containing numbers of the data type indicated in Table 226 on page 1015. 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; $inc1x > 0$. 
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 956. 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 56.

$m$ is the number of sequences to be transformed. Specified as: an integer; $m > 0$.

$scale$ is the scaling constant $scale$. See “Function” on page 1018 for its usage. Specified as: a number of the data type indicated in Table 226 on page 1015, 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 49.

$aux2$ 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, \( naux2 \geq (\text{minimum value required for successful processing}) \). 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 49.

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 \( 1+(n/2-1)inc1y+(m-1)inc2y \), containing numbers of the data type indicated in Table 226 on page 1015.

\( aux1 \) 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. For optimal performance, the preferred value for \( \text{inc}1x \) and \( \text{inc}1y \) is 1. This implies that the sequences are stored with stride 1. In addition, \( \text{inc}2x \) and \( \text{inc}2y \) 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, \( \text{inc}2x \) (or \( \text{inc}2y \)) = 1 and \( \text{inc}1x \) (or \( \text{inc}1y \)) 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 \( \text{inc}1x \) and \( \text{inc}1y \) for _SINF, you should use “STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)” on page 1195. 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 \( \text{inc}1x \) and \( \text{inc}1y \) must be equal, and \( \text{inc}2x \) and \( \text{inc}2y \) must be equal. In this case, output overwrites input. If \( m = 1 \), the \( \text{inc}2x \) and \( \text{inc}2y \) 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 “Vector concepts” on page 71.

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:

\[ (n/4+257)(\min(128, m)) \]

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:

\[ (n/2+257)(\min(128, m)) \]

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:

\[
y_{ki} = \text{scale} \sum_{j=0}^{n/2-1} x_{ji} \sin\left(\frac{jk(2\pi/n)}{n}\right)\]

for:

\( k = 0, 1, \ldots, n/2-1 \)
\( i = 1, 2, \ldots, m \)

where:

\( x_{0i} = 0.0 \)

\( x_{ji} \) are elements of the sequences in array \( X \), where each sequence contains the \( n/2 \) real nonredundant data \( x_{ji}, j = 0, 1, \ldots, n/2-1 \).
$y_{ki}$ are elements of the sequences in array $Y$, where each sequence contains the $n/2$ real nonredundant data $y_{ki}$, $k = 0, 1, \ldots, n/2-1$.

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 1253], [“5” on page 1253], [“27” on page 1254], and [“28” on page 1254].

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.

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, $\text{naux2} = 0$, and unable to allocate work area.

Computational Errors

None

Input-Argument Errors

1. $n > 37748736$
2. $\text{inc1x} \text{ or } \text{inc1y} \leq 0$
3. $\text{inc2x} \text{ or } \text{inc2y} \leq 0$
4. $m \leq 0$
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. $\text{naux1}$ 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 $\text{naux2}=0$, 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 an input array $X$ with a set of $m$ sine sequences of length $n/2$, $\sin(jk(2\pi/n))$, $j = 0, 1, \ldots, n/2-1$, 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 $\text{INIT} \neq 0$. Then use the same calling sequence with $\text{INIT} = 0$ to do the calculation.

Note: Because $\text{NAUX2}=0$, this subroutine dynamically allocates the AUX2 working storage.
Call Statement and Input:

```
INIT  X INC1X INC2X Y INC1Y INC2Y  N   M   SCALE  AUX1  NAUX1  AUX2  NAUX2
|    |   |     |   |   |     |    |   |     |     |      |     |      |
CALL SSINF(INIT, X , 1  , 18 , Y , 1  , 18 , 32 , 3 , SCALE, AUX1 , 414 , AUX2 ,  0)
```

INIT = 1 (for initialization)
INIT = 0 (for computation)
SCALE = 1.0

X contains the following three sequences:

```
0.0000  0.0000  0.0000
0.1951  0.3827  0.5556
0.3827  0.7071  0.9239
0.5556  0.9239  0.9808
0.7071  1.0000  0.7071
0.8315  0.9239  0.1951
0.9239  0.7071  -0.3827
0.9000  -0.3827  -0.8315
1.0000  0.0000  -1.0000
0.9000  -0.3827  -0.8315
0.8315  -0.9239  0.1951
0.7071  -1.0000  0.7071
0.5556  -0.9239  0.9808
0.3827  -0.7071  0.9239
0.1951  -0.3827  0.5556
.       .       .
```

Output:

Y contains the following three sequences:

```
0.0000  0.0000  0.0000
8.0000  0.0000  0.0000
0.0000  8.0000  0.0000
0.0000  0.0000  8.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
0.0000  0.0000  0.0000
.       .       .
```

Example 2

This example shows an input array X with a set of three input spike sequences equal to the output of Example 1. This shows how you can compute the inverse of the transform in Example 1 by using $scale = 4.0/n$, giving as output the three sequences listed in the input for Example 1. First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

Note: Because NAUX2= 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
INIT  X INC1X INC2X Y INC1Y INC2Y  N   M   SCALE  AUX1  NAUX1  AUX2  NAUX2
|    |   |     |   |   |     |    |   |     |     |      |     |      |
CALL SSINF(INIT, X , 1  , 18 , Y , 1  , 18 , 32 , 3 , SCALE, AUX1 , 414 , AUX2 ,  0)
```

INIT = 1 (for initialization)
INIT = 0 (for computation)
SCALE = 4.0/32
X = (same sequences as in output Y in Example 1)

Output:

Y = (same sequences as in output X in Example 1)

Example 3

This example shows another computation using the same arguments initialized in Example 1 and using different input sequence data. The data for this example has frequencies \( k = 14, 15, 17 \). Because only the sequence data has changed, initialization does not have to be done again.

Note: Because NAUX2 = 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

| INIT X INC1X INC2X Y INC1Y INC2Y N   M   SCALE  AUX1  NAUX1  AUX2  NAUX2 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| CALL SSINF( 0 , X , 1 , 18 , Y , 1 , 18 , 32 , 3 , SCALE, AUX1 , 414 , AUX2 , 0 ) |

SCALE = 1.0

X contains the following three sequences:

| 0.0000 | 0.0000 | 0.0000 |
| 0.3827 | 0.1951 | -0.1951 |
| -0.7071 | -0.3827 | 0.3827 |
| 0.9239 | 0.5556 | -0.5556 |
| -1.0000 | -0.7071 | 0.7071 |
| 0.9239 | 0.8315 | -0.8315 |
| -0.7071 | -0.9239 | 0.9239 |
| 0.3827 | 0.9808 | -0.9808 |
| 0.8573 | -1.0000 | 1.0000 |
| -0.3827 | 0.9808 | -0.9808 |
| 0.7071 | -0.9239 | 0.9239 |
| -0.9239 | 0.8315 | -0.8315 |
| 1.0000 | -0.7071 | 0.7071 |
| -0.9239 | 0.5556 | -0.5556 |
| 0.7071 | -0.3827 | 0.3827 |
| -0.3827 | 0.1951 | -0.1951 |
| .        | .        | .        |

Output:

Y contains the following three sequences:

| 0.0000 | 0.0000 | 0.0000 |
| 0.0000 | 0.0000 | 0.0000 |
| 0.0000 | 0.0000 | 0.0000 |
| 0.0000 | 0.0000 | 0.0000 |
| 0.0000 | 0.0000 | 0.0000 |
| 0.0000 | 0.0000 | 0.0000 |
| 0.0000 | 0.0000 | 0.0000 |
| 0.0000 | 0.0000 | 0.0000 |
| 0.0000 | 0.0000 | 0.0000 |
| 0.0000 | 0.0000 | 0.0000 |
| 0.0000 | 0.0000 | 0.0000 |
| 0.0000 | 0.0000 | 0.0000 |
| 0.0000 | 0.0000 | 0.0000 |
| 0.0000 | 0.0000 | 0.0000 |
| 8.0000 | 0.0000 | -8.0000 |
| 0.0000 | 0.0000 | 0.0000 |
| 0.0000 | 0.0000 | 0.0000 |
| .        | .        | .        |

SCFT2 and DCFT2 (Complex Fourier Transform in Two Dimensions)

Purpose

These subroutines compute the two-dimensional discrete Fourier transform of complex data.
### Table 227: Data Types

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y scale</td>
<td>SCFT2</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>SCFT2</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>DCFT2</td>
</tr>
<tr>
<td>Short-precision real</td>
<td></td>
</tr>
<tr>
<td>Long-precision real</td>
<td></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 34.

### Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SCFT2</td>
</tr>
<tr>
<td>C and C++</td>
<td>scft2</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 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 227 on page 1022.
  - 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 1024 for more details.

- **inc1x**
  - is the stride between the elements in array X for the first dimension.
    - If the array is stored in the normal form, inc1x = 1.
    - If the array is stored in the transposed form, inc1x is the leading dimension of the array and inc1x ≥ n2.
    - Specified as: an integer; inc1x > 0. If inc2x = 1, then inc1x ≥ n2.
inc2x is the stride between the elements in array X for the second dimension.
If the array is stored in the transposed form, inc2x = 1.
If the array is stored in the normal form, inc2x is the leading dimension of the array and inc2x ≥ n1.
Specified as: an integer; inc2x > 0. If inc1x = 1, then inc2x ≥ n1.

y
See On Return.

inc1y is the stride between the elements in array Y for the first dimension.
If the array is stored in the normal form, inc1y = 1.
If the array is stored in the transposed form, inc1y is the leading dimension of the array and inc1y ≥ n2.
Specified as: an integer; inc1y > 0. If inc2y = 1, then inc1y ≥ n2.

inc2y is the stride between the elements in array Y for the second dimension.
If the array is stored in the transposed form, inc2y = 1.
If the array is stored in the normal form, inc2y is the leading dimension of the array and inc2y ≥ n1.
Specified as: an integer; inc2y > 0. If inc1y = 1, then inc2y ≥ n1.

n1 is the length of the first dimension of the two-dimensional data in the array to be transformed.
Specified as: an integer; n1 ≤ 37748736 and must be one of the values listed in “Acceptable Lengths for the Transforms” on page 956. 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 56.

n2 is the length of the second dimension of the two-dimensional data in the array to be transformed.
Specified as: an integer; n2 ≤ 37748736 and must be one of the values listed in “Acceptable Lengths for the Transforms” on page 956. 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 56.

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 1026 for its usage.
Specified as: a number of the data type indicated in Table 227 on page 1022, 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.
**aux1**

is the number of doublewords in the working storage specified in aux1.

Specified as: an integer; \( \text{aux1} \geq (\text{minimum value required for successful processing}) \). 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 49.

**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 \( \text{aux2} \) 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, 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, \( \text{naux2} \geq (\text{minimum value required for successful processing}) \). 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 49.

**On Return**

\( \text{y} \)

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 \( \text{y} \), containing the elements resulting from the two-dimensional discrete Fourier transform of the data in \( \text{X} \). Each element \( y_{k1,k2} \), using zero-based indexing, is stored in \( Y(k1(inc1y) + k2(inc2y)) \) for \( k1 = 0, 1, ..., n1-1 \) and \( k2 = 0, 1, ..., n2-1 \).

Returned as: an array of (at least) length \( 1+(n1-1)inc1y+(n2-1)inc2y \), containing numbers of the data type indicated in Table 227 on page 1022.

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 1024 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. \( \text{aux1} \) 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 init ≠ 0 and init = 0.

3. If you specify the same array for X and Y, then inc1x must equal inc1y, and inc2x must equal inc2y. In this case, output overwrites input. If you specify different arrays X and Y, they must have no common elements; otherwise, results are unpredictable. See “Vector concepts” on page 71.

4. By appropriately specifying the inc arguments, this subroutine allows you to specify that it should use one of two forms of its arrays, the normal untransposed form or the transposed form. As a result, you do not have to move any data. Instead, the subroutine performs the adjustments for you. Also, either the input array or the output array can be in transposed form. The FFT computation is symmetrical with respect to n1 and n2. They can be interchanged without the loss of generality. If they are interchanged, an array that is stored in the normal form appears as an array stored in the transposed form and vice 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 inc arguments for each array can be arbitrary, in most cases, one of the inc arguments is 1 for each array. If inc1 = 1, the array is stored in normal form; that is, the first dimension of the array is along the columns. Conversely, if inc2 = 1, the array is stored in the transposed form; that is, the first dimension of the array is along the rows. The rows of the arrays are accessed with a stride that equals the leading dimension of the array. To minimize cache interference in accessing a row, an optimal value should be used for the leading dimension of the array. You should use “STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)” on page 1195 to determine this optimal value. Example 4 in the STRIDE subroutine description explains how it is used to find either inc1 or inc2.

Formulas
Processor-Independent Formulas for SCFT2 for NAUX1 and NAUX2:

The required values of naux1 and naux2 depend on n1 and n2.

AUX1 Formulas
For 32-bit integer arguments:
If max(n1, n2) ≤ 8192, use naux1 = 40000.
If max(n1, n2) > 8192, use naux1 = 40000+1.14(n1+n2).

For 64-bit integer arguments:
If max(n1, n2) ≤ 8192, use naux1 = 60000.
If max(n1, n2) > 8192, use naux1 = 60000+1.14(n1+n2).

NAUX2 Formulas
If max(n1, n2) < 252, use naux2 = 20000.
If max(n1, n2) ≥ 252, use naux2 = 20000+(r+256)(s+1.14), where r = max(n1, n2) and s = min(64, n1, n2).

Processor-Independent Formulas for DCFT2 for NAUX1 and NAUX2:

The required values of naux1 and naux2 depend on n1 and n2.

NAUX1 Formulas
For 32-bit integer arguments:
If max(n1, n2) ≤ 2048, use naux1 = 40000.
If max(n1, n2) > 2048, use naux1 = 40000+2.28(n1+n2).

For 64-bit integer arguments:
If max(n1, n2) ≤ 2048, use naux1 = 60000.
If max(n1, n2) > 2048, use naux1 = 60000+2.28(n1+n2).
**NAUX2 Formulas**

If \( \max(n1, n2) < 252 \), use \( \text{naux2} = 20000 \).

If \( \max(n1, n2) \geq 252 \), use \( \text{naux2} = 20000 + (2r+256)(s+2.28) \), where \( r = \max(n1, n2) \) and \( s = \min(64, n1, n2) \).

**Function**

The two-dimensional discrete Fourier transform of complex data in array \( X \), with results going into array \( Y \), is expressed as follows:

\[
y_{k1,k2} = \text{scale} \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} x_{j1,j2} W_{n1}^{(i\text{sign})j1k1} W_{n2}^{(i\text{sign})j2k2}
\]

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 \( Y \).
\(i\text{sign}\) is + or - (determined by argument \( i\text{sign} \)).
\(\text{scale}\) is a scalar value.

For \( \text{scale} = 1.0 \) and \( i\text{sign} \) 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 \( i\text{sign} \) being negative. See references [“1” on page 1253], [“5” on page 1253], and [“28” on page 1254].

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 \( \text{aux1} \) working storage.
2. With \( \text{init} = 0 \), the subroutine checks that the initialization arguments in the \( \text{aux1} \) 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, \( \text{naux2} = 0 \), and unable to allocate work area.

**Computational Errors**

None

**Input-Argument Errors**

1. \( n1 > 37748736 \)
2. \( n2 > 37748736 \)
3. \( \text{inc1x}|\text{inc2x}|\text{inc1y}|\text{inc2y} \leq 0 \)
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 \( naux2 \neq 0 \), 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 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:

```fortran
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 \( INIT \neq 0 \). 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
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:

```fortran
COMPLEX*16 X(6,8), Y(8,6)
REAL*8 AUX1(20000), AUX2(1)
```

First, initialize \( AUX1 \) using the calling sequence shown below with \( INIT \neq 0 \). 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
INIT  X INC1X INC2X Y INC1Y INC2Y N1  N2 ISIGN SCALE  AUX1  NAUX1  AUX2  NAUX2
|    |   |     |   |   |     |   |   |    |     |     |       |      |       |
CALL DCFT2(INIT, X, 1, 6, Y, 8, 1, 6, 8, -1, SCALE, AUX1, 20000, AUX2, 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.
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>
<thead>
<tr>
<th>Table 228: Data Types.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Types</strong></td>
</tr>
<tr>
<td>X, scale</td>
</tr>
<tr>
<td>---</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 34.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SRCFT2 (init, x, inc2x, y, inc2y, n1, n2, isign, scale, aux1, auxx1, aux2, auxx2, aux3, auxx3)</td>
</tr>
<tr>
<td></td>
<td>CALL DRCFT2 (init, x, inc2x, y, inc2y, n1, n2, isign, scale, aux1, auxx1, aux2, auxx2)</td>
</tr>
<tr>
<td>C and C++</td>
<td>srcft2 (init, x, inc2x, y, inc2y, n1, n2, isign, scale, aux1, auxx1, aux2, auxx2, auxx3, auxx3);</td>
</tr>
<tr>
<td></td>
<td>drctf2 (init, x, inc2x, y, inc2y, n1, n2, isign, scale, aux1, auxx1, aux2, auxx2, auxx2);</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.
is the array X, containing \( n_1 \) rows and \( n_2 \) columns of data to be transformed. The data in each column is stored with stride 1. Specified as: an \( \text{inc2x} \) by (at least) \( n_2 \) array, containing numbers of the data type indicated in Table 228 on page 1028. See “Notes” on page 1030 for more details.

**inc2x**

is the leading dimension (stride between columns) of array X. Specified as: an integer; \( \text{inc2x} \geq n_1 \).

\( y \)

See On Return.

**inc2y**

is the leading dimension (stride between columns) of array Y. Specified as: an integer; \( \text{inc2y} \geq ((n_1)/2)+1 \).

\( n_1 \)

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 \((n_1)/2+1\).

Specified as: an integer; \( n_1 \leq 37748736 \) and must be one of the values listed in “Acceptable Lengths for the Transforms” on page 956. 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 56.

\( n_2 \)

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; \( n_2 \leq 37748736 \) and must be one of the values listed in “Acceptable Lengths for the Transforms” on page 956. 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 56.

\( \text{isign} \)

controls the direction of the transform, determining the sign \( \text{Isign} \) of the exponents of \( W_{n_1} \) and \( W_{n_2} \), 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 \( \text{isign} < 0 \).

\( \text{scale} \)

is the scaling constant \( \text{scale} \). See “Function” on page 1031 for its usage.

Specified as: a number of the data type indicated in Table 228 on page 1028, 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 49.

\( \text{aux2} \)

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, \( SRCFT2 \) and \( DRCFT2 \) dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, \( naux2 \geq \) (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 49.

\( aux3 \)

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 \( Y \), containing the results of the complex discrete Fourier transform of \( X \). The output consists of \( n2 \) columns of data. The data in each column is stored with stride 1. Due to complex conjugate symmetry, the output consists of only the first \( ((n1)/2)+1 \) rows of the array—that is, \( y_{k1,k2} \), where \( k1 = 0, 1, \ldots, (n1)/2 \) and \( k2 = 0, 1, \ldots, n2-1 \).

Returned as: an \( inc2y \) by (at least) \( n2 \) array, containing numbers of the data type indicated in Table 228 on page 1028.

\( aux1 \)

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 \textbf{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. If you specify the same array for \( X \) and \( Y \), then \( inc2x \) must equal \((2)(inc2y)\). In this case, output overwrites input. If you specify different arrays \( X \) and \( Y \), they must have no common elements; otherwise, results are unpredictable. See “Vector concepts” on page 71.
4. For selecting optimal strides (or leading dimensions \( \text{inc2x} \) and \( \text{inc2y} \)) for your input and output arrays, you should use “STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)” on page 1195. 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 \( n2 \).

\textbf{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 \( \text{naux1} = 60000+0.82n1+1.14n2 \).

\textbf{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 \( n2 \).

\textbf{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 \( n = \max(n1/2, n2) \).

\textbf{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:
\[
y_{k1,k2} = scale \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} x_{j1,j2} W_{n1}^{(I\text{sign})} j1 k1 W_{n2}^{(I\text{sign})} j2 k2
\]

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:

\[x_{j1,j2}\] are elements of array \(X\).
\[y_{k1,k2}\] are elements of array \(Y\).
\(I\text{sign}\) is + or - (determined by argument \(I\text{sign}\)).
\(scale\) is a scalar value.

The output in array \(Y\) is complex. For \(scale = 1.0\) and \(I\text{sign}\) 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 \(I\text{sign}\) being negative. See references [“1” on page 1253], [“5” on page 1253], [“27” on page 1254], and [“28” on page 1254].

Two invocations of this subroutine are necessary:

1. With \(I\text{nt} \neq 0\), the subroutine tests and initializes arguments of the program, setting up the \(aux1\) working storage.
2. With \(I\text{nt} = 0\), 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. \(I\text{sign} = 0\)
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 \( \text{naux2} \neq 0 \), 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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPLEX*8</td>
<td>( Y(0:6,0:7) )</td>
</tr>
<tr>
<td>REAL*4</td>
<td>( X(0:11,0:7) )</td>
</tr>
<tr>
<td>REAL*8</td>
<td>( \text{AUX1}(1000), \text{AUX2}(1), \text{AUX3}(1) )</td>
</tr>
</tbody>
</table>

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:**

<table>
<thead>
<tr>
<th>NAUX3</th>
<th>INIT</th>
<th>X</th>
<th>INC2X</th>
<th>Y</th>
<th>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>
<tbody>
<tr>
<td>CALL</td>
<td>SRCFT2(INIT, X, 12, Y, 7, 12, 8, 1, SCALE, AUX1, 1000, AUX2, 0, AUX3, 0)</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>
<td></td>
</tr>
</tbody>
</table>

\( \text{INIT} = 1 \) (for initialization)  
\( \text{INIT} = 0 \) (for computation)  
\( \text{SCALE} = 1.0 \)

\( X \) 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 \( \text{AUX1} \) in Example 1.

**Note:** Because \( \text{NAUX2} = 0 \), this subroutine dynamically allocates the \( \text{AUX2} \) working storage.

**Call Statement and Input:**

<table>
<thead>
<tr>
<th>INIT</th>
<th>X</th>
<th>INC2X</th>
<th>Y</th>
<th>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>
<tbody>
<tr>
<td>CALL</td>
<td>SRCFT2(0, X, 12, Y, 7, 12, 8, 1, SCALE, AUX1, 1000, AUX2, 0, AUX3, 0)</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>

\( \text{SCALE} = 1.0 \)

\( X \) 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 \( \text{isign} = -1 \) and \( \text{scale} = 1/((\text{N1})(\text{N2})) \). The arrays are declared as follows:

<table>
<thead>
<tr>
<th>Type</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPLEX*16</td>
<td>( Y(0:8,0:7) )</td>
</tr>
<tr>
<td>REAL*8</td>
<td>( X(0:19,0:7) )</td>
</tr>
<tr>
<td>REAL*8</td>
<td>( \text{AUX1}(1000), \text{AUX2}(1), \text{AUX3}(1) )</td>
</tr>
</tbody>
</table>
Arrays X and Y are made equivalent by the following statement, making them occupy the same storage.

```
EQUIVALENCE (X,Y)
```

This requires $inc_2x \geq 2/inc_2y \). 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 X INC2X Y INC2Y N1 N2 ISIGN SCALE AUX1 NAUX1 AUX2 NAUX2 AUX3
```

<table>
<thead>
<tr>
<th>NAUX3</th>
<th>INIT X INC2X Y INC2Y N1 N2 ISIGN SCALE AUX1 NAUX1 AUX2 NAUX2 AUX3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CALL DRCFT2(INIT, X, Y, N1, N2, ISIGN, SCALE, AUX1, NAUX1, AUX2, NAUX2, AUX3, NAUX3)</td>
</tr>
</tbody>
</table>

INIT = 1 (for initialization)
INIT = 0 (for computation)
SCALE = 1.0/128.0

```
X =
```

```
| 2.0 2.0 -2.0 -2.0 2.0 2.0 -2.0 -2.0 |
| 2.0 -2.0 -2.0 2.0 -2.0 2.0 -2.0 2.0 |
| -2.0 2.0 2.0 -2.0 -2.0 2.0 2.0 -2.0 |
| 2.0 2.0 -2.0 2.0 2.0 2.0 -2.0 -2.0 |
| -2.0 -2.0 2.0 2.0 -2.0 2.0 2.0 2.0 |
| 2.0 -2.0 2.0 2.0 -2.0 2.0 2.0 2.0 |
| -2.0 -2.0 2.0 -2.0 2.0 2.0 2.0 2.0 |
| -2.0 2.0 2.0 2.0 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 229: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y, scale</td>
</tr>
<tr>
<td>Short-precision complex</td>
<td>Short-precision real</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>Long-precision real</td>
</tr>
</tbody>
</table>
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 34.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
</table>
| **Fortran** | CALL SCRFT2 (init, x, inc2x, y, inc2y, n1, n2, isign, scale, aux1, aux2, aux3, aux1, aux3, naux1, naux3)  
         CALL DCRFT2 (init, x, inc2x, y, inc2y, n1, n2, isign, scale, aux1, aux2, aux2, naux2) |
| **C and C++** | scrft2 (init, x, inc2x, y, inc2y, n1, n2, isign, scale, aux1, aux2, aux3, naux3);  
dcrft2 (init, x, inc2x, y, inc2y, n1, n2, isign, scale, aux1, aux2, aux2, naux2); |

**On Entry**

- **init** is a flag, where:
  - If init ≠ 0, trigonometric functions and other parameters, depending on arguments other than x, are computed and saved in aux1. The contents of x and y are not used or changed.
  - If init = 0, the discrete Fourier transform of the given array is computed. The only arguments that may change after initialization are x, y, and aux2. All scalar arguments must be the same as when the subroutine was called for initialization with init ≠ 0.

  Specified as: 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(i, j) = 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 229 on page 1034.

- **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 ≥ 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 ≤ 37748736 and must be one of the values listed in “Acceptable Lengths for the Transforms” on page 956. 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 56.
\[ n2 \]
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 956. 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 56.

\[ isign \]
controls the direction of the transform, determining the sign \( Isign \) of the exponents of \( W_n^1 \) and \( W_n^2 \), 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 1038 for its usage.

Specified as: a number of the data type indicated in Table 229 on page 1034, 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 49.

\[ aux2 \]
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, \( SCRFT2 \) and \( DCRFT2 \) dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, \( naux2 \geq (minimum \ value \ required \ for \ successful \ processing) \). 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 49.

\[ aux3 \]
is the 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 ≠ 0, this argument is not used, and its contents remain unchanged.

If init = 0, this is the array Y, containing n1 rows and n2 columns of results of the real discrete Fourier transform of X. The data in each column of Y is stored with stride 1.

Returned as: an inc2y by (at least) n2 array, containing numbers of the data type indicated in Table 229 on page 1034. See “Notes” on page 1037 for more details.

**aux1**

is the working storage for this subroutine, where:

If init ≠ 0, it contains information ready to be passed in a subsequent invocation of this subroutine.

If init = 0, its contents are unchanged.

Returned as: the contents are not relevant.

**Notes**

1. **aux1** should not be used by the calling program between program calls to this subroutine with init ≠ 0 and init = 0. However, it can be reused after intervening calls to this subroutine with different arguments.

2. 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. If you specify the same array for X and Y, then (2)(inc2x) must equal inc2y. In this case, output overwrites input. If you specify different arrays X and Y, they must have no common elements; otherwise, results are unpredictable. See “Vector concepts” on page 71.

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 1195. 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) ≤ 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) ≤ 8192, use naux1 = 65000. If max(n1/2, n2) > 8192, use naux1 = 60000+0.82n1+1.14n2.

**NAUX2 Formulas**

If n1 ≤ 16384 and n2 < 252, use naux2 = 20000.

If n1 > 16384 and n2 < 252, use naux2 = 20000+0.57n1.
If \( n_2 \geq 252 \), add the following to the above storage requirements:
\[
(n_2+256)(1.14+s)
\]
where \( s = \min(64, 1+n_1/2) \).

Processor-Independent Formulas for DCRFT2 for NAUX1 and NAUX2:

The required values of \( naux1 \) and \( naux2 \) depend on \( n_1 \) and \( n_2 \).

**NAUX1 Formulas**

For 32-bit integer arguments:

- If \( n \leq 2048 \), use \( naux1 = 42000 \).
- If \( n > 2048 \), use \( naux1 = 40000+1.64n_1+2.28n_2 \), where \( n = \max(n_1/2, n_2) \).

For 64-bit integer arguments:

- If \( n \leq 2048 \), use \( naux1 = 62000 \).
- If \( n > 2048 \), use \( naux1 = 60000+1.64n_1+2.28n_2 \), where \( n = \max(n_1/2, n_2) \).

**NAUX2 Formulas**

- If \( n_1 \leq 4096 \) and \( n_2 < 252 \), use \( naux2 = 20000 \).
- If \( n_1 > 4096 \) and \( n_2 < 252 \), use \( naux2 = 20000+1.14n_1 \).
- If \( n_2 \geq 252 \), add the following to the above storage requirements:
\[
((2)n_2+256)(2.28+s)
\]
where \( s = \min(64, 1+n_1/2) \).

**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:

\[
W_{n1} = e^{-2\pi(s-1)/n1}
\]
\[
W_{n2} = e^{-2\pi(s-1)/n2}
\]

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.
Because of the complex conjugate symmetry, the output in array \( Y \) is real. For \( scale = 1.0 \) and \( isign \) being positive, you obtain the discrete Fourier transform, a function of frequency. The inverse Fourier transform is obtained with \( scale = 1.0/((n1)(n2)) \) and \( isign \) being negative. See references [“1” on page 1253], [“5” on page 1253], and [“28” on page 1254].

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. \( inc2x < (n1)/2+1 \)
4. \( inc2y < n1+2 \)
5. \( scale = 0.0 \)
6. \( isign = 0 \)
7. The subroutine has not been initialized with the present arguments.
8. The length of one of the transforms in \( n1 \) or \( n2 \) is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
9. \( naux1 \) is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
10. Error 2015 is recoverable or \( naux2 \neq 0 \), 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
REAL*4     Y(0:13,0:7)
COMPLEX*8  X(0:6,0:7)
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
<table>
<thead>
<tr>
<th>NAUX3</th>
<th>INIT</th>
<th>X</th>
<th>INC2X</th>
<th>Y</th>
<th>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>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>SCRFT2(INIT, X, 7, Y, 14, 12, 8, -1, SCALE, AUX1, 1000, AUX2, 0, AUX3, 0)</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>
```
INIT = 1 (for initialization)
INIT = 0 (for computation)
SCALE = 1.0/96.0

X is an array with 7 rows and 8 columns with (1.0, 0.0) in all locations.

Output:

\[
Y = \begin{bmatrix}
1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots
\end{bmatrix}
\]

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:

\[
\text{SCALE} = 1.0/96.0
\]

Y is an array with 7 rows and 8 columns having (96.0, 0.0) in location \(X(0,0)\) and (0.0, 0.0) in all others.

Output:

\[
Y = \begin{bmatrix}
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots
\end{bmatrix}
\]

Example 3

This example shows the same array being used for input and output. The arrays are declared as follows:

\[
\text{REAL*8} \quad Y(0:17,0:7) \\
\text{COMPLEX*16} \quad X(0:8,0:7) \\
\text{REAL*8} \quad AUX1(1000), AUX2(1), AUX3(1)
\]
Arrays X and Y are made equivalent by the following statement, making them occupy the same storage.

\[
\text{EQUIVALENCE (X,Y)}
\]

This requires \( inc2y = 2 \cdot (inc2x) \). 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>NAUX3</th>
<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>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>DCRFT2(INIT, X , 9 , Y , 18 , 16 , 8 , 1 , SCALE , AUX1 , 1000 , AUX2 , 0 , AUX3 , 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 is an array with 9 rows and 8 columns having \((1.0, 1.0)\) in location \( X(4, 2) \) and \((0.0, 0.0)\) in all others.

**Output:**

\[
Y = \begin{bmatrix}
2.0 & 2.0 & -2.0 & -2.0 & 2.0 & 2.0 & -2.0 & -2.0 \\
2.0 & -2.0 & -2.0 & 2.0 & 2.0 & -2.0 & -2.0 & 2.0 \\
-2.0 & -2.0 & 2.0 & 2.0 & -2.0 & -2.0 & 2.0 & 2.0 \\
2.0 & 2.0 & -2.0 & -2.0 & 2.0 & 2.0 & -2.0 & -2.0 \\
2.0 & -2.0 & -2.0 & 2.0 & 2.0 & -2.0 & -2.0 & 2.0 \\
-2.0 & -2.0 & 2.0 & 2.0 & -2.0 & -2.0 & 2.0 & 2.0 \\
-2.0 & 2.0 & 2.0 & -2.0 & -2.0 & 2.0 & 2.0 & -2.0 \\
2.0 & 2.0 & -2.0 & -2.0 & 2.0 & 2.0 & -2.0 & -2.0 \\
2.0 & -2.0 & -2.0 & 2.0 & 2.0 & -2.0 & -2.0 & 2.0 \\
-2.0 & 2.0 & 2.0 & -2.0 & -2.0 & 2.0 & 2.0 & -2.0 \\
-2.0 & 2.0 & 2.0 & -2.0 & -2.0 & 2.0 & 2.0 & -2.0 \\
| . | . | . | . | . | . | . | . |
| . | . | . | . | . | . | . | . |
\end{bmatrix}
\]

**SCFT3 and DCFT3 (Complex Fourier Transform in Three Dimensions)**

**Purpose**

These subroutines compute the three-dimensional discrete Fourier transform of complex data.

**Table 230: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>scale</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y (Short)</td>
<td>Short-precision real</td>
<td>SCFT3</td>
</tr>
<tr>
<td>X, Y (Long)</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 34.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SCFT3</td>
</tr>
<tr>
<td>C and C++</td>
<td>scft3</td>
</tr>
</tbody>
</table>

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 230 on page 1041. 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 959. For more details, see “Notes” on page 1043.

- \(inc2x\) 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 956. 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 56.

- \(n2\) 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 956. 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 56.
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 956. 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 56.

isign
controls the direction of the transform, determining the sign \( \text{isign} \) of the exponents of \( W_{n1} \), \( W_{n2} \), and \( W_{n3} \), 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 \( \text{scale} \). See “Function” on page 1045 for its usage.
Specified as: a number of the data type indicated in Table 230 on page 1041, where \( \text{scale} > 0.0 \) or \( \text{scale} < 0.0 \).

aux
has the following meaning:
If \( \text{naux} = 0 \) and error 2015 is unrecoverable, \( \text{aux} \) is ignored.
Otherwise, it is a storage work area used by this subroutine.
Specified as: an area of storage, containing \( \text{naux} \) long-precision real numbers. On output, the contents are overwritten.

naux
is the number of doublewords in the working storage specified in \( \text{aux} \).
Specified as: an integer, where:
If \( \text{naux} = 0 \) 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, \( \text{naux} \geq (\text{minimum value required for successful processing}) \). 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 49.

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 \( 1, \text{inc2y}(\geq n1), \) and \( \text{inc3y}(\geq (n2)(\text{inc2y})) \), respectively.
Returned as: an array, containing numbers of the data type indicated in Table 230 on page 1041. 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 959. For more details, see “Notes” on page 1043.

Notes
1. If you specify the same array for \( X \) and \( Y \), then \( \text{inc2x} \) must be greater than or equal to \( \text{inc2y} \), and \( \text{inc3x} \) must be greater than or equal to \( \text{inc3y} \). In this case, output overwrites input. When using the ESSL SMP
Libraries in a multithreaded environment, if \(\text{inc2}x > \text{inc2}y\) or \(\text{inc3}x > \text{inc3}y\), 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 “Vector concepts” on page 71.

2. You should use “STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)” on page 1195 to determine the optimal values for the strides \(\text{inc2}y\) and \(\text{inc3}y\) 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 959.

Formulas
Processor-Independent Formulas for SCFT3 for NAUX:

Use the following formulas for calculating \(\text{naux}\):

**For 32-bit integer arguments:**

1. If \(\max(n2, n3) < 252\) and:
   - If \(n1 \leq 8192\), use \(\text{naux} = 60000\).
   - If \(n1 > 8192\), use \(\text{naux} = 60000+2.28n1\).

2. If \(n2 \geq 252\), \(n3 < 252\), and:
   - If \(n1 \leq 8192\), use \(\text{naux} = 60000+\lambda\).
   - If \(n1 > 8192\), use \(\text{naux} = 60000+2.28n1+\lambda\),
     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 \(\text{naux} = 60000+\psi\).
   - If \(n1 > 8192\), use \(\text{naux} = 60000+2.28n1+\psi\),
     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.

**For 64-bit integer arguments:**

1. If \(\max(n2, n3) < 252\) and:
   - If \(n1 \leq 8192\), use \(\text{naux} = 90000\).
   - If \(n1 > 8192\), use \(\text{naux} = 90000+2.28n1\).

2. If \(n2 \geq 252\), \(n3 < 252\), and:
   - If \(n1 \leq 8192\), use \(\text{naux} = 90000+\lambda\).
   - If \(n1 > 8192\), use \(\text{naux} = 90000+2.28n1+\lambda\),
     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 \(\text{naux} = 90000+\psi\).
   - If \(n1 > 8192\), use \(\text{naux} = 90000+2.28n1+\psi\),
     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 \( n_{aux} \):

**For 32-bit integer arguments:**

1. If \( \max(n_2, n_3) < 252 \) and:
   - If \( n_1 \leq 2048 \), use \( n_{aux} = 60000 \).
   - If \( n_1 > 2048 \), use \( n_{aux} = 60000 + 4.56n_1 \).
2. If \( n_2 \geq 252, n_3 < 252 \), and:
   - If \( n_1 \leq 2048 \), use \( n_{aux} = 60000 + \lambda \).
   - If \( n_1 > 2048 \), use \( n_{aux} = 60000 + 4.56n_1 + \lambda \),
     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 \( n_{aux} = 60000 + \psi \).
   - If \( n_1 > 2048 \), use \( n_{aux} = 60000 + 4.56n_1 + \psi \),
     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.

**For 64-bit integer arguments:**

1. If \( \max(n_2, n_3) < 252 \) and:
   - If \( n_1 \leq 2048 \), use \( n_{aux} = 90000 \).
   - If \( n_1 > 2048 \), use \( n_{aux} = 90000 + 4.56n_1 \).
2. If \( n_2 \geq 252, n_3 < 252 \), and:
   - If \( n_1 \leq 2048 \), use \( n_{aux} = 90000 + \lambda \).
   - If \( n_1 > 2048 \), use \( n_{aux} = 90000 + 4.56n_1 + \lambda \),
     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 \( n_{aux} = 90000 + \psi \).
   - If \( n_1 > 2048 \), use \( n_{aux} = 90000 + 4.56n_1 + \psi \),
     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:

\[
Y_{k_1,k_2,k_3} = scale \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \sum_{j_3=0}^{n_3-1} X_{j_1,j_2,j_3} \, W_{n_1}^{(I\text{sign})j_1k_1} \, W_{n_2}^{(I\text{sign})j_2k_2} \, W_{n_3}^{(I\text{sign})j_3k_3}
\]

for:
\begin{align*}
  W_{n1} &= e^{-2\pi i / n1} \\
  W_{n2} &= e^{-2\pi i / n2} \\
  W_{n3} &= e^{-2\pi i / n3}
\end{align*}

where:

- \(k_1 = 0, 1, \ldots, n_1 - 1\)
- \(k_2 = 0, 1, \ldots, n_2 - 1\)
- \(k_3 = 0, 1, \ldots, n_3 - 1\)

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 \(Y\).
- \(i_{\text{sign}}\) is + or - (determined by argument \(i_{\text{sign}}\)).
- \(\text{scale}\) is a scalar value.

For \(\text{scale} = 1.0\) and \(i_{\text{sign}}\) 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 \(i_{\text{sign}}\) being negative. See references [“1” on page 1253], [“5” on page 1253], [“6” on page 1253], [“27” on page 1254], and [“28” on page 1254].

Error conditions

Resource Errors
- Error 2015 is unrecoverable, \(n_{aux} = 0\), 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. \(\text{inc}_2y < n_1\)
7. \(\text{inc}_3y < (n_2)(\text{inc}_2y)\)
8. \(\text{scale} = 0.0\)
9. \(i_{\text{sign}} = 0\)
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 \(n_{aux} \neq 0\), and \(n_{aux}\) 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, \(\text{INC}_2X \geq \text{INC}_2Y\) and \(\text{INC}_3X \geq \text{INC}_3Y\), so that the same array can be used for both input and output. The \text{STRIDE} subroutine is called to select good values for the \(\text{INC}_2Y\) and \(\text{INC}_3Y\) strides. (As explained below, \text{STRIDE} is not called for \(\text{INC}_2X\) and \(\text{INC}_3X\).) Using the transform lengths (\(N_1 = 32, N_2 = 64,\) and \(N_3 = \ldots\)
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:

```c
CALL STRIDE (N2,N1,INC2Y,'C',0)
```

The output value returned for INC2Y is 32. Then STRIDE is called again for INC3Y:

```c
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:

```c
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:

```c
EQUIVALENCE (X,Y)
```

**Note:** Because NAUX = 0, this subroutine dynamically allocates the AUX working storage.

Call Statement and Input:

```c
CALL SCFT3( X , 32 , 2080 , Y , 32 , 2056 , 32 , 64 , 40 , 1 , SCALE , AUX , 0 )
```

SCALE = 1.0

X has (1.0,2.0) in location X(1,1,1) and (0.0,0.0) in all other locations.

Output:

Y has (1.0,2.0) in locations Y(ij,k), where i = 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 231: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, scale</td>
<td>Y</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>

**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 34.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SRCFT3</td>
</tr>
<tr>
<td>C and C++</td>
<td>srcft3</td>
</tr>
</tbody>
</table>

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 231 on page 1047. 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 959. For more details, see “Notes” on page 1049.

- **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; \( inc2y \geq n1/2+1 \).

- **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 956. 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 56.

- **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 956. 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 56.

- **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 956. 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 56.

**isign**
controls the direction of the transform, determining the sign $I_{sign}$ of the exponents of $W_{n1}$, $W_{n2}$, and $W_{n3}$, 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 $isign < 0$.

**scale**
is the scaling constant $scale$. See “Function” on page 1051 for its usage.

Specified as: a number of the data type indicated in Table 231 on page 1047, 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, $naux \geq$ (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 49.

**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 $1$, $inc2y(\geq n1/2+1)$, and $inc3y(\geq (n2)(inc2y))$, respectively.

Returned as: an array, containing numbers of the data type indicated in Table 231 on page 1047. 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 959. For more details, see “Notes” on page 1049.

**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 “Vector concepts” on page 71.

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 1195 to determine the optimal values for the strides inc2x and inc3x 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 959.

Formulas

**Processor-Independent Formulas for SRCFT3 for 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 \( n2 \geq 252 \), \( n3 < 252 \), 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 \( s = \min(64, 1+n1/2) \).
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 inc2x or inc3x is an odd number, or if array X is not aligned on a doubleword boundary, you should add the following amount to all the formulas given above:

\[ n2(1+n1/2) \]

**Processor-Independent Formulas for DRCFT3 for NAUX**

Use the following formulas for calculating naux:
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 \( n_{aux} = 92000 \).
   - If \( n_1 > 4096 \), use \( n_{aux} = 90000 + 2.78n_1 \).

2. If \( n_2 \geq 252, n_3 < 252 \), and:
   - If \( n_1 \leq 4096 \), use \( n_{aux} = 92000 + \lambda \).
   - If \( n_1 > 4096 \), use \( n_{aux} = 90000 + 2.78n_1 + \lambda \),
     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 \( n_{aux} = 92000 + \psi \).
   - If \( n_1 > 4096 \), use \( n_{aux} = 90000 + 2.78n_1 + \psi \),
     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:

\[
Y_{k_1, k_2, k_3} = \text{scale} \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \sum_{j_3=0}^{n_3-1} x_{j_1, j_2, j_3} W_{n_1}^{(j_1)k_1} W_{n_2}^{(j_2)k_2} W_{n_3}^{(j_3)k_3}
\]

for:
\( k_1 = 0, 1, ..., n_1 - 1 \)
\( k_2 = 0, 1, ..., n_2 - 1 \)
\( k_3 = 0, 1, ..., n_3 - 1 \)

where:

\[
\begin{align*}
W_{n1} &= e^{-2\pi(k_1 - 1)/n1} \\
W_{n2} &= e^{-2\pi(k_2 - 1)/n2} \\
W_{n3} &= e^{-2\pi(k_3 - 1)/n3}
\end{align*}
\]

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.

The output in array \( Y \) is complex. For \( scale = 1.0 \) and \( isign \) being positive, you obtain the discrete Fourier transform, a function of frequency. The inverse Fourier transform is obtained with \( scale = 1.0/((n1)(n2)(n3)) \) and \( isign \) being negative. See references ["1" on page 1253], ["5" on page 1253], ["6" on page 1253], ["27" on page 1254], and ["28" on page 1254].

**Error conditions**

**Resource Errors**

Error 2015 is unrecoverable, \( naux = 0 \), and unable to allocate work area.

**Computational Errors**

None

**Input-Argument Errors**

1. \( n1 > 37748736 \)
2. \( n2 > 37748736 \)
3. \( n3 > 37748736 \)
4. \( inc2x < n1 \)
5. \( inc3x < (n2)(inc2x) \)
6. \( inc2y < n1/2 + 1 \)
7. \( inc3y < (n2)(inc2y) \)
8. \( scale = 0.0 \)
9. \( isign = 0 \)
10. The length of one of the transforms in \( n1 \), \( n2 \), or \( n3 \) is not an allowable value. Return code 1 is returned if error 2030 is recoverable.
11. Error 2015 is recoverable or \( naux \neq 0 \), 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 compute a three-dimensional transform. In this example, \( INC2X \geq (2)(INC2Y) \) and \( INC3X \geq (2)(INC3Y) \), so that the same array can be used for both input and output. The STRIDE subroutine is called to select good values for the INC2Y and INC3Y strides. Using the
For the transform lengths \((N_1 = 32, N_2 = 64, \text{ and } N_2 = 40)\) along with the output data type (short-precision complex: \('C'\)), STRIDE is called once for each stride needed. First, it is called for INC2Y:

```fortran
CALL STRIDE (N2, N1/2+1, INC2Y, 'C', 0)
```

The output value returned for INC2Y is 17. (This value is equal to \(N_1/2+1\).) Then STRIDE is called again for INC3Y:

```fortran
CALL STRIDE (N3, N2*INC2Y, INC3Y, 'C', 0)
```

The output value returned for INC3Y is 1088. Because INC3Y is a multiple of INC2Y—that is, INC3Y = \((N_2)(INC2Y)\)—Y is declared as a three-dimensional array, \(Y(17, 64, 40)\). (In general, for larger arrays, these types of values for INC2Y and INC3Y are not returned by STRIDE, and you are probably not able to declare \(Y\) as a three-dimensional array.)

To equivalence the \(X\) and \(Y\) arrays requires \(INC2X \geq (2)(INC2Y)\) and \(INC3X \geq (2)(INC3Y)\). 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:

```fortran
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>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision complex</td>
<td>SCRFT2</td>
</tr>
<tr>
<td>Long-precision complex</td>
<td>DCRFT2</td>
</tr>
</tbody>
</table>

**Note:**

Fourier Transforms, Convolutions and Correlations, and Related Computations 1053
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 34.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SCRFT3</td>
</tr>
<tr>
<td>C and C++</td>
<td>scrft3</td>
</tr>
</tbody>
</table>

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 \(X((j1+2j2+3j3))\) for \(j1 = 0, 1, ..., n1/2, j2 = 0, 1, ..., n2-1,\) and \(j3 = 0, 1, ..., n3-1\). Due to complex conjugate symmetry, the input consists of only the first \(n1/2+1\) values along the first dimension of the array, for \(j1 = 0, 1, ..., n1/2\). The strides for the elements in the first, second, and third dimensions are assumed to be 1, \(inc2x \geq n1/2+1\), and \(inc3x \geq (n2)(inc2x)\), respectively.

Specified as: an array, containing numbers of the data type indicated in Table 232 on page 1053. 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 959. For more details, see “Notes” on page 1056.

\(inc2x\)

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 956. 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 56.

\(n2\)

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 956. 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 56.

\( n3 \)

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 956. 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 56.

\( isign \)

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 1057 for its usage.

Specified as: a number of the data type indicated in Table 232 on page 1053, 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, \( naux \geq \) (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 49.

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 \( k1 = 0, 1, \ldots, n1-1 \), \( k2 = 0, 1, \ldots, n2-1 \), and \( k3 = 0, 1, \ldots, n3-1 \). The strides for the elements in the first, second, and third dimensions are assumed to be 1, \( inc2y \geq n1+2 \), and \( inc3y \geq (n2)(inc2y) \), respectively.

Returned as: an array, containing numbers of the data type indicated in Table 232 on page 1053. 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 959. For more details, see “Notes” on page 1056.
Notes

1. If you specify the same array for X and Y, then inc2y must equal \((2)\)inc2x and inc3y must equal \((2)\)inc3x. In this case, output overwrites input. If you specify different arrays X and Y, they must have no common elements; otherwise, results are unpredictable. See “Vector concepts” on page 71.

2. You should use “STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)” on page 1195 to determine the optimal values for the strides inc2y and inc3y for your output array. To obtain the best performance, you should use inc2x = inc2y/2 and inc3x = inc3y/2. Example 9 in the STRIDE subroutine description explains how it is used for these subroutines. For additional information on how to set up your data, see “Setting Up Your Data” on page 959.

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 \(n2 \geq 252\), \(n3 \leq 252\), 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 \(n2 \geq 252\), \(n3 \leq 252\), 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 \(s = \min(64, 1+n1/2)\).

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 $\text{inc2y}$ or $\text{inc3y}$ 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 $\text{naux}$:

**For 32-bit integer arguments:**

1. If $\max(n2, n3) < 252$ and:
   - If $n1 \leq 4096$, use $\text{naux} = 62000$.
   - If $n1 > 4096$, use $\text{naux} = 60000+2.78n1$.
2. If $n2 \geq 252$, $n3 < 252$, and:
   - If $n1 \leq 4096$, use $\text{naux} = 62000+\lambda$.
   - If $n1 > 4096$, use $\text{naux} = 60000+2.78n1+\lambda$,
     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 $\text{naux} = 62000+\psi$.
   - If $n1 > 4096$, use $\text{naux} = 60000+2.78n1+\psi$,
     where $\psi = ((2)n3+256)(s+4.56)$
     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 $n1 \leq 4096$, 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 $s = \min(64, n1/2)$.
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 $s = \min(64, n2(1+n1/2))$.
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:
\[ y_{k_1,k_2,k_3} = scale \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \sum_{j_3=0}^{n_3-1} x_{j_1,j_2,j_3} \, W_{n_1}^{(i \text{sign})j_1k_1} \, W_{n_2}^{(i \text{sign})j_2k_2} \, W_{n_3}^{(i \text{sign})j_3k_3} \]

for:
\[ k_1 = 0, 1, \ldots, n_1 - 1 \]
\[ k_2 = 0, 1, \ldots, n_2 - 1 \]
\[ k_3 = 0, 1, \ldots, n_3 - 1 \]

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 \( Y \).
\( i \text{sign} \) is + or - (determined by argument \( i \text{sign} \)).
\( scale \) is a scalar value.

Because of the complex conjugate symmetry, the output in array \( Y \) is real. For \( scale = 1.0 \) and \( i \text{sign} \) being positive, you obtain the discrete Fourier transform, a function of frequency. The inverse Fourier transform is obtained with \( scale = 1.0/((n_1)(n_2)(n_3)) \) and \( i \text{sign} \) being negative. See references [“1” on page 1253], [“5” on page 1253], [“6” on page 1253], [“27” on page 1254], and [“28” on page 1254].

Error conditions

**Resource Errors**
Error 2015 is unrecoverable, \( naux = 0 \), 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. \( inc2x < n_1/2+1 \)
5. \( inc3x < (n_2)(inc2x) \)
6. \( inc2y < n_1+2 \)
7. \( inc3y < (n_2)(inc2y) \)
8. \( scale = 0.0 \)
9. \( i \text{sign} = 0 \)
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 \( naux\neq0 \), 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

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:

\[
\text{CALL STRIDE (N2,N1+2,INC2Y,'S',0)}
\]

The output value returned for INC2Y is 34. (This value is equal to N1+2.) Then STRIDE is called again for INC3Y:

\[
\text{CALL STRIDE (N3,N2+INC2Y,INC3Y,'S',0)}
\]

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:

\[
\begin{align*}
\text{COMPLEX*8} & \quad X(17,64,40) \\
\text{REAL*4} & \quad Y(34,64,40) \\
\text{REAL*8} & \quad \text{AUX}(1)
\end{align*}
\]

Arrays \(X\) and \(Y\) are made equivalent by the following statement, making them occupy the same storage:

\[
\text{EQUIVALENCE (X,Y)}
\]

Note: Because NAUX = 0, this subroutine dynamically allocates the AX working storage.

Call Statement and Input:

\[
\text{CALL SCRFT3( X , 17 , 1088 , Y , 34 , 2176 , 32 , 64 , 40 , 1 , SCALE , AUX , 0 )}
\]

\[
\text{SCALE = 1.0}
\]

\(X\) 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 960.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SCON</td>
</tr>
<tr>
<td>C and C++</td>
<td>scon</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 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 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 + (m-1)inc2x + (Nx-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.
**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; \( N_y > 0 \) for SCON and \( N_y \geq -N_h+1 \) for SCOR.

**aux1**

is no longer used in the computation, but must still be specified as a dummy argument (for migration purposes from Version 1 of ESSL). It can have any value.

**naux1**

is no longer used in the computation, but must still be specified as a dummy argument (for migration purposes from Version 1 of ESSL). It can have any value.

**aux2**

is no longer used in the computation, but must still be specified as a dummy argument (for migration purposes from Version 1 of ESSL). It can have any value.

**naux2**

is no longer used in the computation, but must still be specified as a dummy argument (for migration purposes from Version 1 of ESSL). It can have any value.

**On Return**

**y**

is array Y, consisting of \( m \) output sequences of length \( N_y \) that are the result of the convolutions or correlations of the sequence in array H with the sequences in array X. Returned as: an array of (at least) length \( 1 + (m-1)inc2y + (N_y-1)inc1y \), containing short-precision real numbers.

**Notes**

1. Output should not overwrite input; that is, input arrays X and H must have no common elements with output array Y. Otherwise, results are unpredictable. See “Vector concepts” on page 71.

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. 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:

\[ y_{ki} = \min \left( N_h - 1, k \right) \sum_{j=\max(0,k-N_x+1)}^{\min(1, N_y-1)} h_j x_{k-j,i} \]

Correlations for SCOR:

\[ y_{ki} = \min(1, N_x-1-k) \sum_{j=\min(0,-k)}^{\max(1, N_y-1)} h_j x_{k-j,i} \]

for:

\[ k = i y_0, i y_0 + 1, \ldots, i y_0 + N_y - 1 \]

\[ i = 1, 2, \ldots, m \]

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.

It is assumed that elements outside the range of definition are zero. See references [“25” on page 1254] and [“104” on page 1258].

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. \( nh, nx, ny, \) or \( m \leq 0 \)
2. \( \text{inc1h, inc1x, inc2x, inc1y, or inc2y} \leq 0 \)
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(0:499999)
REAL*8   AUX1, AUX2
```

Call Statement and Input:

```
INIT  H INC1H X INC1X INC2X Y INC1Y INC2Y NH NX  M IY0 NY  AUX1 NAUX1 AUX2 NAUX2
|    |   |   |   |     |   |   |     |    |  |  |  |   |   |     |    |     |
CALL SCON(INIT, H , 1 , X , 1 , 10 , Y , 1 , 15 , 4, 10, 3, 0, 15, AUX1 , 0 , AUX2 , 0)
```

**INIT = 0** (for computation)
**H = (1.0, 2.0, 3.0, 4.0)**

\( X \) contains the following three sequences:

- 1.0  2.0  3.0
- 2.0  1.0  2.0
- 3.0  2.0  1.0
- 4.0  3.0  2.0
- 5.0  4.0  3.0
- 6.0  5.0  4.0
- 5.0  6.0  5.0
- 4.0  5.0  6.0
- 3.0  4.0  5.0
- 2.0  3.0  4.0

Output:

\( Y \) contains the following three sequences:

- 1.0   2.0   3.0
- 4.0   5.0   8.0
- 10.0  10.0  14.0
- 20.0  18.0  22.0
- 30.0  20.0  18.0
- 40.0  30.0  20.0
- 48.0  40.0  30.0
- 52.0  48.0  40.0
- 50.0  52.0  48.0
- 40.0  50.0  52.0
- 29.0  38.0  47.0
- 18.0  25.0  32.0
- 8.0  12.0  16.0
- 0.0   0.0   0.0
- 0.0   0.0   0.0

Example 2

This example shows how the output from Example 1 differs when the values for \( NY \) and \( inc2y \) are 10 rather than 15. The output is the same except that it consists of only the first 10 values produced in Example 1.

Output:

\( Y \) contains the following three sequences:

- 1.0   2.0   3.0
- 4.0   5.0   8.0
- 10.0  10.0  14.0
- 20.0  18.0  22.0
- 30.0  20.0  18.0
- 40.0  30.0  20.0
- 48.0  40.0  30.0
Example 3

This example shows how the output from Example 2 differs if the value for IY0 is 3 rather than 0. The output is the same except it starts at element 3 of the convolution sequences rather than element 0.

Output:

Y contains the following three sequences:

| 20.0 | 18.0 | 22.0 |
| 30.0 | 20.0 | 18.0 |
| 40.0 | 30.0 | 20.0 |
| 48.0 | 40.0 | 30.0 |
| 52.0 | 48.0 | 40.0 |
| 50.0 | 52.0 | 48.0 |
| 40.0 | 50.0 | 52.0 |
| 29.0 | 38.0 | 47.0 |
| 18.0 | 25.0 | 32.0 |
| 8.0  | 12.0 | 16.0 |

Example 4

This example shows how to compute a correlation of a sequence in H, which is a ramp function, and three sequences in X, a triangular function and its cyclic translates. It computes the full range of nonzero values of the correlation plus two extra points, which are set to 0. The arrays are declared as follows:

| REAL*4   H(0:4999), X(0:4999), Y(0:4999) |
| REAL*8   AUX1, AUX2                   |

Call Statement and Input:

| INIT  H INC1H X INC1X INC2X Y INC1Y INC2Y NH NX  M IY0  NY  AUX1 NAUX1 AUX2 NAUX2 |
| CALL SCOR(INIT, H , 1 , X , 1  , 10 , Y , 1  , 15 , 4, 10, 3, -3, 15, AUX1 , 0 , AUX2 , 0) |

INIT = 0 (for computation)
H = (1.0, 2.0, 3.0, 4.0)
X contains the following three sequences:

| 1.0  | 2.0  | 3.0  |
| 2.0  | 1.0  | 2.0  |
| 3.0  | 2.0  | 1.0  |
| 4.0  | 3.0  | 2.0  |
| 5.0  | 4.0  | 3.0  |
| 6.0  | 5.0  | 4.0  |
| 5.0  | 6.0  | 5.0  |
| 4.0  | 5.0  | 6.0  |
| 3.0  | 4.0  | 5.0  |
| 2.0  | 3.0  | 4.0  |

Output:

Y contains the following three sequences:

| 4.0  | 8.0  | 12.0 |
| 11.0 | 10.0 | 17.0 |
| 20.0 | 15.0 | 16.0 |
| 30.0 | 22.0 | 18.0 |
| 40.0 | 30.0 | 22.0 |
| 50.0 | 40.0 | 30.0 |
| 52.0 | 50.0 | 40.0 |
| 48.0 | 52.0 | 50.0 |
| 40.0 | 48.0 | 52.0 |
| 30.0 | 40.0 | 48.0 |
| 16.0 | 22.0 | 28.0 |
| 7.0  | 16.0 | 13.0 |
| 2.0  | 3.0  | 4.0  |
Example 5

This example shows how the output from Example 4 differs when the values for NY and INC2Y are 10 rather than 15. The output is the same except that it consists of only the first 10 values produced in Example 4.

Output:

Y contains the following three sequences:

- 4.0 8.0 12.0
- 11.0 16.0 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 [“5” on page 1253].
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 “Use of SIMD Algorithms by Some Subroutines in the Libraries Provided by ESSL” on page 34.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SCOND</td>
</tr>
</tbody>
</table>
**Language**

<table>
<thead>
<tr>
<th>C and C++</th>
</tr>
</thead>
</table>

**Syntax**

scond | scord (h, inch, x, incx, y, incy, nh, nx, iy0, ny);

---

**On Entry**

- **h**
  - is the array H, consisting of the sequence of length \(N_h\) to be convolved or correlated with the sequence in array X.
  - Specified as: an array of (at least) length \(1+(N_h-1)|inch|\), containing short-precision real numbers.

- **inch**
  - is the stride between the elements within the sequence in array H.
  - Specified as: 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 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 \(1+(N_y-1)|incy|\), containing short-precision real numbers.

**Notes**

1. Output should not overwrite input—that is, input arrays X and H must have no common elements with output array Y. Otherwise, results are unpredictable. See “Vector concepts” on page 71.
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:**

\[
y_k = \min(N_h - 1, k) \sum_{j=\max(0, k - N_x + 1)}^{N_h - 1} h_j x_{k - j}
\]

**Correlation for SCORD:**

\[
y_k = \min(N_h - 1, N_x - 1 - k) \sum_{j=\max(0, -k)}^{N_h - 1} h_j x_{k + j}
\]

for \( k = iy0, iy0 + 1, \ldots, iy0 + ny - 1 \)

where:

\( y_k \) are elements of the sequence of length \( N_y \) in array \( Y \).

\( x_k \) are elements of the sequence of length \( N_x \) in array \( X \).

\( h_j \) are elements of the sequence of length \( N_h \) in array \( H \).

\( iy0 \) is the convolution or correlation index of the element to be stored in the first position of each sequence in array \( Y \).

\( \min \) and \( \max \) select the minimum and maximum values, respectively.

It is assumed that elements outside the range of definition are zero. See reference [“5” on page 1253].

**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:

```plaintext
IH  INCH  X  INCX  Y  INCY  NH  NX IY0  NY

CALL SCOND( H , 1  , X , 1  , Y , 1  , 4 , 8 , 0 , 11 )
```

\( H = (1.0, 2.0, 3.0, 4.0) \)

\( X = (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0) \)

Output:

\( Y = (11.0, 34.0, 70.0, 120.0, 130.0, 140.0, 150.0, 160.0, 151.0, 122.0, 72.0) \)

Example 2

This example shows how the output from Example 1 differs when the value for \( IY0 \) is -2 rather than 0, and \( NY \) is 15 rather than 11. The output has two zeros at the beginning and end of the sequence, for points outside the range of nonzero output.

Call Statement and Input:

```plaintext
IH  INCH  X  INCX  Y  INCY  NH  NX IY0  NY

CALL SCOND( H , 1  , X , 1  , Y , 1  , 4 , 8 , -2 , 15 )
```

\( H = (1.0, 2.0, 3.0, 4.0) \)

\( X = (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0) \)

Output:

\( Y = (0.0, 0.0, 11.0, 34.0, 70.0, 120.0, 130.0, 140.0, 150.0, 160.0, 151.0, 122.0, 72.0, 0.0, 0.0) \)

Example 3

This example shows how the same output as Example 1 can be obtained when \( H \) and \( X \) are interchanged, because the convolution is symmetric in \( H \) and \( X \). (The arguments are switched in the calling sequence.)

Call Statement and Input:

```plaintext
IH  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:

```plaintext
IH  INCH  X  INCX  Y  INCY  NH  NX IY0  NY

CALL SCOND( H , -1 , X , 1  , Y , 1  , 4 , 8 , 0 , 11 )
```
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:

```fortran
CALL SCOND( X, 1, X, 1, Y, 1, 4, 4, 0, 7 )
```

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 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:

```fortran
CALL SCORD( H, 1, X, 1, Y, 1, 4, 8, -3, 11 )
```

Output:

\[
Y = (44.0, 81.0, 110.0, 130.0, 140.0, 150.0, 160.0, 170.0, 104.0, 53.0, 18.0)
\]

Example 7

This example shows how the output from Example 6 differs when the value for \( IY0 \) is -5 rather than -3 and \( NY \) is 15 rather than 11. The output has two zeros at the beginning and end of the sequence, for points outside the range of nonzero output.

Call Statement and Input:

```fortran
CALL SCORD( H, 1, X, 1, Y, 1, 4, 8, -5, 15 )
```

Output:

\[
Y = (0.0, 0.0, 44.0, 81.0, 110.0, 130.0, 140.0, 150.0, 160.0, 170.0, 104.0, 53.0, 18.0, 0.0, 0.0)
\]

Example 8

This example shows how the output from Example 6 differs when \( H \) and \( X \) are interchanged (in the calling sequence). The output sequence is the reverse of that in Example 6. To get the full range of output, \( IY0 \) is set to \(-NX+1\).
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:

```
CALL SCORD( H , -1 , X , 1 , Y , 1 , 4 , 8 , -3 , 11 )
```

H = (1.0, 2.0, 3.0, 4.0)
X = (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)

Output:

Y = (11.0, 34.0, 70.0, 120.0, 130.0, 140.0, 150.0, 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:

```
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, 478.0, 630.0, 478.0, 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 34.
CALL SCONF | SCORF (init, h, inc1h, x, inc1x, inc2x, y, inc1y, inc2y, nh, nx, m, iy0, ny, aux1, aux1, aux2, aux2, aux2, aux2, aux2, aux2);

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.

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.

nh
is the number of elements, Nh, 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 \( init \neq 0 \), the working storage is computed.
If \( init = 0 \), the working storage is used in the computation of the convolutions.
Specified as: an area of storage, containing \( naux1 \) long-precision real numbers.

\( naux1 \)

is the number of doublewords in the working storage specified in \( aux1 \).
Specified as: an integer; \( naux1 > 23 \) (32-bit integer arguments) or 45 (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 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 49.

\( aux2 \)

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, SCONF and SCORF dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.
Otherwise, \( naux2 \geq \) (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 49.

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 “Vector concepts” on page 71.

5. If \( iy0 \) and \( ny \) are such that output outside the basic range is needed, the subroutine stores zeros. These ranges are: \( 0 \leq k \leq N_x+N_h-2 \) 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 \( naux1 \) and \( naux2 \), you must determine the length \( n \) of the Fourier transform. The value of \( n \) is based on \( nf \). You can use one of two techniques to determine \( nf \):

- Use the simple overestimate of \( nf = nx+nh-1 \). (If \( iy0 = 0 \) and \( ny > nh+nx \), this is the actual value, not an overestimate.)

- Use the values of the arguments \( iy0 \), \( nh \), \( nx \), and \( ny \) inserted into the following formulas to get a value for the variable \( nf \):

\[
\begin{align*}
\text{iy0p} &= \max(iy0, 0) \\
\text{ix0} &= \max((iy0p+1)-nh, 0) \\
\text{ih0} &= \max((iy0p+1)-nx, 0) \\
\text{nd} &= \text{ix0}+\text{ih0} \\
\text{n1} &= iy0+ny \\
\text{nxx} &= \min(n1, nx)-ix0 \\
\text{nhh} &= \min(n1, nh)-ih0 \\
\text{ntt} &= nxx+nhh-1 \\
\text{nn1} &= n1-nd \\
\text{iyy0} &= iy0p-nd \\
\text{nzleft} &= \max(0, nhh-iyy0-1) \\
\text{nzrt} &= \min(nn1, ntt)-nxx \\
\text{nf} &= \max(12, nxx+\max(nzleft, nzrt))
\end{align*}
\]

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 956, 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:

\[
y_{ki} = \min\left( N_h - 1, k \right) \sum_{j = \max(0, k - N_x + 1)}^{\min(N_h - 1, k)} h_j x_{k - j, i}
\]

Correlations for SCORF:

\[
y_{ki} = \min\left( N_h - 1, N_x - 1 - k \right) \sum_{j = \max(0, -k)}^{\min(N_h - 1, N_x - 1 - k)} h_j x_{k + j, i}
\]

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 \( naux1 \) and \( naux2 \) is the same length used by the Fourier transform subroutines called by this subroutine. It is assumed that elements outside the range of definition are zero. See references [“25” on page 1254] and [“104” on page 1258].
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, \( naux2 = 0 \), and unable to allocate work area.

Computational Errors
None

Input-Argument Errors

1. \( nh, nx, ny, \) or \( m \leq 0 \)
2. \( \text{inc}1h, \text{inc}1x, \text{inc}2x, \text{inc}1y, \) or \( \text{inc}2y \leq 0 \)
3. The resulting internal Fourier transform length \( n \), is too large. See “Convolutions and Correlations by Fourier Methods” on page 961.
4. The subroutine has not been initialized with the present arguments.
5. \( naux1 \leq 23 \)
6. \( naux1 \) is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
7. Error 2015 is recoverable or \( naux2 \neq 0 \), 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 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:

\[
\text{REAL*4} \quad H(8), X(10,1), Y(17)
\]

Because this convolution is symmetric in \( H \) and \( X \), you can interchange the \( H \) and \( X \) sequences, leaving all other arguments the same, and you get the same output shown below. First, initialize AUX1 using the calling sequence shown below with \( \text{INIT} \neq 0 \). 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:

\[
\text{INIT} \quad H \quad \text{INC}1H \quad X \quad \text{INC}1X \quad \text{INC}2X \quad Y \quad \text{INC}1Y \quad \text{INC}2Y \quad \text{NH} \quad \text{NX} \quad \text{M} \quad \text{IY}0 \quad \text{NY} \quad \text{AUX1} \quad \text{NAUX1} \quad \text{AUX2} \quad \text{NAUX2}
\]

\[
\text{CALL SCONF(INIT, H , 1 , X , 1  ,  1 , Y,  1  ,  1  , 8, 10, 1, 0, 17, AUX1, 128, AUX2,  0)
\]

\( \text{INIT} = 1 \) (for initialization)
\( \text{INIT} = 0 \) (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:

\[
Y \quad = \quad (11.0, 34.0, 70.0, 120.0, 185.0, 266.0, 364.0, 480.0,
516.0, 552.0, 567.0, 560.0, 530.0, 476.0, 397.0, 292.0, 160.0)
\]
Example 2

This example shows how the output from Example 1 differs when the value for NY is 21 rather than 17, and the value for IY0 is -2 rather than 0. This yields two zeros on each end of the convolution.

Output:

\[ Y = (0.0, 0.0, 11.0, 34.0, 70.0, 120.0, 185.0, 266.0, 364.0, 480.0, 516.0, 552.0, 567.0, 560.0, 530.0, 476.0, 397.0, 292.0, 160.0, 0.0, 0.0) \]

Example 3

This example shows how to compute the autoconvolution by letting the two input sequences be the same for Example 2. First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

**Note:** Because NAUX2 = 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```plaintext
INIT  H  INC1H X  INC1X INC2X Y  INC1Y INC2Y NH  NX  M  IY0  NY  AUX1  NAUX1  AUX2
|    |   |   |   |     |   |   |     |    |   |  |   |    |   |     |    |     |
CALL SCONF(INIT, H , 1 , H , 1 , 1 , Y, 1 , 1 , 8, 10, 1, -2, 21, AUX1, 128, AUX2, 0)
```

INIT = 1 (for initialization)
INIT = 0 (for computation)

Output:

\[ Y = (1.0, 4.0, 10.0, 20.0, 35.0, 56.0, 84.0, 120.0, 147.0, 164.0, 170.0, 164.0, 145.0, 112.0, 64.0) \]

Example 4

This example shows how to compute all nonzero values of the convolution of the sequence in H with the two sequences in X. First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

**Note:** Because NAUX2 = 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```plaintext
INIT  H  INC1H X  INC1X INC2X Y  INC1Y INC2Y NH  NX  M  IY0  NY  AUX1  NAUX1  AUX2  NAUX2
|    |   |   |   |     |   |   |     |    |   |  |  |   |   |     |    |     |
CALL SCONF(INIT, H , 1 , X, 10 , Y, 1 , 17 , 8, 10, 2, 0, 17, AUX1, 148, 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 contains the following two sequences:

\[
\begin{align*}
11.0 & \quad 12.0 \\
12.0 & \quad 13.0 \\
13.0 & \quad 14.0 \\
14.0 & \quad 15.0 \\
15.0 & \quad 16.0 \\
16.0 & \quad 17.0 \\
17.0 & \quad 18.0 \\
18.0 & \quad 19.0 \\
19.0 & \quad 20.0 \\
20.0 & \quad 11.0
\end{align*}
\]

Output:
Y contains the following two sequences:

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11.0</td>
<td>12.0</td>
<td>34.0</td>
<td>37.0</td>
<td>78.0</td>
<td>76.0</td>
<td>128.0</td>
<td>130.0</td>
<td>185.0</td>
<td>200.0</td>
<td>266.0</td>
<td>287.0</td>
</tr>
<tr>
<td>364.0</td>
<td>392.0</td>
<td>480.0</td>
<td>516.0</td>
<td>552.0</td>
<td>582.0</td>
<td>567.0</td>
<td>582.0</td>
<td>530.0</td>
<td>520.0</td>
<td>476.0</td>
<td>452.0</td>
</tr>
<tr>
<td>397.0</td>
<td>358.0</td>
<td>292.0</td>
<td>237.0</td>
<td>160.0</td>
<td>88.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

```fortran
REAL*4   H(8), X(10,1)
```

First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.

**Note:** Because NAUX2 = 0, this subroutine dynamically allocates the AUX2 working storage.

**Call Statement and Input:**

```
NAUX2
 CALL SCORF(INIT, H , 1 , X, 1  , 1 , Y, 1  , 1  , 8, 10, 1, -7, 17, AUX1, 128, AUX2, 0)
```

**INIT** = 1(for initialization)

**INIT** = 0(for computation)

**H** = (same as input H in Example 1)

**X** = (same as input X in Example 1)

**Output:**

Y = (88.0, 173.0, 254.0, 330.0, 400.0, 463.0, 518.0, 564.0, 600.0, 636.0, 504.0, 385.0, 280.0, 190.0, 116.0, 59.0, 20.0)

Example 6

This example shows how the output from Example 5 differs when the value for NY is 21 rather than 17, and the value for IY0 is -9 rather than 0. This yields two zeros on each end of the correlation.

**Output:**

Y = (0.0, 0.0, 88.0, 173.0, 254.0, 330.0, 400.0, 463.0, 518.0, 564.0, 600.0, 636.0, 504.0, 385.0, 280.0, 190.0, 116.0, 59.0, 20.0, 0.0, 0.0)

Example 7

This example shows the effect of interchanging H and X. It uses the same input as Example 5, with H and X switched in the calling sequence, and with IY0 with a value of -9. Unlike convolution, as noted in Example 1, the correlation is not symmetric in H and X. First, initialize AUX1 using the calling sequence shown below with INIT ≠ 0. Then use the same calling sequence with INIT = 0 to do the calculation.
Note: Because NAUX2 = 0, this subroutine dynamically allocates the AUX2 working storage.

Call Statement and Input:

```
INIT  H INC1H X INC1X INC2X Y INC1Y INC2Y NH NX M IY0 NY AUX1 NAUX1 AUX2 NAUX2

CALL SCONF(INIT, H, 1, X, 1, Y, 1, 8, 1, -7, 15, AUX1, 148, AUX2, 0)
```

INIT = 1 (for initialization)
INIT = 0 (for computation)

Output:

```
Y = (8.0, 23.0, 44.0, 70.0, 100.0, 133.0, 168.0, 204.0, 168.0,
133.0, 100.0, 70.0, 44.0, 23.0, 8.0)
```

Example 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:

```
INIT  H INC1H X INC1X INC2X Y INC1Y INC2Y NH NX M IY0 NY AUX1 NAUX1 AUX2 NAUX2

CALL SCONF(INIT, H, 1, X, 1, Y, 1, 8, 1, -7, 15, AUX1, 148, AUX2, 0)
```

INIT = 1 (for initialization)
INIT = 0 (for computation)

Output:

```
Y = (28.0, 59.0, 116.0, 190.0, 280.0, 385.0, 504.0, 636.0, 600.0,
564.0, 518.0, 463.0, 400.0, 330.0, 254.0, 173.0,
88.0)
```

Example 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:

```
INIT  H INC1H X INC1X INC2X Y INC1Y INC2Y NH NX M IY0 NY AUX1 NAUX1 AUX2 NAUX2

CALL SCONF(INIT, H, 1, X, 1, Y, 1, 8, 1, -7, 17, AUX1, 128, AUX2, 0)
```

INIT = 1 (for initialization)
INIT = 0 (for computation)

H = (same as input H in Example 4)
X = (same as input X in Example 4)
Y contains the following two sequences:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>88.0</td>
<td>96.0</td>
<td>173.0</td>
<td>188.0</td>
<td>254.0</td>
<td>275.0</td>
</tr>
<tr>
<td>330.0</td>
<td>356.0</td>
<td>400.0</td>
<td>430.0</td>
<td>463.0</td>
<td>496.0</td>
</tr>
<tr>
<td>518.0</td>
<td>553.0</td>
<td>564.0</td>
<td>600.0</td>
<td>600.0</td>
<td>636.0</td>
</tr>
<tr>
<td>636.0</td>
<td>592.0</td>
<td>504.0</td>
<td>462.0</td>
<td>385.0</td>
<td>346.0</td>
</tr>
<tr>
<td>280.0</td>
<td>245.0</td>
<td>190.0</td>
<td>160.0</td>
<td>116.0</td>
<td>92.0</td>
</tr>
<tr>
<td>20.0</td>
<td>11.0</td>
<td>59.0</td>
<td>42.0</td>
<td>20.0</td>
<td>11.0</td>
</tr>
</tbody>
</table>

**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 233: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h, x, y )</td>
<td>SDCON</td>
</tr>
<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 34.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SDCON</td>
</tr>
<tr>
<td>C and C++</td>
<td>sdcon</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 \(1+(N_h-1)|\text{inch}|\), containing numbers of the data type indicated in Table 233 on page 1079.

**\text{inch}**
is the stride between the elements within the sequence in array \(H\).
Specified as: an integer; \(\text{inch} > 0\) or \(\text{inch} < 0\).

**\(x\)**
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 \(1+(N_x-1)|\text{incx}|\), containing numbers of the data type indicated in Table 233 on page 1079.

**\text{incx}**
is the stride between the elements within the sequence in array \(X\).
Specified as: an integer; \(\text{incx} > 0\) or \(\text{incx} < 0\).

\(y\)
See On Return.

**\text{incy}**
is the stride between the elements within the sequence in output array \(Y\).
Specified as: an integer; \(\text{incy} > 0\) or \(\text{incy} < 0\).

**\(nh\)**
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; \(N_y > 0\).

**\(id\)**
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 \(1+(N_y-1)|\text{incy}|\), containing numbers of the data type indicated in Table 233 on page 1079.

**Notes**
1. If you specify the same array for \(X\) and \(Y\), the following conditions must be true: \(\text{incx} = \text{incy}\), \(\text{incx} > 0\), \(\text{incy} > 0\), \(id = 1\), and \(iy0 \geq N_h-1\) for \_DCON and \(iy0 \geq 0\) for \_DCOR. In this case, output overwrites input.
In all other cases, output should not overwrite input; that is, input arrays \(X\) and \(H\) must have no common elements with output array \(Y\). Otherwise, results are unpredictable. See “Vector concepts” on page 71.

2. If \(iy0\) and \(ny\) are such that output outside the basic range is needed, where the basic range is \(0 \leq k \leq (nh+nx-2)\) for SDCON and DDCON and is \((-nh+1) \leq k \leq (nx-1)\) for SDCOR and DDCOR, the subroutine stores zeros using scalar code. It is not efficient to store many zeros in this manner. If you anticipate that this will happen, you may want to adjust \(iy0\) and \(ny\), so the subroutine computes only for \(k\) in the above range, or use the ESSL subroutine SSCAL or DSCAL to store the zeros, so you achieve better performance.

**Function**

The convolution and correlation of a sequence in array \(H\) with a sequence in array \(X\), with decimated output, are expressed as follows:

**Convolution for SDCON and DDCON:**

\[
y_k = \min\left(\frac{N_h-1}{k}, \frac{N_x}{k} - 1\right) \sum_{j=\max(0, k - N_x + 1)}^{\min\left(\frac{N_h-1}{k}, \frac{N_x}{k} - 1 - k\right)} h_j x_{k-j}
\]

**Correlation for SDCOR and DDCOR:**

\[
y_k = \min\left(\frac{N_h-1}{k}, \frac{N_x}{k} - 1\right) \sum_{j=\max(0, -k)}^{\min\left(\frac{N_h}{k}, \frac{N_x}{k} - 1 - k\right)} h_j x_{k+j}
\]

for \(k = iy0, iy0+id, iy0+(2) id, ..., iy0+(Ny-1) id\)

where:

- \(y_k\) are elements of the sequence of length \(Ny\) in array \(Y\).
- \(x_k\) are elements of the sequence of length \(Nx\) in array \(X\).
- \(h_j\) are elements of the sequence of length \(Nh\) 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 [“5” on page 1253].

**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**
1. $nh$, $nx$, or $ny \leq 0$
2. $inch$, $incx$, or $incy = 0$
3. $id \leq 0$

**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. 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:

```plaintext
CALL SDCON( H, 1, X, 1, Y, 1, 4, 8, 0, 6, 2 )
```

$H = (1.0, 2.0, 3.0, 4.0)$

$X = (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)$

Output:

$Y = (11.0, 70.0, 130.0, 150.0, 151.0, 72.0)$

**Example 2**

This example shows how to compute a correlation of a sequence in $H$ with a sequence in $X$, where both sequences are ramp functions. It shows how a decimated output can be obtained, using the same input as Example 6 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:

```plaintext
CALL SDCOR( H, 1, X, 1, Y, 1, 4, 8, -3, 6, 2 )
```

$H = (1.0, 2.0, 3.0, 4.0)$

$X = (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)$

Output:

$Y = (44.0, 110.0, 140.0, 160.0, 104.0, 18.0)$

**Example 3**

This example shows how to compute the same function as computed in Example 1 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:

<table>
<thead>
<tr>
<th></th>
<th>H</th>
<th>INCH</th>
<th>X</th>
<th>INCX</th>
<th>Y</th>
<th>INCY</th>
<th>NH</th>
<th>NX</th>
<th>IY0</th>
<th>NY</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>H</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>Y</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

CALL SDCON( H , 1 , X , 1 , Y , 1 , 4 , 8 , 0 , 11 , 1 )

H = (1.0, 2.0, 3.0, 4.0)
X = (11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0)

Output:

Y = (11.0, 34.0, 70.0, 120.0, 130.0, 140.0, 150.0, 160.0, 151.0, 122.0, 72.0)

SACOR (Autocorrelation of One or More Sequences)

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 [“5” on page 1253].

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SACOR (init, x, inc1x, inc2x, y, inc1y, inc2y, nx, m, ny, aux1, aux1, aux2, aux2)</td>
</tr>
<tr>
<td>C and C++</td>
<td>sacor (init, x, inc1x, inc2x, y, inc1y, inc2y, nx, m, ny, aux1, 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, 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 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 “Vector concepts” on page 71.
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. 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:
\( k = 0, 1, \ldots, N_y - 1 \)
\( i = 1, 2, \ldots, m \)

where:

\( y_{ki} \) are elements of the \( m \) sequences of length \( N_y \) in array \( Y \).
\( x_{ji} \) and \( x_{j+i, k} \) are elements of the \( m \) sequences of length \( N_x \) in array \( X \).

See references (“25” on page 1254) and (“104” on page 1258).

Only one invocation of this subroutine is needed:

1. You do not need to invoke the subroutine with \( init \neq 0 \). If you do, however, the subroutine performs error checking, exits back to the calling program, and no computation is performed.
2. With \( init = 0 \), the subroutine performs the calculation of the convolutions or correlations.

**Error conditions**

**Computational Errors**

None

**Input-Argument Errors**

1. \( nx, ny, \) or \( m \leq 0 \)
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:**

```
INIT  X INC1X INC2X Y INC1Y INC2Y NX  M   NY  AUX1 NAUX1 AUX2   NAUX2
|    |   |     |   |   |     |   |   |   |    |     |    |       |
CALL SACOR(INIT, X , 1  ,  7 , Y , 1  ,  7 , 7 , 3 , 7 , AUX1 , 0 , AUX2 ,   0)
```

\( INIT = 0 \) (for computation)

\( X \) contains the following three sequences:

\[
1.0 \quad 2.0 \quad 3.0 \\
2.0 \quad 1.0 \quad 2.0 \\
3.0 \quad 2.0 \quad 1.0 \\
4.0 \quad 3.0 \quad 2.0 \\
4.0 \quad 4.0 \quad 3.0 \\
3.0 \quad 4.0 \quad 4.0 \\
2.0 \quad 3.0 \quad 4.0
\]

**Output:**

\( Y \) contains the following three sequences:

\[
59.0 \quad 59.0 \quad 59.0 \\
54.0 \quad 50.0 \quad 44.0 \\
43.0 \quad 39.0 \quad 30.0 \\
29.0 \quad 27.0 \quad 24.0 \\
16.0 \quad 18.0 \quad 21.0 \\
7.0 \quad 11.0 \quad 20.0 \\
2.0 \quad 6.0 \quad 12.0
\]
Example 2

This example shows how the output from Example 1 differs when the values for \( NY \) and \( INC2Y \) are 9 rather than 7. This shows that when \( NY \) is greater than \( NX \), the output array is longer, and that part is filled with zeros.

Output:

\( Y \) contains the following three sequences:

\[
\begin{array}{cccc}
59.0 & 59.0 & 59.0 \\
54.0 & 50.0 & 44.0 \\
43.0 & 39.0 & 30.0 \\
29.0 & 27.0 & 24.0 \\
16.0 & 18.0 & 21.0 \\
7.0 & 11.0 & 20.0 \\
2.0 & 6.0 & 12.0 \\
0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 \\
\end{array}
\]

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:

\[
\begin{array}{cccc}
59.0 & 59.0 & 59.0 \\
54.0 & 50.0 & 44.0 \\
43.0 & 39.0 & 30.0 \\
29.0 & 27.0 & 24.0 \\
16.0 & 18.0 & 21.0 \\
\end{array}
\]

**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 34.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SACORF (init, x, inc1x, inc2x, y, inc1y, inc2y, nx, m, ny, aux1, aux1, aux2, aux2)</td>
</tr>
<tr>
<td>C and C++</td>
<td>sacorf (init, x, inc1x, inc2x, y, inc1y, inc2y, nx, m, ny, aux1, aux1, aux2, aux2);</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 \( \text{init} = 0 \), the autocorrelations of the sequence in \( x \) 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 \( m \) input sequences of length \( N_x \), to be autocorrelated. Specified as: an array of (at least) length \( 1 + (N_x - 1) \text{inc1x} + (m - 1) \text{inc2x} \), containing short-precision real numbers.

\( y \) See On Return.

\( \text{inc1x} \) is the stride between the elements within each sequence in array \( X \).

Specified as: an integer; \( \text{inc1x} > 0 \).

\( \text{inc2x} \) is the stride between the first elements of the sequences in array \( X \).

Specified as: an integer; \( \text{inc2x} > 0 \).

\( \text{inc1y} \) is the stride between the elements within each sequence in output array \( Y \).

Specified as: an integer; \( \text{inc1y} > 0 \).

\( \text{inc2y} \) is the stride between the first elements of each sequence in output array \( Y \).

Specified as: an integer; \( \text{inc2y} > 0 \).

\( n_x \) 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; \( m > 0 \).

\( n_y \) is the number of elements, \( N_y \), in each sequence in array \( Y \).

Specified as: an integer; \( N_y > 0 \).

\( \text{aux1} \) 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 autocorrelations.

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; \( \text{naux1} > 21 \) (32-bit integer arguments) or 43 (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 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 49.

\( \text{aux2} \) 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 \( naux2 \) 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, \( naux2 \geq (\text{minimum value required for successful processing}) \). 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 49.

**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.

\( aux1 \)
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 “Vector concepts” on page 71.

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 \( naux1 \) and \( naux2 \), you must determine the length \( n \) of the Fourier transform. To do this, you use the values of the arguments \( nx \) and \( ny \), inserted into the following formula, to get a value for the variable \( nf \). After calculating \( nf \), reference the formula or table of allowable values of \( n \) in “Acceptable Lengths for the Transforms” on page 956, selecting the value equal to or greater than \( nf \). Following is the formula for determining \( nf \):
\[ nf = \min(ny, nx) + nx + 1 \]

**Processor-Independent Formulas for NAUX1 and NAUX2**

The required values of \( naux1 \) and \( naux2 \) depend on the value determined for \( n \) in Formula for Calculating the Length of the Fourier Transform and the argument \( m \).

**NAUX1 Formulas:**

**For 32-bit integer arguments:**
- If \( n \leq 16384 \), use \( naux1 = 55000 \).
- If \( n > 16384 \), use \( naux1 = 40000 + 1.89n \).

**For 64-bit integer arguments:**
- If \( n \leq 16384 \), use \( naux1 = 75000 \).
- If \( n > 16384 \), use \( naux1 = 60000 + 1.89n \).  

**NAUX2 Formulas:**
- If \( n \leq 16384 \), use \( naux2 = 50000 \).
- If \( n > 16384 \), use \( naux2 = 40000 + 1.64n \).

**Function**

The autocorrelations of the sequences in array \( X \) are expressed as follows:

\[ y_{ki} = \sum_{j=0}^{N_x - 1 - k} x_{ji} x_{j+k,i} \]

for:

- \( k = 0, 1, \ldots, N_y - 1 \)
- \( i = 1, 2, \ldots, m \)

where:

- \( y_{ki} \) are elements of the \( m \) sequences of length \( N_y \) in array \( Y \).
- \( x_{ji} \) and \( x_{j+k,i} \) are elements of the \( m \) sequences of length \( N_x \) in array \( X \).

This subroutine uses a Fourier transform method with a mixed-radix capability. This provides maximum performance for your application. The length of the transform, \( n \), that you must calculate to determine the correct sizes for \( naux1 \) and \( naux2 \) is the same length used by the Fourier transform subroutines called by this subroutine. See references [“25” on page 1254] and [“104” on page 1258].

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, \( naux2 = 0 \), and unable to allocate work area.

**Computational Errors**
- None
Input-Argument Errors

1. \( nx, ny, \text{ or } m \leq 0 \)
2. \( \text{inc1x, inc2x, inc1y, \text{ or } inc2y} \leq 0 \) (or incompatible)
3. The resulting correlation is too long.
4. The subroutine has not been initialized with the present arguments.
5. \( naux1 \leq 21 \)
6. \( naux1 \) is too small—that is, less than the minimum required value. Return code 1 is returned if error 2015 is recoverable.
7. Error 2015 is recoverable or \( naux2 \neq 0 \), 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 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:

\[
\begin{array}{cccc}
\text{REAL*4} & X(0:49999), Y(0:49999) \\
\text{REAL*8} & AUX1(2959), AUX2(1)
\end{array}
\]

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 \( NAUX2 \neq 0 \), this subroutine dynamically allocates the \( AUX2 \) working storage.

Call Statement and Input:

\[
\begin{array}{cccccccccccc}
\text{INIT} & X & \text{INC1X} & \text{INC2X} & Y & \text{INC1Y} & \text{INC2Y} & NX & M & NY & AUX1 & NAUX1 & AUX2 & NAUX2 \\
0 & & & & & & & & & & & & & &
\end{array}
\]

\[
\begin{array}{cccccccccccc}
\text{CALL SACORF(INIT, X , 1  ,  7 , Y , 1  ,  7 , 7 , 3 , 7 , AUX1, 2959, AUX2, 0)}
\end{array}
\]

**INIT** = 1 (for initialization)
**INIT** = 0 (for computation)

\( X \) contains the following three sequences:

\[
\begin{array}{cccc}
1.0 & 2.0 & 3.0 \\
2.0 & 1.0 & 2.0 \\
3.0 & 2.0 & 1.0 \\
4.0 & 3.0 & 2.0 \\
4.0 & 4.0 & 3.0 \\
3.0 & 4.0 & 4.0 \\
2.0 & 3.0 & 4.0 \\
\end{array}
\]

Output:

\( Y \) contains the following three sequences:

\[
\begin{array}{cccc}
59.0 & 59.0 & 59.0 \\
54.0 & 50.0 & 44.0 \\
43.0 & 39.0 & 38.0 \\
29.0 & 27.0 & 24.0 \\
16.0 & 18.0 & 21.0 \\
7.0 & 11.0 & 20.0 \\
2.0 & 6.0 & 12.0 \\
\end{array}
\]

Example 2

This example shows how the output from Example 1 differs when the value for \( NY \) and \( \text{INC2Y} \) 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:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>59.0</td>
<td>59.0</td>
<td>59.0</td>
</tr>
<tr>
<td>54.0</td>
<td>50.0</td>
<td>44.0</td>
</tr>
<tr>
<td>43.0</td>
<td>39.0</td>
<td>30.0</td>
</tr>
<tr>
<td>29.0</td>
<td>27.0</td>
<td>24.0</td>
</tr>
<tr>
<td>16.0</td>
<td>18.0</td>
<td>21.0</td>
</tr>
<tr>
<td>7.0</td>
<td>11.0</td>
<td>20.0</td>
</tr>
<tr>
<td>2.0</td>
<td>6.0</td>
<td>12.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

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:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>59.0</td>
<td>59.0</td>
<td>59.0</td>
</tr>
<tr>
<td>54.0</td>
<td>50.0</td>
<td>44.0</td>
</tr>
<tr>
<td>43.0</td>
<td>39.0</td>
<td>30.0</td>
</tr>
<tr>
<td>29.0</td>
<td>27.0</td>
<td>24.0</td>
</tr>
<tr>
<td>16.0</td>
<td>18.0</td>
<td>21.0</td>
</tr>
</tbody>
</table>

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 \(y\):

\[
y_i = u_0 + u_1 x_i + u_2 x_i^2 + \ldots + u_k x_i^k \quad \text{for} \quad i = 1, 2, \ldots, n
\]

where \(u_k, x_i, \) and \(y_i\) are elements of \(u, x, \) and \(y\), respectively.

### Table 234: Data Types.

<table>
<thead>
<tr>
<th>(u, x, y)</th>
<th>Subroutine</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>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SPOLY</td>
</tr>
<tr>
<td>C and C++</td>
<td>spoly</td>
</tr>
</tbody>
</table>

Fourier Transforms, Convolutions and Correlations, and Related Computations 1091
On Entry

$u$ is the coefficient vector $u$ of length $k+1$. It contains elements $u_0, u_1, u_0, u_1, u_2, ... , u_k$, which are stored in this order. Specified as: a one-dimensional array of (at least) length $1+|incu|$, containing numbers of the data type indicated in Table 234 on page 1091.

$incu$ 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 $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 234 on page 1091.

$incx$ is the stride for vector $x$.
Specified as: an integer. It can have any value.

$y$ See On Return.

$incy$ 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 234 on page 1091.

Notes

Vectors $u, x$, and $y$ must have no common elements; otherwise, results are unpredictable. See “Vector concepts” on page 71.

Function

The evaluation of the polynomial:

$$y_i = u_0 + u_1 x_i + u_2 x_i^2 + \ldots + u_k x_i^k \quad \text{for } i = 1, 2, \ldots, n$$

is expressed as follows:

$$y_i = u_0 + x_i (u_1 + x_i (u_2 + \ldots + x_i (u_{k-1} + x_i u_k) \ldots) \quad \text{for } i = 1, 2, \ldots, n$$

See reference [“100” on page 1258] 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:

```
U   INCU   K   X   INCX   Y   INCY   N
|    |     |   |    |     |   |    | 
CALL SPOLY( U , INCU , 0 , X , INCX , Y , 1 , 3 )
```

$U = (4.0)$
$INCU = (not relevant)$
$X = (not relevant)$
$INCX = (not relevant)$

Output:

$Y = (4.0, 4.0, 4.0)$

Example 2
This example shows a polynomial evaluation, using a negative stride $INCU$ for vector $u$. For $u$, processing begins at element $U(4)$ which is 1.0.

Call Statement and Input:

```
U  INCU  K   X  INCX  Y  INCY  N
|   |    |   |   |    |   |    | 
CALL SPOLY( U , -1 , 3 , X , 1 , Y , 1 , 3 )
```

$U = (4.0, 3.0, 2.0, 1.0)$
$INCU = (not relevant)$
$X = (2.0, 1.0, -3.0)$

Output:

$Y = (49.0, 10.0, -86.0)$

Example 3
This example shows a polynomial evaluation, using a stride $INCX$ of 0 for input vector $x$.

Call Statement and Input:

```
U  INCU  K   X  INCX  Y  INCY  N
|   |    |   |   |    |   |    | 
CALL SPOLY( U , 1 , 3 , X , 0 , Y , 1 , 3 )
```

$U = (4.0, 3.0, 2.0, 1.0)$
$INCU = (not relevant)$
$X = (2.0, ., .)$

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:

```
<table>
<thead>
<tr>
<th>U</th>
<th>INCU</th>
<th>K</th>
<th>X</th>
<th>INCX</th>
<th>Y</th>
<th>INCY</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL SPOLY( U, 1, 3, X, 2, Y, -2, 3 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U   = (4.0, 3.0, 2.0, 1.0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X   = (2.0, . , -3.0, . , 1.0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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 235: Data Types.**

<table>
<thead>
<tr>
<th>( x )</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>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SIZC</td>
</tr>
<tr>
<td>C and C++</td>
<td>sizc</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 235 on page 1094.

\( idrx \)  
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 **ky**₁ and **ky**₂, where:

If the \( i \)-th zero crossing is found:

• **ky**₁ = \( j \), where \( j \) is the position of the element \( x_j \) at the point that the \( i \)-th zero crossing is found. The position is always relative to the beginning of the vector regardless of the scan direction.

• **ky**₂ = \( i \)

If the \( i \)-th zero crossing is not found:

• **ky**₁ = 0

• **ky**₂ = the total number of zero crossings encountered in the scan.

Returned as: an array of (at least) length 2, containing 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 **ky**₁ as the position of the \( i \)-th zero crossing, and \( i \) is returned in **ky**₂.

If \( idrx \geq 0 \):

TRUE = (\( x_{j-1} < 0 \) and \( x_j \geq 0 \)) or (\( x_{j-1} \geq 0 \) and \( x_j < 0 \)) for \( j = 2, n \)

If \( idrx < 0 \):

TRUE = (\( x_{j+1} < 0 \) and \( x_j \geq 0 \)) or (\( x_{j+1} \geq 0 \) 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 **y**₁ and the number of zero crossings encountered in the scan is returned in **y**₂.

SIZC provides the same functions as the IBM 3838 functions NZCP and NZCN, with restrictions removed. It combines these functions into one ESSL subroutine. DIZC provides a long-precision computation that is not included in the IBM 3838 functions. See the IBM 3838 Array Processor Functional Characteristics manual.

**Error conditions**

**Computational Errors**

None

**Input-Argument Errors**

1. \( n \leq 1 \)
2. \( i \leq 0 \)
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:

```
CALL SIZC( X , 1 , 12 , 5 , KY )
```

<table>
<thead>
<tr>
<th>X</th>
<th>IDRX</th>
<th>N</th>
<th>I</th>
<th>KY</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0</td>
<td>1.0</td>
<td>-3.0</td>
<td>0.0</td>
<td>8.0</td>
</tr>
<tr>
<td></td>
<td>2.0</td>
<td>-9.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output:

```
KY = (9, 5)
```

Example 2

This example shows a scan of a vector $x$ from the last element to the first. It is looking for the seventh zero crossing, which is encountered at position 3. Because IDRX is negative, X is scanned from the last element, $X(12)$, to the first element, $X(1)$.

Call Statement and Input:

```
CALL SIZC( X , -1 , 12 , 7 , KY )
```

<table>
<thead>
<tr>
<th>X</th>
<th>IDRX</th>
<th>N</th>
<th>I</th>
<th>KY</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0</td>
<td>-1.0</td>
<td>3.0</td>
<td>-3.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>2.0</td>
<td>-9.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

```
CALL SIZC( X , 1 , 12 , 10 , KY )
```

<table>
<thead>
<tr>
<th>X</th>
<th>IDRX</th>
<th>N</th>
<th>I</th>
<th>KY</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0</td>
<td>-1.0</td>
<td>3.0</td>
<td>0.0</td>
<td>8.0</td>
</tr>
<tr>
<td></td>
<td>2.0</td>
<td>-9.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 $y$. 

1096 IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and Reference
Table 236: 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

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL STREC</td>
</tr>
<tr>
<td>C and C++</td>
<td>strec</td>
</tr>
</tbody>
</table>

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 236 on page 1097.

u

is the target vector u of length n.

Specified as: a one-dimensional array of (at least) length 1+\( (n-1)|\text{incu}| \), containing numbers of the data type indicated in Table 236 on page 1097.

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+\( (n-1)|\text{incx}| \), containing numbers of the data type indicated in Table 236 on page 1097.

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 ≥ 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 236 on page 1097.
Notes
Vectors $u, x$, and $y$ must have no common elements; otherwise, results are unpredictable. See “Vector concepts” on page 71.

Function
The first-order time-varying recursive equation is expressed as follows:

\[
\begin{align*}
y_1 &= s + u_1 x_1 \\
y_2 &= u_2 y_1 + u_1 x_2 \\
\vdots \\
y_i &= u_i y_{i-1} + u_1 x_i 	ext{ for } i = 3, 4, \ldots, n
\end{align*}
\]

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:

<table>
<thead>
<tr>
<th>S</th>
<th>U</th>
<th>INCU</th>
<th>X</th>
<th>INCX</th>
<th>Y</th>
<th>INCY</th>
<th>N</th>
<th>IOPT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALL STREC( 1.0 , U , 1 , X , 1 , Y , 1 , 8 , 0 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>=</td>
<td>(1.0, 2.0, 3.0, 3.0, 2.0, 1.0, 1.0, 2.0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>=</td>
<td>(3.0, 2.0, 1.0, 1.0, 2.0, 3.0, 3.0, 2.0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>S</th>
<th>U</th>
<th>INCU</th>
<th>X</th>
<th>INCX</th>
<th>Y</th>
<th>INCY</th>
<th>N</th>
<th>IOPT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALL STREC( 1.0 , U , 2 , X , 1 , Y , 1 , 4 , 0 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>=</td>
<td>(1.0, , 3.0, , 2.0, , 1.0, , )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>=</td>
<td>(3.0, 2.0, 1.0, 1.0, 2.0, 3.0, 3.0, 2.0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Output:
\[ Y = (4.0, 14.0, 29.0, 30.0) \]

Example 3
This example shows a stride, \( \text{INCU} \), of 1 for vector \( u \), a stride, \( \text{INCX} \), that is greater than 1 for vector \( x \), and a negative stride, \( \text{INCY} \), for vector \( y \). For \( y \), results are stored beginning at element \( Y(4) \).

Call Statement and Input:
\[
\begin{array}{cccccccc}
S & U & \text{INCU} & X & \text{INCX} & Y & \text{INCY} & N & \text{IOPT}
\end{array}
\]
\[
\text{CALL STREC}( 1.0 , U , 1 , X , 2 , Y , -1 , 4 , 1 )
\]
\[
U = (1.0, 2.0, 3.0, 3.0, 2.0, 1.0, 1.0, 2.0)
\]
\[
X = (3.0, \ , 1.0, \ , 2.0, \ , 3.0)
\]

Output:
\[ Y = (90.0, 29.0, 9.0, 4.0) \]

SQINT and DQINT (Quadratic Interpolation)

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 237: 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>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SQINT</td>
</tr>
<tr>
<td>C and C++</td>
<td>sqint</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 237 on page 1099.
- \( g \) is the scalar \( g \), containing the sample interval.
  - Specified as: a number of the data type indicated in Table 237 on page 1099; \( g > 0.0 \).
- \( \text{omega} \) is the output scaling parameter \( \Omega \).
  - Specified as: a number of the data type indicated in Table 237 on page 1099.
- \( x \) is the vector \( x \) of length \( n \), containing the trace data.
Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 237 on page 1099.

$incx$

is the stride for vector $x$.
Specified as: an integer; $incx > 0$ or $incx < 0$.

$n$

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 $1+(m-1)|inct|$, containing numbers of the data type indicated in Table 237 on page 1099.

$inct$

is the stride for vector $t$.
Specified as: an integer; $inct > 0$ or $inct < 0$.

$y$

See On Return.

$incy$

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 237 on page 1099.

**Function**

The quadratic interpolation, which is expressed as follows:

$$y_i = \Omega \left( trace_{k_i} \left( t_i^2 - f_i \right) + 2 \; trace_{k_i+1} \left( 1 - f_i^2 \right) + trace_{k_i+2} \left( f_i^2 + t_i \right) \right)$$

for $i = 1, 2, \ldots, m$

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:

- $trace_1 = 3x_1 - 3x_2 + x_3$
- $trace_{i+1} = x_i$ for $i = 1, 2, \ldots, n$
so = s + 2.0  

\[ w_i = \frac{so + t_i}{g} \]  for \( i = 1, 2, ..., m \)  

\[ f_i = \text{fraction part of } w_i \]  

\[ k_i + 1 = \text{integer part of } w_i \]  

**Note:** Allowing \( k_i + 1 \) to have a value of 2 results in performance degradation. If possible, avoid specifying a point at which this occurs.  

If \( n \) or \( m \) is 0, no computation is performed.  

SQINT provides the same function as the IBM 3838 function INT, with restrictions removed. DQINT provides a long-precision computation that is not included in the IBM 3838 functions. See the *IBM 3838 Array Processor Functional Characteristics* manual.  

### Error conditions  

#### Computational Errors  

The condition \((k_i + 1 > n)\) or \((k_i + 1 \leq 2)\) has occurred, where \( n \) is the number of elements in vector \( x \). See “Function” on page 1100 for how to calculate \( k_i \).  

- The lower range \( l \) and the upper range \( j \) of the vector are identified in the computational error message.  
- The return code is set to 1.  
- The ranges \( l \) and \( j \) of the vector can be determined at run time by using the ESSL error-handling facilities. To obtain this information, you must use ERRSET to change the number of allowable errors for error code 2100 in the ESSL error option table; otherwise, the default value causes your program to terminate when this error occurs. For details, see “What Can You Do about ESSL Computational Errors?” on page 65.  

#### Input-Argument Errors  

1. \( n < 3 \)  
2. \( m < 0 \)  
3. \( g \leq 0 \)  
4. \( \text{incx} = 0 \)  
5. \( \text{inct} = 0 \)  
6. \( \text{incy} = 0 \)  

### Examples  

#### Example 1  

This example shows a quadratic interpolation, using vectors with strides of 1.  

Call Statement and Input:  

```plaintext  
S     G   OMEGA  X  INCX  N   T  INCT  Y  INCY  M  
|     |     |    |   |    |   |   |    |   |    | 
CALL SQINT( 2.0 , 1.0 , 1.0 , X , 1  , 8 , T , 1  , Y , 1  , 4 )  
X        =  (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0)  
T        =  (1.5, 2.5, 3.5, 4.5)  
```

Output:  

```plaintext  
Y        =  (9.0, 11.0, 13.0, 15.0)  
```

#### Example 2  

This example shows a quadratic interpolation, using vectors with a positive stride of 1 and negative strides of -1.
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 , -1 , 8 , T , -1 , Y , 1 , 4 )
```
```
X = (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0)
T = (1.5, 2.5, 3.5, 4.5)
```
Output:
```
Y = (3.0, 5.0, 7.0, 9.0)
```

Example 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:
```
CALL SQINT( 3.0 , 10.0 , 1.0 , X , 1 , 8 , T , 2 , Y , 3 , 4 )
```
```
X = (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0)
T = (1.5, . , 2.5, . , 3.5, . , 4.5)
```
Output:
```
Y = (8.3, . , . , 8.5, . , . , 8.7, . , . , 8.9)
```

SWLEV, DWLEV, CWLEV, and ZWLEV (Wiener-Levinson Filter Coefficients)

Purpose
These subroutines compute the coefficients of an n-point Wiener-Levinson filter, using vector \( \mathbf{x} \), the trace for which the filter is to be designed, and vector \( \mathbf{u} \), the right-hand side of the system, chosen to remove reverberations or sharpen the wavelet. The result is returned in vector \( \mathbf{y} \).

<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>
</tbody>
</table>
### Table 238: Data Types. (continued)

<table>
<thead>
<tr>
<th>$x$, $u$, $y$</th>
<th>$aux$</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<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

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>`CALL SWLEV</td>
</tr>
<tr>
<td>C and C++</td>
<td>`swlev</td>
</tr>
</tbody>
</table>

### On Entry

- **$x$**
  - is the vector $x$ of length $n$, containing the trace data for which the filter is to be designed.
  - For SWLEV and DWLEV, $x$ represents the first row (or the first column) of a positive definite or negative definite symmetric Toeplitz matrix, which is the autocorrelation matrix for which the filter is designed.
  - For CWLEV and ZWLEV, $x$ represents the first row of a positive definite or negative definite complex Hermitian Toeplitz matrix, which is the autocorrelation matrix for which the filter is designed.
  - Specified as: a one-dimensional array of (at least) length $1+(n-1)|incx|$, containing numbers of the data type indicated in Table 238 on page 1102.

- **$incx$**
  - is the stride for vector $x$.
  - Specified as: an integer; $incx > 0$.

- **$u$**
  - is the vector $u$ of length $n$, containing the right-hand side of the system to be solved.
  - Specified as: a one-dimensional array of (at least) length $1+(n-1)|incu|$, containing numbers of the data type indicated in Table 238 on page 1102.

- **$incu$**
  - is the stride for vector $u$.
  - Specified as: an integer. It can have any value.

- **$y$**
  - See On Return.

- **$incy$**
  - is the stride for vector $y$.
  - Specified as: an integer; $incy > 0$ or $incy < 0$.

- **$n$**
  - is the number of elements in vectors $x$, $u$, and $y$.
  - 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 these subroutines.
  - Specified as: an area of storage of length $naux$, containing numbers of the data type indicated in Table 238 on page 1102.
**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, SWLEV, DWLEV, CWLEV, and ZWLEV dynamically allocate the work area used by the subroutine. The work area is deallocated before control is returned to the calling program.

Otherwise, naux ≥ 3n.

You cannot use dynamic allocation if you need the information returned in AUX(1).

**On Return**

**y**

is the vector y of length n, containing the solution vector—that is, the coefficients of the n-point Wiener-Levinson filter. Returned as: a one-dimensional array of (at least) length 1+(n-1)|incy|, containing numbers of the data type indicated in Table 238 on page 1102.

**aux**

is the storage work area used by these subroutines, where if naux ≠ 0:

If AUX(1) = 0.0, the input Toeplitz matrix is positive definite or negative definite.

If AUX(1) > 0.0, the input Toeplitz matrix is indefinite (that is, it is not positive definite and it is not negative definite). The value returned in AUX(1) is the order of the first submatrix of A that is indefinite. The subroutine continues processing. See reference [“75” on page 1257] for information about under what circumstances your solution vector y would be valid.

All other values in aux are overwritten and are not significant.

Returned as: an area of storage of length naux, containing numbers of the data type indicated in Table 238 on page 1102, where AUX(1) ≥ 0.0.

**Notes**

1. For a description of a positive definite or negative definite symmetric Toeplitz matrix, see “Positive Definite or Negative Definite Symmetric Toeplitz Matrix” on page 86.

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 87.

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 49.

**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.

- u is the vector specifying the right side of the system, chosen to remove reverberations or to sharpen the wavelet.

- y is the solution vector.
See reference [“75” on page 1257], [“36” on page 1255], and the IBM 3838 Array Processor Functional Characteristics.

If \( n \) is 0, no computation is performed. For SWLEV and CWLEV, intermediate results are accumulated in long precision.

SWLEV provides the same function as the IBM 3838 function WLEV, with restrictions removed. See the IBM 3838 Array Processor Functional Characteristics manual.

**Error conditions**

**Resource Errors**

Error 2015 is unrecoverable, \( naux = 0 \), and unable to allocate work area.

**Computational Errors**

None

**Input-Argument Errors**

1. \( n < 0 \)
2. \( incx \leq 0 \)
3. \( incy = 0 \)
4. Error 2015 is recoverable or \( naux \neq 0 \), and \( naux \) is too small—that is, less than the minimum required value specified in the syntax for this argument. Return code 1 is returned if error 2015 is recoverable.

**Examples**

**Example 1**

This example shows how to compute filter coefficients in vector \( y \) by solving the system \( Ay = u \). Matrix \( A \) is:

\[
\begin{bmatrix}
50.0 & -8.0 & 7.0 & -5.0 \\
-8.0 & 50.0 & -8.0 & 7.0 \\
7.0 & -8.0 & 50.0 & -8.0 \\
-5.0 & 7.0 & -8.0 & 50.0
\end{bmatrix}
\]

This input Toeplitz matrix is positive definite, as indicated by the zero value in \( AUX(1) \) on output.

Call Statement and Input:

```
CALL SWLEV( X , 1  , U , 1  , Y , 1  , 4 , AUX , 12 )
```

\( X = (50.0, -8.0, 7.0, -5.0) \)

\( U = (40.0, -10.0, 30.0, 20.0) \)

\( AUX = \) (not relevant)

Output:

\( Y = (0.7667, -0.0663, 0.5745, 0.5778) \)

\( AUX = (0.0, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots) \)

**Example 2**

This example shows how to compute filter coefficients in vector \( y \) by solving the system \( Ay = u \). Matrix \( A \) is:

\[
\begin{bmatrix}
10.0 & -8.0 & 7.0 & -5.0 \\
-8.0 & 10.0 & -8.0 & 7.0 \\
7.0 & -8.0 & 10.0 & -8.0
\end{bmatrix}
\]
This input Toeplitz matrix is not positive definite, as indicated by the zero value in AUX(1) on output.

Call Statement and Input:

\[
\begin{array}{|cccc|}
\hline
-5.0 & 7.0 & -8.0 & 10.0 \\
\hline
\end{array}
\]

\[
X = (10.0, -8.0, 7.0, -5.0)
\]
\[
U = (40.0, -10.0, 30.0, 20.0)
\]
\[
AUX = (\text{not relevant})
\]

Output:

\[
Y = (5.1111, 5.5555, 12.2222, 10.4444)
\]
\[
AUX = (0.0, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots)
\]

**Example 3**

This example shows a vector \(x\) with a stride greater than 1, a vector \(u\) with a negative stride, and a vector \(y\) with a stride of 1. It uses the same input Toeplitz matrix as in Example 2, which is not positive definite.

Call Statement and Input:

\[
X = (10.0, \ldots, -8.0, \ldots, 7.0, \ldots, -5.0)
\]
\[
U = (20.0, \ldots, 30.0, \ldots, -10.0, \ldots, 40.0)
\]
\[
AUX = (\text{not relevant})
\]

Output:

\[
Y = (5.1111, 5.5555, 12.2222, 10.4444)
\]
\[
AUX = (0.0, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots)
\]

**Example 4**

This example shows how to compute filter coefficients in vector \(y\) by solving the system \(Ay = u\). Matrix \(A\) is:

\[
\begin{bmatrix}
(10.0, 0.0) & (2.0, -3.0) & (-3.0, 1.0) & (1.0, 1.0) \\
(2.0, 3.0) & (10.0, 0.0) & (2.0, -3.0) & (-3.0, 1.0) \\
(-3.0, -1.0) & (2.0, 3.0) & (10.0, 0.0) & (2.0, -3.0) \\
(1.0, -1.0) & (-3.0, -1.0) & (2.0, 3.0) & (10.0, 0.0)
\end{bmatrix}
\]

This input complex Hermitian Toeplitz matrix is positive definite, as indicated by the zero value in AUX(1) on output.

Call Statement and Input:

\[
X = ((10.0, 0.0), (2.0, -3.0), (-3.0, 1.0), (1.0, 1.0))
\]
\[
U = ((8.0, 3.0), (21.0, -5.0), (67.0, -13.0), (72.0, 11.0))
\]
\[
AUX = (\text{not relevant})
\]
Output:

\[
\begin{align*}
Y &= ((1.0, 0.0), (3.0, 0.0), (5.0, 0.0), (7.0, 0.0)) \\
AUX &= ((0.0, 0.0), \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots)
\end{align*}
\]

Example 5

This example shows a vector \(x\) with a stride greater than 1, a vector \(u\) with a negative stride, and a vector \(y\) with a stride of 1. It uses the same input complex Hermitian Toeplitz matrix as in Example 4.

This input complex Hermitian Toeplitz matrix is positive definite, as indicated by the zero value in AUX(1) on output.

Call Statement and Input:

\[
\begin{array}{cccccccc}
X & \text{INCX} & U & \text{INCU} & Y & \text{INCY} & N & \text{AUX} & \text{NAUX} \\
\hline
\text{CALL} & \text{ZWLEV}(X, 2, U, -2, Y, 1, 4, AUX, 12)
\end{array}
\]

\[
\begin{align*}
X &= ((10.0, 0.0), \ldots, (2.0, -3.0), \ldots, (-3.0, 1.0), \ldots, (1.0, 1.0)) \\
U &= ((72.0, 11.0), \ldots, (67.0, -13.0), \ldots, (21.0, -5.0), \ldots, (8.0, 3.0), \ldots) \\
AUX &= \text{(not relevant)}
\end{align*}
\]

Output:

\[
\begin{align*}
Y &= ((1.0, 0.0), (3.0, 0.0), (5.0, 0.0), (7.0, 0.0)) \\
AUX &= ((0.0, 0.0), \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots)
\end{align*}
\]
Chapter 13. Sorting and searching subroutines

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 239: 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 1110</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 1111</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 1114</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 1117</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 1120</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 73.

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 \( iopt \) argument.

© Copyright IBM Corp. 2018
ISORT, SSORT, and DSORT (Sort the Elements of a Sequence)

**Purpose**

These subroutines sort the elements of sequence \( x \).

<table>
<thead>
<tr>
<th>Data Types</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>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL ISORT</td>
</tr>
<tr>
<td>C and C++</td>
<td>isort</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 240 on page 1110.

\( incx \)

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 240 on page 1110.

**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 [“91” on page 1258].

**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:

```plaintext
CALL ISORT( X, 2, 5 )
X = (2, . , -1, . , 5, . , 4, . , -2)
```

Output:

```plaintext
X = (-2, . , -1, . , 2, . , 4, . , 5)
```

**Example 2**
This example shows a sequence \( x \) with a negative stride.

Call Statement and Input:

```plaintext
CALL ISORT( X, -1, 5 )
X = (2, -1, 5, 4, -2)
```

Output:

```plaintext
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, \( INDX \). 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 241: Data Types.

<table>
<thead>
<tr>
<th>Data Types</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

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL ISORTX</td>
</tr>
<tr>
<td>C and C++</td>
<td>isortx</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\| elements, containing numbers of the data type indicated in Table 241 on page 1112.

- **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 ≥ 0.

- **indx**
  - See On Return.

On Return

- **x**
  - is the sequence x of length n, with its elements sorted into designated order in the array. Returned as: a one-dimensional array, containing numbers of the data type indicated in Table 241 on page 1112.

- **indx**
  - is the array, referred to as INDX, containing the n indices that indicate, for the elements in the sorted output sequence, the original positions of those elements in input sequence x.

  **Note:** It is important to remember that when you specify a negative stride, ESSL assumes that the order of the input and output sequence elements in the X array is reversed; however, the elements in INDX are not reversed. See “Function” on page 1112.
  - Returned as: a one-dimensional array of length n, containing integers; 1 ≤ (INDX elements) ≤ 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 [*91* on page 1258].

**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:**

```
X  INCX  N   INDX
|   |    |    |
CALL ISORTX( X , 2  , 5 , INDX )
```

\[ X = (2, \ldots, -1, \ldots, 5, \ldots, 1, \ldots, -2) \]

**Output:**

\[ X = (-2, \ldots, -1, \ldots, 1, \ldots, 2, \ldots, 5) \]

\[ \text{INDX} = (5, 2, 4, 1, 3) \]

**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, 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 \( \text{INDX}(4) = 5 \).

Call Statement and Input:

```
X   INCX  N   INDX
|    |    |    |
CALL ISORTX( X , -1  , 5 , INDX )
X        =  (2, -1, 5, 1, -2)
```

Output:

```
X        =  (5, 2, 1, -1, -2)
INDX     =  (1, 4, 2, 5, 3)
```

---

**ISORTS, SSORTS, and DSORTS (Sort the Elements of a Sequence Using a Stable Sort and Note the Original Element Positions)**

**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 \( \text{INDX} \).

**Note:** If you need a stable sort, then you should use these subroutines rather than ISORTX, SSORTX, or DSORTX.

**Table 242: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x, work )</td>
<td>ISORTS</td>
</tr>
<tr>
<td>Integer</td>
<td>SSORTS</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>DSORTS</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DSORTS</td>
</tr>
</tbody>
</table>

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL ISORTS</td>
</tr>
<tr>
<td>C and C++</td>
<td>i sorts</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 \( 1+(n-1)|\text{incx}| \) elements, containing numbers of the data type indicated in Table 242 on page 1114.
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 242 on page 1114.

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 \( x \) 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 242 on page 1114.

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 1115.
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, \((x_n, x_{n-1}, \ldots, x_1)\), 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:
\[
\text{INDX}(k) = j \quad \text{for} \ j = 1, n \text{ and } k = 1, n
\]
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}, ..., x_1)

The sequence elements of x are stored in your output array by ESSL as follows:

X = (xx_n, xx_{n-1}, ..., xx_1)

where the elements xx_k are the elements x_j, sorted into descending order in X. As an example of how INDX is calculated, if xx_1 = x_{n-1}, then INDX(1) = n-1.

If n is 0, no computation is performed. See references [“38” on page 1255] and [“91” on page 1258].

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 INDX(2) = 2 and INDX(3) = 4.

Call Statement and Input:

```
X  INCX  N   INDX   WORK  LWORK
|    |    |    |      |      |
CALL ISORTS( X , 2  , 5 , INDX , WORK ,  5  )
X        =  (2, . , -1, . , 5, . , -1, . , -2)
```

Output:

```
X        =  (-2, . , -1, . , -1, . , 2, . , 5)
INDX     =  (5, 2, 4, 1, 3)
```

Example 2

This example shows how to sort a sequence x into descending order by specifying a negative stride. Therefore, both the input and output sequences are assumed to be reversed in the array X. The input sequence is assumed to be stored as follows:

X = (x_5, x_4, x_3, x_2, x_1) = (2, -1, 5, -1, -2)

The output sequence is stored by ESSL as follows:

X = (xx_5, xx_4, xx_3, xx_2, xx_1) = (5, 2, -1, -1, -2)

As a result, INDX is defined as follows:

INDX = (indx_1, indx_2, indx_3, indx_4, indx_5) = (1, 2, 4, 5, 3)

For example, because output sequence element xx_4 = 2 is input sequence element x_5, then INDX(4) = 5. Also, because this is a stable sort, the -1 elements remain in the same relative order in the output sequence, indicated by INDX(2) = 2 and INDX(3) = 4.

Call Statement and Input:

```
X  INCX  N   INDX   WORK  LWORK
|    |    |    |      |      |
CALL ISORTS( X , -1  , 5 , INDX , WORK ,  5  )
```
**X** = (2, -1, 5, -1, -2)

Output:

**X** = (5, 2, -1, -1, -2)
**INDX** = (1, 2, 4, 5, 3)

**IBSRCH, SBSRCH, and DBSRCH (Binary Search for Elements of a Sequence X in a Sorted Sequence Y)**

**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 INDX, and, optionally, return codes indicating whether the exact elements were found are returned in array RC.

**Table 243: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>x, y</td>
<td></td>
</tr>
<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>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL IBSRCH</td>
</tr>
<tr>
<td>C and C++</td>
<td>ibsrch</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 243 on page 1117. It must have at least 1+(n-1)|incx| elements.

- **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 INDX and RC.
  Specified as: an integer; n ≥ 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 \(1+(m-1)|\text{incy}|\), containing numbers of the data type indicated in Table 243 on page 1117.

\textit{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; \(m \geq 0\).

\textit{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**

\textit{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 \(y\). The first occurrence of the element found in sequence \(y\) is indicated in array \(\text{INDX}\). 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 \(\text{INDX}\).

Returned as: a one-dimensional array of length \(n\), containing integers; \(1 \leq (\text{INDX} \text{ elements}) \leq m+1\).

\(rc\)

has the following meaning, where:

If \(iopt = 0\), then \(rc\) is not used, and its contents remain unchanged.

If \(iopt = 1\), it is the array, referred to as \(\text{RC}\), containing the \(n\) return codes that indicate whether the elements in sequence \(x\) were found in sequence \(y\). For \(i = 1, n\), elements \(\text{RC}(i) = 0\) if \(x_i\) matches an element in sequence \(y\), and \(\text{RC}(i) = 1\) if an exact match is not found in sequence \(y\).

Returned as: a one-dimensional array of length \(n\), containing integers; \(\text{RC}(i) = 0\) 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 1110.

2. If you do not need the results provided in array \(\text{RC}\) 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:

- \(\text{INDX}\), 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 \(\text{INDX}\).
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 \( \text{INDX} \) and \( \text{RC} \) arrays are expressed as follows:

\[
\text{For } i = 1, n \\
\text{for all } y_j \geq x_i, j = 1, m, \text{INDX}(i) = \min(j) \\
\text{if all } y_j < x_i, j = 1, m, \text{INDX}(i) = m+1 \\
\text{And for } i = 1, n \\
\text{if } x_i = y_{\text{INDX}(i)}, \text{RC}(i) = 0 \\
\text{if } x_i \neq y_{\text{INDX}(i)}, \text{RC}(i) = 1
\]

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 [“91” on page 1258]. 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 1110.

### Error conditions

#### Computational Errors
None

#### Input-Argument Errors

1. \( n < 0 \)
2. \( m < 0 \)
3. \( iopt \neq 0 \) 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:**

```
CALL IBSRCH( X , 2  , 5 , Y , 1  , 10 , INDX , RC , 1 )
```

\( X \) = \((-3, . , 125, . , 30, . , 20, . , 70)\)
\( Y \) = \((10, 20, 30, 30, 40, 50, 60, 80, 90, 100)\)

**Output:**

\( \text{INDX} = (1, 11, 3, 2, 8) \)
\( \text{RC} = (1, 1, 0, 0, 1) \)

#### 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.
Example 3
This example shows a search where sequence \( x \) has a negative stride, and sequence \( y \) has a positive stride. The optional return codes are not requested, because \( IOPT \) is specified as 0.

Call Statement and Input:

```
X  INCX  N   Y  INCY  M   INDX   RC  IOPT
|   |    |   |   |    |     |     |    |
CALL IBSRCH( X , -2 , 5 , Y , -1 , 10 , INDX , RC , 0 )
```

\( X = (-3, . , 125, . , 30, . , 20, . , 70) \)
\( Y = (100, 90, 80, 60, 50, 40, 30, 30, 20, 10) \)

Output:

```
INDX     =  (8, 2, 3, 11, 1)
RC       =(not relevant)
```

Example 4
This example shows a search where sequence \( x \) has a positive stride, and sequence \( y \) has a negative stride. As shown below, elements of \( y \) are in descending order in array \( Y \). The optional return codes are not requested, because \( IOPT \) is specified as 0.

Call Statement and Input:

```
X  INCX  N   Y  INCY  M   INDX   RC  IOPT
|   |    |   |    |    |     |     |    |
CALL IBSRCH( X , 2 , 5 , Y , -1 , 10 , INDX , RC , 0 )
```

\( X = (-3, . , 125, . , 30, . , 20, . , 70) \)
\( Y = (100, 90, 80, 60, 50, 40, 30, 30, 20, 10) \)
\( RC = (not relevant) \)

Output:

```
INDX     =  (1, 11, 3, 2, 8)
```

ISSRCH, SSSRCH, and DSSRCH (Sequential Search for Elements of a Sequence \( x \) in the Sequence \( y \))

**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>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x, y )</td>
<td>ISSRCH</td>
</tr>
<tr>
<td>Integer</td>
<td>SSSRCH</td>
</tr>
</tbody>
</table>

**Table 244: Data Types.**
Table 244: Data Types. (continued)

<table>
<thead>
<tr>
<th>Data Types</th>
<th>DSSRCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long-precision real</td>
<td></td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL ISSRCH</td>
</tr>
<tr>
<td>C and C++</td>
<td>issrch</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 of (at least) length 1+(n-1)|incx|, containing numbers of the data type indicated in Table 244 on page 1120.

- **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 244 on page 1120.

- **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 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 [“91” on page 1258]. 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  INCX  N   Y  INCY  M   IDIR  INDEX
CALL ISSRCH( X , 1 , 3 , Y , 2 , 8 , 1 , INDEX )
```

\[
\begin{align*}
X & = (0, 12, 3) \\
Y & = (0, \ldots, 8, \ldots, 12, \ldots, 0, \ldots, 1, \ldots, 4, \ldots, 0, \ldots, 2)
\end{align*}
\]
Example 2
This example shows a search where sequences $x$ and $y$ have positive strides, and the search direction indicator, \textit{idir}, is negative.

Call Statement and Input:

\begin{verbatim}
CALL ISSRCH(X, 2, 3, Y, 2, 8, -1, INDX)
\end{verbatim}

Output:

\begin{verbatim}
INDX = (1, 3, 0)
\end{verbatim}

Example 3
This example shows a search where sequences $x$ and $y$ have negative strides, and the search direction indicator, \textit{idir}, is positive.

Call Statement and Input:

\begin{verbatim}
CALL ISSRCH(X, -1, 3, Y, -2, 8, 1, INDX)
\end{verbatim}

Output:

\begin{verbatim}
INDX = (7, 3, 0)
\end{verbatim}

Example 4
This example shows a search where sequences $x$ and $y$ have negative strides, and the search direction indicator, \textit{idir}, is negative.

Call Statement and Input:

\begin{verbatim}
CALL ISSRCH(X, -2, 3, Y, -1, 8, -1, INDX)
\end{verbatim}

Output:

\begin{verbatim}
INDX = (0, 6, 8)
\end{verbatim}
Chapter 14. Interpolation subroutines

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 245 on page 1125).

Table 245: 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 1126</td>
</tr>
<tr>
<td>Local Polynomial Interpolation</td>
<td>STPINT</td>
<td>DTPINT</td>
<td>“STPINT and DTPINT (Local Polynomial Interpolation)” on page 1130</td>
</tr>
<tr>
<td>Cubic Spline Interpolation</td>
<td>SCSINT</td>
<td>DCSINT</td>
<td>“SCSINT and DCSINT (Cubic Spline Interpolation)” on page 1133</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 1137</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 61.

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>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>x, y, c, t, s</td>
<td>SPINT</td>
</tr>
<tr>
<td>Short-precision real</td>
<td></td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DPINT</td>
</tr>
</tbody>
</table>

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SPINT</td>
</tr>
<tr>
<td>C and C++</td>
<td>spint</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.

  Specified as: a one-dimensional array of (at least) length n, containing numbers of the data type indicated in Table 246 on page 1126.

- **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 246 on page 1126.

- **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 c 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_j for j = 1, ninit, for the interpolating polynomial through the data points (x_j, y_j) for j = 1, ninit. If ninit < n, the values of c_j for j = ninit+1, n are undefined.

  Specified as: a one-dimensional array of (at least) length n, containing numbers of the data type indicated in Table 246 on page 1126.

- **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_j, y_j) for j = 1, ninit, where:
• If \( n_{\text{init}} = n \), all the Newton divided difference coefficients in \( c \) were computed for the data points. No additional coefficients are computed on this entry.

• If \( n_{\text{init}} < n \), the first \( n_{\text{init}} \) Newton divided difference coefficients in \( c \) were computed for the data points \((x_j, y_j)\) for \( j = 1, n_{\text{init}} \). The coefficients are updated for the additional data points \((x_j, y_j)\) for \( j = n_{\text{init}} + 1, n \) on this entry.

Specified as: an integer; \( n_{\text{init}} \leq n \).

\( t \)

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 246 on page 1126.

\( s \)

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 246 on page 1126.

\( n_{\text{init}} \)

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 \( n_{\text{init}} \).) Returned as: an integer; \( n_{\text{init}} = n \).

\( s \)

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 246 on page 1126.

**Notes**

1. In your C program, argument \( n_{\text{init}} \) 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:

\( x_j \) are elements of vector \( x \).

\( y_j \) are elements of vector \( y \).

The interpolated value at each \( t_i \) is returned in \( s_i \) for \( i = 1, m \). See references ["23" on page 1254] and ["65" on page 1256]. 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 \textit{ninit} and \textit{m} indicate which combination of functions are performed by this subroutine: computing the coefficients, performing the interpolation, or both. If \textit{m} = 0, only the divided difference coefficients are computed. No interpolation is performed. If \textit{n} = 0, no computation or interpolation is performed.

For SPINT, the Newton divided differences and interpolating values are accumulated in long precision.

\textbf{Newton Divided Differences and Interpolating Values:}

The Newton divided differences of the following data points:

\[(x_j, y_j) \text{ for } j = 1, n\]

where \(x_j \neq x_l\) if \(j \neq l\) for \(j, l = 1, n\)

are denoted by \(\delta_k y_j\) for \(k = 0, 1, 2, ..., n-1\) and \(j = 1, 2, ..., n-k\), and are defined as follows:

For \(k = 0\) and \(1:\)

\[\delta_0 y_j = y_j \text{ for } j = 1, 2, ..., n\]

\[\delta_1 y_j = (y_{j+1} - y_j) / (x_{j+1} - x_j) \text{ for } j = 1, 2, ..., n-1\]

For \(k = 2, 3, ..., n-1:\)

\[\delta_k y_j = (\delta_{k-1} y_{j+1} - \delta_{k-1} y_j) / (x_{j+k} - x_j) \text{ for } j = 1, 2, ..., n-k\]

The value \(s\) of the Newton divided difference form of the interpolating polynomial evaluated at an abscissa \(t\) is given by:

\[s = y_n + (t-x_n) \delta_1 y_{n-1} + (t-x_n) (t-x_n) \delta_2 y_{n-2} + \ldots + (t-x_2) (t-x_3) \ldots (t-x_n) \delta_{n-1} y_1\]

Therefore, on output, the coefficients in vector \(c\) are as follows:

\[c_n = y_n\]

\[c_{n-1} = \delta_1 y_{n-1}\]

\[c_{n-2} = \delta_2 y_{n-2}\]

\[\ldots\]

\[c_1 = \delta_{n-1} y_1\]

Also, the interpolating values in \(s\), in terms of \(c\), are as follows for \(i = 1, m:\)

\[s_i = c_n + (t_i-x_n) c_{n-1} + (t_i-x_{n-1}) (t_i-x_n) c_{n-2} + \ldots + (t_i-x_2) (t_i-x_3) \ldots (t_i-x_n) c_1\]

\textbf{Error conditions}

\textbf{Computational Errors}

None

\textbf{Input-Argument Errors}

1. \(n < 0\)
2. \(ninit > n\)
3. \(m < 0\)
Examples

Example 1

This example shows a quadratic polynomial interpolation on the initial call with the specified data points; that is, NINIT = 0, and C contains all undefined values. On output, NINIT and C are updated with new values.

Call Statement and Input:

```
X  Y  N  C  NINIT  T  S  M
|   |   |   |    |   |   |
CALL SPINT( X, Y, 3, C, 0, T, S, 2 )
```

```
X = (-0.50, 0.00, 1.00)
Y = (0.25, 0.00, 1.00)
C = (., ., .)
T = (-0.2, 0.2)
```

Output:

```
C = (1.00, 1.00, 1.00)
NINIT = 3
S = (0.04, 0.04)
```

Example 2

This example shows a quadratic polynomial interpolation on a subsequent call with the same data points specified in Example 1, but using a different set of abscissas in T. In this case, NINIT = N = 3, and C contains the values defined on output in Example 1. On output here, the values in NINIT and C are unchanged.

Call Statement and Input:

```
X  Y  N  C  NINIT  T  S  M
|   |   |   |    |   |   |
CALL SPINT( X, Y, 3, C, 3, T, S, 2 )
```

```
X = (-0.50, 0.00, 1.00)
Y = (0.25, 0.00, 1.00)
C = (1.00, 1.00, 1.00)
T = (-0.10, 0.10)
```

Output:

```
C = (1.00, 1.00, 1.00)
NINIT = 3
S = (0.01, 0.01)
```

Example 3

This example is the same as Example 2 except that it specifies additional data points on the subsequent call to the subroutine. In this case, 0 < NINIT < N. On output here, the values in NINIT and C are updated. The interpolating polynomial is a degree of 4.

Call Statement and Input:

```
X  Y  N  C  NINIT  T  S  M
|   |   |   |    |   |   |
CALL SPINT( X, Y, 5, C, 3, T, S, 2 )
```

```
X = (-0.50, 0.00, 1.00, -1.00, 0.50)
Y = (0.25, 0.00, 1.00, 1.10, 0.26)
C = (1.00, 1.00, 1.00, ., .)
T = (-0.10, 0.10)
```
Output:

\[
\begin{align*}
C &= (0.04, -0.06, 1.02, -0.56, 0.26) \\
NINIT &= 5 \\
S &= (0.0072, 0.0130)
\end{align*}
\]

**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 247: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</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**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</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 247 on page 1130.

\(y\)  
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 247 on page 1130.

\(n\)  
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 247 on page 1130.

\(s\)  
See On Return.
\( m \) is the number of elements in vectors \( \mathbf{t} \) and \( \mathbf{s} \)—that is, the number of interpolations to be performed. Specified as: an integer; \( m \geq 0 \).

\( \text{aux} \) 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 this subroutine. Its size is specified by \( \text{naux} \).

Specified as: an area of storage, containing numbers of the data type indicated in Table 247 on page 1130. 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, \( \text{STPINT} \) and \( \text{DTPINT} \) 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 \( \text{STPINT} \)

For 32-bit integer arguments

\[ \text{naux} \geq nint + m \]

For 64-bit integer arguments

\[ \text{naux} \geq nint + 2m \]

For \( \text{DTPINT} \)

\[ \text{naux} \geq nint + m \]

On Return

\( \mathbf{s} \) is the vector \( \mathbf{s} \) 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 247 on page 1130.

Notes

1. Vectors \( \mathbf{x}, \mathbf{y}, \) and \( \mathbf{t} \) must have no common elements with vector \( \mathbf{s} \) or work area \( \text{aux} \); otherwise, results are unpredictable. See “Vector concepts” on page 71.

2. The elements of vector \( \mathbf{x} \) 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 1110.

3. The elements of vector \( \mathbf{t} \) 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 49.

Function

Polynomial interpolation is performed at specified abscissas, \( t_i \) for \( i = 1, m \), in vector \( \mathbf{t} \), using \( nint \) points selected from the following data:

\[(x_j, y_j) \quad \text{for} \quad j = 1, n\]

where:
\( x_1 < x_2 < x_3 < \ldots < x_n \)

\( x_j \) are elements of vector \( x \).

\( y_j \) are elements of vector \( y \).

The points \((x_j, y_j)\), used in the interpolation at a given abscissa \( t_i \), are chosen as follows, where \( k = nint/2 \):

For \( t_i \leq x_{k+1} \), the first \( nint \) points are used.

For \( t_i > x_{n-nint+k} \), the last \( nint \) points are used.

Otherwise, points \( h \) through \( h+nint-1 \) are used, where:

\[ x_{h+k-1} < t_i \leq x_{h+k} \]

The interpolated value at each \( t_i \) is returned in \( s_i \) for \( i = 1, m \). See references ["23" on page 1254] and ["65" on page 1256]. If \( n, nint, \) or \( m \) is 0, no computation is performed. For a definition of the polynomial interpolation function performed through a set of data points, see “Function” on page 1127.

For STPINT, the Newton divided differences and interpolating values are accumulated in long precision.

**Error conditions**

**Resource Errors**

Error 2015 is unrecoverable, \( naux = 0 \), and unable to allocate work area.

**Computational Errors**

None

**Input-Argument Errors**

1. \( n < 0 \)
2. \( nint < 0 \) or \( nint > n \)
3. \( m < 0 \)
4. Error 2015 is recoverable or \( naux\neq0 \), 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 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  )
```

\( X \) = \( (0.0, 0.4, 1.0, 1.5, 2.1, 2.6, 3.0, 3.4, 3.9, 4.3) \)

\( Y \) = \( (1.0, 2.0, 3.0, 4.0, 5.0, 5.0, 4.0, 3.0, 2.0, 1.0) \)

\( T \) = \( (-1.0, 0.1, 1.1, 1.2, 3.9) \)

**Output:**

\( S \) = \( (-1.5000, 1.2500, 3.2000, 3.4000, 2.0000) \)

**Example 2**

This example shows interpolation using three data points—that is, quadratic interpolation—at each \( t_i \) value.

**Call Statement and Input:**

```
CALL STPINT( X , Y , 10 ,  3  , T , S , 5 , AUX , 8  )
```

---

1132 IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and Reference
Output:
\[
S = (-2.6667, 1.2750, 3.2121, 3.4182, 2.0000)
\]

**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 248: Data Types.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Types</strong></td>
</tr>
<tr>
<td>(x, y, C, t, s)</td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
</tbody>
</table>

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SCSINT</td>
</tr>
<tr>
<td>C and C++</td>
<td>scsint</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 248 on page 1133.

\(y\)

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 248 on page 1133.

\(c\)

is the matrix \(C\) with elements \(c_{jk}\) for \(j = 1, n\) and \(k = 1, 4\) that contain the following:

- If \(init \leq 0\), all elements of \(c\) are undefined on entry.
- If \(init = 1\), \(c_{11}\) contains the spline derivative at \(x_1\).
- If \(init = 2\), \(c_{21}\) contains the spline derivative at \(x_n\).
- If \(init = 3\), \(c_{11}\) contains the spline derivative at \(x_1\), and \(c_{21}\) contains the spline derivative at \(x_n\).
- If \(init > 3\), \(c\) contains the coefficients of the spline computed for the data points \((x_j, y_j)\) for \(j = 1, n\) on a previous call to this subroutine.

Specified as: an \(n\) by (at least) 4 array, containing numbers of the data type indicated in Table 248 on page 1133.
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; \( n \geq 0 \).

\( \text{init} \) 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 \( \text{init} \leq 0 \), 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 248 on page 1133.

\( s \) 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 248 on page 1133.

\( \text{init} \) 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 248 on page 1133.

**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 1110.
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:

$$(x_j, y_j) \quad \text{for } j = 1, n$$

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 [“23” on page 1254] and [“65” on page 1256]. The coefficients of the spline, $c_{jk}$ for $j = 1, n$ and $k = 1, 4$, are returned in matrix $C$. These coefficients can then be reused on subsequent calls to this subroutine, using the same data points $(x_j, y_j)$, but with new values of $t_i$. The cubic spline values returned in $s$ are computed using the coefficients as follows:

$$s_i = c_{j1} + c_{j2} (x_j - t_i) + c_{j3} (x_j - t_i)^2 + c_{j4} (x_j - t_i)^3 \quad \text{for } i = 1, m$$

where:

$j = 1$ for $t_i \leq x_1$

$j = k$ for $x_1 < t_i \leq x_n$, such that $x_{k-1} < t_i \leq x_k$

$j = n$ for $x_n < t_i$

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:

```
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:

```
CALL SCSINT( X , Y , C , 6 , 2 , T , S , 4 )
```

```
| .   .  .  . |
| 0.1  .  .  . |
```

C =

```
| .   .  .  . |
| .   .  .  . |
| .   .  .  . |
| .   .  .  . |
| .   .  .  . |
```

T = (-1.000, 2.500, 4.000, 7.000)

Output:

```
|  0.000  -0.865   0.000  -0.135 |
|  1.000  -1.270   0.405  -0.135 |
|  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   0.180   1.897  -0.797 |
```
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:

\[
\begin{array}{cccccccc}
X & Y & C & N & INIT & T & S & M \\
1.000 & 2.000 & 3.000 & 4.000 & 5.000 & 6.000 & & \\
0.000 & 1.000 & 2.000 & 1.100 & 0.000 & -1.000 & & \\
\end{array}
\]

\[
C = \begin{pmatrix}
1.000 & -1.770 & -0.460 & 1.230 \\
2.000 & 0.079 & -1.389 & 0.310 \\
1.100 & 1.152 & 0.316 & -0.568 \\
0.000 & 1.312 & -0.476 & 0.264 \\
-1.000 & -0.100 & 1.888 & -0.788 \\
\end{pmatrix}
\]

Output:

\[
C = \begin{pmatrix}
0.000 & 1.000 & 3.230 & 1.230 \\
1.000 & -1.770 & -0.460 & 1.230 \\
2.000 & 0.079 & -1.389 & 0.310 \\
1.100 & 1.152 & 0.316 & -0.568 \\
0.000 & 1.312 & -0.476 & 0.264 \\
-1.000 & -0.100 & 1.888 & -0.788 \\
\end{pmatrix}
\]

Example 4

This example evaluates the spline at a set of points, using the coefficients obtained in Example 3.

Call Statement and Input:

\[
\begin{array}{cccccccc}
X & Y & C & N & INIT & T & S & M \\
1.000 & 2.000 & 3.000 & 4.000 & 5.000 & 6.000 & & \\
0.000 & 1.000 & 2.000 & 1.100 & 0.000 & -1.000 & & \\
\end{array}
\]

\[
C \text{ (same as output } C \text{ in Example 3) }
\]

\[
T = (-1.000, 2.500, 4.000, 7.000)
\]

Output:

\[
C \text{ (same as output } C \text{ in Example 3) }
\]

\[
S = (24.762, 1.731, 1.100, 1.776)
\]

\[
INIT = 4
\]

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 249: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x, y, z, t, u, aux, S )</td>
<td></td>
</tr>
<tr>
<td>Short-precision real</td>
<td>SCSIN2</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DCSIN2</td>
</tr>
</tbody>
</table>

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>`CALL SCSIN2</td>
</tr>
<tr>
<td>C and C++</td>
<td>`scsin2</td>
</tr>
</tbody>
</table>

**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 249 on page 1138.

- \( y \) 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 249 on page 1138.

- \( z \) is the matrix \( Z \), containing the data at \( (x_i, y_j) \) for \( i = 1, n1 \) and \( j = 1, n2 \) that defines the spline.
  
  Specified as: an \( ldz \) by (at least) \( n2 \) array, containing numbers of the data type indicated in Table 249 on page 1138.

- \( n1 \) 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; \( n1 \geq 0 \).

- \( n2 \) is the number of elements in vector \( y \) and the number of columns in matrix \( Z \)—that is, the number of y-coordinates at which the spline is defined.
  
  Specified as: an integer; \( n2 \geq 0 \).

- \( ldz \) is the leading dimension of the array specified for \( z \).
  
  Specified as: an integer; \( ldz > 0 \) and \( ldz \geq n1 \).

- \( t \) is the vector \( t \) of length \( m1 \), containing the x-coordinates at which the spline is evaluated.
  
  Specified as: a one-dimensional array of (at least) length \( m1 \), containing numbers of the data type indicated in Table 249 on page 1138.

- \( u \) is the vector \( u \) of length \( m2 \), containing the y-coordinates at which the spline is evaluated.
Specified as: a one-dimensional array of (at least) length $m_2$, containing numbers of the data type indicated in Table 249 on page 1138.

$m_1$

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 247 on page 1130. 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 249 on page 1138.

Notes

1. The cyclic reduction method used to solve the equations in this subroutine can generate underflows on well-scaled problems. This does not affect accuracy, but it may decrease performance. For this reason, you may want to disable underflow before calling this subroutine.

2. Vectors $x$, $y$, $t$, and $u$, matrix $Z$, and the $aux$ work area must have no common elements with matrix $S$; otherwise, results are unpredictable.

3. The elements within vectors $x$ and $y$ must be distinct. In addition, the elements in the vectors must be sorted into ascending order; that is, $x_1 < x_2 < \ldots < x_{n_1}$ and $y_1 < y_2 < \ldots < y_{n_2}$. Otherwise, results are
unpredictable. For details on how to do this, see “ISORT, SSORT, and DSORT (Sort the Elements of a Sequence)” on page 1110.

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 49.

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:
\[S_{kh} = S_{t_k}(u_h) \quad \text{for } l = 1, m2\]
See references [“65” on page 1256] and [“73” on page 1257]. 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:

```
CALL SCSIN2( X , Y , Z , 6 , 5 , 6 , T , U , 4 , 3 , S , 4 , AUX , 90 )
```

X        =  (0.0, 0.2, 0.3, 0.4, 0.5, 0.7)
Y        =  (0.0, 0.2, 0.3, 0.4, 0.6)

| 0.000  0.008  0.027  0.064  0.216 |
| 0.008  0.016  0.035  0.072  0.224 |
| 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  0.376  0.407  0.559 |

T        =  (0.10, 0.15, 0.25, 0.35)
U        =  (0.05, 0.25, 0.45)

Output:

S        =  | 0.001  0.017  0.095 |
            | 0.003  0.019  0.097 |
            | 0.016  0.031  0.110 |
            | 0.043  0.059  0.137 |

Interpolation subroutines 1141
Chapter 15. Numerical quadrature subroutines

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 250: 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 1145</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 1147</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 1149</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 1155</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 1158</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 1161</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 [“34” on page 1254], [“74” on page 1257], and [“113” on page 1259].

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 1151.
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 61.

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, ti, where ti 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: yi = f(ti) 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, si, where si 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, ti, where ti 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: zij = f(si, tj) 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 1.
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 125
• “Setting Up a User-Supplied Subroutine for ESSL in C” on page 144
• “Setting Up a User-Supplied Subroutine for ESSL in C++” on page 157

**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>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x, y, xyint, eest)</td>
<td>SPTNQ</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>SPTNQ</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DPTNQ</td>
</tr>
</tbody>
</table>

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SPTNQ</td>
</tr>
<tr>
<td>C and C++</td>
<td>sptnq</td>
</tr>
</tbody>
</table>

**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 251 on page 1145.

\(y\)

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 251 on page 1145.

\(n\)

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 1146.

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 251 on page 1145.

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 ["34" on page 1254] and ["74" on page 1257].

Returned as: a number of the data type indicated in Table 251 on page 1145.

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 1110.

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 ["74" on page 1257] and ["34" on page 1254].

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:

```
CALL SPTNQ( X , Y , 10 , XYINT , EEST )
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
<td>N</td>
<td>XYINT</td>
<td>EEST</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.4</td>
<td>1.0</td>
<td>1.5</td>
<td>2.1</td>
</tr>
<tr>
<td>1.0</td>
<td>2.0</td>
<td>3.0</td>
<td>4.0</td>
<td>5.0</td>
</tr>
</tbody>
</table>

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:

```
CALL SPTNQ( X , Y , 10 , XYINT , EEST )
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
<td>N</td>
<td>XYINT</td>
<td>EEST</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3</td>
<td>3.9</td>
<td>3.4</td>
<td>3.0</td>
<td>2.6</td>
</tr>
<tr>
<td>3.3</td>
<td>3.5</td>
<td>3.0</td>
<td>4.0</td>
<td>4.5</td>
</tr>
</tbody>
</table>

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 252: Data Types.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Types</strong></td>
</tr>
<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>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>SGLNQ</td>
</tr>
<tr>
<td>C and C++</td>
<td>sglnq</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 1144.

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 252 on page 1147.

\( b \)

is the upper limit of integration, \( b \).

Specified as: a number of the data type indicated in Table 252 on page 1147.

\( n \)

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 252 on page 1147.

**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 1148, 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 1144.

**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 [“34” on page 1254] and [“113” on page 1259]. 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:

$$\int_{0.0}^{2.0} \left(x^2 + e^x\right) dx$$

The user-supplied subroutine **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>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a, b, c, d, Z, Result$</td>
<td>$\text{SGLNQ2}$</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>$\text{SGLNQ2}$</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>$\text{DGLNQ2}$</td>
</tr>
</tbody>
</table>
Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>SGLNQ2</td>
</tr>
<tr>
<td>C and C++</td>
<td>sglqn2</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 1144.

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 253 on page 1149.

*b*

is the upper limit of integration, b, for the first variable integrated.

Specified as: a number of the data type indicated in Table 253 on page 1149.

*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 253 on page 1149.

*d*

is the upper limit of integration, d, for the second variable integrated.

Specified as: a number of the data type indicated in Table 253 on page 1149.

*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 253 on page 1149.

*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 253 on page 1149.
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 1144.

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:

\[(a, b) \text{ for } s \]
\[(c, d) \text{ for } t \]

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 1147 and references [“34” on page 1254] and [“113” on page 1259]. The result is returned as the function value.

Special Usage
To achieve optimal performance in this subroutine and in the functional evaluation, specify the first variable integrated in this subroutine as the variable having more points. The first variable integrated is the variable in the inner integral. For example, in the following integration, \(x\) is the first variable integrated:

\[ \int_{u1}^{u2} \int_{r1}^{r2} f(x, y) \, dx \, dy \]

This is the suggested order of integration if the \(x\) variable has more points than the \(y\) variable. On the other hand, if the \(y\) variable has more points, you make \(y\) the first variable integrated.

Because the order of integration does not matter to the resulting approximation, you may be able to reverse the order that \(x\) and \(y\) are integrated and get better performance. This can be expressed as:

\[ \int_{u1}^{u2} \int_{r1}^{r2} f(x, y) \, dx \, dy = \int_{r1}^{r2} \int_{u1}^{u2} f(x, y) \, dy \, dx \]

Results are mathematically equivalent. However, because the algorithm is computed in a different way, results may not be bitwise identical.

Table 254 on page 1152 shows how to assign your variables to the _GLNQ2 and subf arguments for the \(x-y\) integration shown on the left and for the \(y-x\) integration shown on the right. For examples of how to do each of these, see Example 1 and Example 2.
Table 254: 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>r₁</td>
<td>u₁</td>
</tr>
<tr>
<td>b</td>
<td>r₂</td>
<td>u₂</td>
</tr>
<tr>
<td>n₁</td>
<td>(order for x)</td>
<td>(order for y)</td>
</tr>
<tr>
<td>c</td>
<td>u₁</td>
<td>r₁</td>
</tr>
<tr>
<td>d</td>
<td>u₂</td>
<td>r₂</td>
</tr>
<tr>
<td>n₂</td>
<td>(order for y)</td>
<td>(order for x)</td>
</tr>
<tr>
<td>For 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>n₁</td>
<td>(order for x)</td>
<td>(order for y)</td>
</tr>
<tr>
<td>n₂</td>
<td>(order for y)</td>
<td>(order for x)</td>
</tr>
</tbody>
</table>

**Error conditions**

**Computational Errors**
- None

**Input-Argument Errors**
- 1. ldz ≤ 0
- 2. n₁ > ldz
- 3. n₁ or n₂ 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:

\[
 \int_{-2.0}^{2.0} \int_{0.0}^{-1.0} (e^x \sin y) \, dx \, dy
\]

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:

```fortran
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
```

1152 IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and Reference
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:

\[
\begin{align*}
\text{DO 1 I=1,N1} \\
1 & \quad \text{T1(I)=EXP(S(I))} \\
\text{DO 2 J=1,N2} \\
2 & \quad \text{T2(J)=SIN(T(J))} \\
\text{DO 3 J=1,N2} \\
3 & \quad \text{Z(I,J)=T1(I)*T2(J)} \\
\text{CONTINUE} \\
\end{align*}
\]

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:

```c
void fun1(s, n1, t, n2, z, ldz)
float *s, *t, *z;
int *n1, *n2, *ldz;
{
    int i, j;
    for(j = 0; j < *n2; ++j, z += *ldz)
    {
        for(i = 0; i < *n1; ++i)
            z[i] = exp(s[i]) * sin(t[j]);
    }
}
```

Using C++ for SUBF:

The user-supplied subroutine FUN1, which evaluates the integrand function, is coded in C++ as follows:

```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)
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:

$$\int_0^{20.0} \int_0^{1.0} (\cos x \sin y) \, dx \, dy = \int_0^{1.0} \int_0^{20.0} (\cos x \sin y) \, dy \, dx$$

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
   Z(I,J)=COS(T(J))*SIN(S(I))
1 CONTINUE
RETURN
END
```

**Note:** The same coding principles for achieving good performance that are noted in Example 1 also apply to this user-supplied subroutine FUN2.

Program Statements and Input:

```fortran
EXTERNAL FUN2.
.
.
.
SUBF    A      B    N1    C     D    N2   Z  LDZ
|      |      |    |     |     |    |    |   |
YXINT = SGLNQ2( FUN2 , 0.0 , 20.0 , 48 , 0.0 , 1.0 , 5 , Z , 48 )
|      |      |    |     |     |    |    |   |
.
.
FUN2   = (see above)
Z      = (not relevant)
```

Output:

```
YXINT   =  0.4981
```
SGLGQ and DGLGQ (Numerical Quadrature Performed on a Function Using Gauss-Laguerre Quadrature)

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 255: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>a, b, Result</td>
<td></td>
</tr>
<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>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>SGLGQ</td>
</tr>
<tr>
<td>C and C++</td>
<td>sglgq</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 1144.

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 255 on page 1155.

b

is the scaling constant \( b \) for the exponential.

Specified as: a number of the data type indicated in Table 255 on page 1155; \( b > 0 \) 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 255 on page 1155.

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 1156, 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 1144.

**Function**

The integral is approximated for a real valued function over a semi-infinite interval, using the Gauss-Laguerre Quadrature method of specified order. The region of integration is:

\[(a, \infty) \text{ if } b > 0\]
\[(-\infty, a) \text{ if } b < 0\]

The method of order \(n\) is theoretically exact for integrals of the following form, where \(f\) is a polynomial of degree less than \(2n\):

\[
\int_{a}^{\infty} f(x)e^{-bx} \, dx \quad \text{if } b > 0
\]
\[
\int_{-\infty}^{a} f(x)e^{-bx} \, dx \quad \text{if } b < 0
\]

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 ["34" on page 1254] and ["113" on page 1259]. 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 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
1156 IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and Reference
```
Y(I)=SIN(3.0*T(I))*EXP(-1.5*T(I))

RETURN
END

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 \texttt{FUN2}, which evaluates the integrand function, is coded in Fortran as follows:

```fortran
SUBROUTINE FUN2 (T, Y, N)
    INTEGER*4 N
    REAL*4 T(*), Y(*), TEMP
    DO 1 I=1,N
        1     Y(I)=SIN(3.0*T(I))*EXP(1.5*T(I))
    RETURN
END
```

Output:

XINT = -0.011
SGRAQ and DGRAQ (Numerical Quadrature Performed on a Function Using Gaussian-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>
<thead>
<tr>
<th>Table 256: Data Types.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Types</strong></td>
</tr>
<tr>
<td><strong>a, b, Result</strong></td>
</tr>
<tr>
<td>Short-precision real</td>
</tr>
<tr>
<td>Long-precision real</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>SGRAQ</td>
</tr>
<tr>
<td>C and C++</td>
<td>sgraq</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 1144.

Specified as: \( \text{subf} \) 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 256 on page 1158.

**b**

is the centering constant \( b \) for the integrand.

Specified as: a number of the data type indicated in Table 256 on page 1158.

**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, \text{ or } 256. 

On Return

**Function value**

is the approximation of the integral. Returned as: a number of the data type indicated in Table 256 on page 1158.

Notes

1. Declare the DGRAQ function in your program as returning a long-precision real number. Declare the SGRAQ function, if necessary, as returning a short-precision real number.
2. The subroutine specified for subf must be declared as external in your program. Also, data types used by subf must agree with the data types specified by this ESSL subroutine. The variable \( x \), described under “Function” on page 1159, 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 1144.

**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:

\[
(a, \infty) \quad \text{if} \quad a + b > 0
\]
\[
(-\infty, a) \quad \text{if} \quad a + b < 0
\]

The method of order \( n \) is theoretically exact for integrals of the following form, where \( f \) is a polynomial of degree less than \( 2n \):

\[
\int_a^\infty f \left( \frac{1}{x + b} \right) \frac{1}{(x + b)^2} \, dx \quad \text{if} \quad a + b > 0
\]

\[
\int_{-\infty}^a f \left( \frac{1}{x + b} \right) \frac{1}{(x + b)^2} \, dx \quad \text{if} \quad a + b < 0
\]

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 [“34” on page 1254] and [“113” on page 1259]. 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:

\[ f(x) = \frac{e^{1.0/x}}{x^2} \]

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)
    Y(I)=EXP(TEMP)*TEMP**2
  1
  RETURN
END
```

Program Statements and Input:

```
EXTERNAL FUN1

SUBF     A     B    N
|       |     |    |
XINT = SGRAQ( FUN1 , -2.0 , 0.0 , 10 )
```

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)
    Y(I)=TEMP**2+10.0*TEMP**11
  1
  RETURN
END
```

Program Statements and Input:

```
EXTERNAL FUN2

SUBF    A      B    N
|      |      |    |
```

1160 IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and Reference
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 257: Data Types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$, $b$, Result</td>
<td></td>
</tr>
<tr>
<td>Short-precision real</td>
<td>SGHMQ</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DGHMQ</td>
</tr>
</tbody>
</table>

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>SGHMQ</td>
</tr>
<tr>
<td>C and C++</td>
<td>sghmq</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 1144.

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 257 on page 1161.

$b$

is the scaling constant $b$ for the exponential.

Specified as: a number of the data type indicated in Table 257 on page 1161; $b > 0$.

$n$

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 257 on page 1161.

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 1162, 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 1144.

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 -∞ to ∞. 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 [“34” on page 1254] and [“113” on page 1259]. 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 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 subroutines

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
- Tausworthe exclusive-or algorithm

<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 1166</td>
</tr>
</tbody>
</table>

Table 258: List of Random Number Generation Initialization Subroutines.

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SURNG</td>
<td>“SURNG (Generate a Vector of Uniformly Distributed Pseudo-Random Numbers)” on page 1170</td>
</tr>
<tr>
<td>SNRNG</td>
<td>“SNRNG (Generate a Vector of Normally Distributed Pseudo-Random Numbers)” on page 1173</td>
</tr>
<tr>
<td>SURAND</td>
<td>“SURAND (Generate a Vector of Uniformly Distributed Random Numbers)” on page 1176</td>
</tr>
<tr>
<td>SNRAND</td>
<td>“SNRAND (Generate a Vector of Normally Distributed Random Numbers)” on page 1178</td>
</tr>
<tr>
<td>SURXOR§</td>
<td>“SURXOR (Generate a Vector of Long Period Uniformly Distributed Random Numbers)” on page 1181</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.

INITRNG (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 INITRNG with one or more calls to SURNG, DURNG, SNRNG or DNRNG.

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL INITRNG (iopt, irepeat, iseed, liseed, istate, listate)</td>
</tr>
<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 $iopt = 1$, a single-precision, SIMD-oriented Mersenne Twister pseudo-random number generator with a period of $2^{19937} - 1$ (SFMT19937) is used.
  - If $iopt = 2$, 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; $iopt = 1$ or 2.

- **irepeat** indicates whether repeatable or non-repeatable pseudo-random number sequences will be generated, where:
  - If $irepeat = 0$, the pseudo-random number generator uses values from $iseed$ to generate repeatable pseudo-random number sequences.
  - If $irepeat = 1$, the pseudo-random number generator uses hardware-generated values to generate non-repeatable pseudo-random number sequences.
  Specified as: an integer; $irepeat = 0$ or 1.

- **iseed** If $irepeat = 0$, $iseed$ 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 $irepeat = 1$, $iseed$ is ignored.
  Specified as: a one-dimensional integer array of (at least) length max(1, liseed).

- **liseed** is the number of elements in array ISEED, where:
  - If $irepeat = 0$, $liseed$ is determined as follows:
    - **32-bit integer environment**
      - If $iopt = 1$ or 2, $liseed ≥ 624$.  

**64-bit integer environment**

If \( iopt = 1 \) or \( 2 \), \( liseed \geq 312 \).

**Note:** If \( irepeat = 0 \) and insufficient seeds are provided, the seed values supplied in \( iseed \) 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".

**listate**

If \( listate \neq -1 \), \( listate \) 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 \( iopt = 1 \), \( listate \geq 696 \).
- If \( iopt = 2 \), \( listate \geq 839 \).

**64-bit integer environment**

- If \( iopt = 1 \), \( listate \geq 348 \).
- If \( iopt = 2 \), \( listate \geq 420 \).

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 [“98” on page 1258]) 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 “99” on page 1258]) with a period length equal to $2^{19937}-1$ of the produced sequence.

See references [“97” on page 1258], [“98” on page 1258] and [“99” on page 1258].

Error conditions

Computational Errors
None

Input-Argument Errors
1. \texttt{iopt} ≠ 1 or 2
2. \texttt{irepeat} = 0 and \texttt{liseed} < 1
3. In a 32-bit integer environment:
   • \texttt{iopt} = 1 and \texttt{listate} ≠ -1 and \texttt{listate} < 696.
   • \texttt{iopt} = 2 and \texttt{listate} ≠ -1 and \texttt{listate} < 839.
4. In a 64-bit integer environment:
   • \texttt{iopt} = 1 and \texttt{listate} ≠ -1 and \texttt{listate} < 348.
   • \texttt{iopt} = 2 and \texttt{listate} ≠ -1 and \texttt{listate} < 420.

Examples

Example 1
This example shows a call to INITRNG to find the optimal size of the \texttt{istate} array needed by the SFMT19937 pseudo-random number generator.

Call Statement and Input:

\begin{verbatim}
IOPT  IREPEAT  ISEED  LISEED  ISTATE  LISTATE
|      |        |        |        |      |      |
CALL INITRNG(  1 ,  IREPEAT ,  ISEED ,  LISEED ,  ISTATE , -1 )
\end{verbatim}

Output:

\begin{verbatim}
LISTATE  =  696 (in a 32-bit integer environment)
LISTATE  =  348 (in a 64-bit integer environment)
\end{verbatim}

Example 2
This example shows a call to INITRNG to find the optimal size of the \texttt{istate} array needed by the DSFMT19937 pseudo-random number generator.

Call Statement and Input:

\begin{verbatim}
IOPT  IREPEAT  ISEED  LISEED  ISTATE  LISTATE
|      |        |        |        |      |      |
CALL INITRNG(  2 ,  IREPEAT ,  ISEED ,  LISEED ,  ISTATE , -1 )
\end{verbatim}

Output:

\begin{verbatim}
LISTATE  =  839 (in a 32-bit integer environment)
LISTATE  =  420 (in a 64-bit integer environment)
\end{verbatim}
**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>1</td>
<td>0</td>
<td>ISEED</td>
<td>LISEED</td>
<td>ISTATE</td>
<td>1000</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 DSFMT19937 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>2</td>
<td>0</td>
<td>ISEED</td>
<td>LISEED</td>
<td>ISTATE</td>
<td>1000</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 DSFMT19937 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 DSFMT19937 pseudo-random number generator to generate non-repeatable random sequences in a subsequent call to DURNG or DNRNG.

Call Statement and Input:

```
IOPT  IREPEAT  ISEED  LISEED  ISTATE  LISTATE
   2          1     ISEED   LISEED   ISTATE   1000
```

Output:

istate contains 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 260: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x, a, b )</td>
<td>Short-precision real: SURNG</td>
</tr>
</tbody>
</table>

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SURNG</td>
</tr>
<tr>
<td>C and C++</td>
<td>surng</td>
</tr>
</tbody>
</table>

**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 260 on page 1170.

\( b \)

is the right boundary of the interval \([a, b]\).

Specified as: a number of the data type indicated in Table 260 on page 1170.
\( x \)

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 260 on page 1170.

\( istate \)

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 [“97” on page 1258], [“98” on page 1258] and [“99” on page 1258].

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:

```
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:

```
| 0.439785 |
| 0.064906 |
| 0.385660 |
| 0.695451 |
| 0.496463 |
| 0.154272 |
| 0.002247 |
| 0.725402 |
| 0.837238 |
| 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:

```
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:

| 0.948207 |
| 0.388311 |
| 0.758121 |
| 0.430842 |
| 0.261129 |
| 0.693552 |
| 0.113275 |
| 0.607048 |
| 0.192948 |
| 0.669879 |

X = 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 261: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x), (rmean), (sigma)</td>
<td>SNRNG</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>SNRNG</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DNRNG</td>
</tr>
</tbody>
</table>

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SNRNG</td>
</tr>
<tr>
<td>C and C++</td>
<td>SNRNG</td>
</tr>
</tbody>
</table>

**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 261 on page 1173.
**sigma**

is the standard deviation value of the distribution.

Specified as: a number of the data type indicated in Table 261 on page 1173.

**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* ≥ 696.
- If INITRNG was called with *iopt* = 2, *listate* ≥ 839.

**64-bit pointer environment**

- If INITRNG was called with *iopt* = 1, *listate* ≥ 348.
- If INITRNG was called with *iopt* = 2, *listate* ≥ 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 261 on page 1173.

**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 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.
Random number generation subroutines

Error conditions

Computational Errors
None

Input-Argument Errors

1. \( n < 0 \) or \( n \) is an odd number
2. \( \sigma \leq 0 \)
3. \( iState \) is not initialized (by a preceding call to INITRNG)
4. \( iState \) 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:

\[
\begin{array}{cccccc}
N & RMEAN & SIGMA & X & ISTATE & LISTATE \\
\hline
& & & \hline
\end{array}
\]

\[
\text{CALL SNRNG( 10 , 0.0 , 1.0 , X , ISTATE , 1000 )}
\]

\[
\text{ISTATE = (same as output ISTATE in Example 3)}
\]

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:

\[
X = \begin{bmatrix}
-0.426951 \\
0.988221 \\
0.929709 \\
-0.331744 \\
-0.965826 \\
0.662854 \\
0.866279 \\
-0.010326 \\
-0.172133 \\
0.215101 \\
\end{bmatrix}
\]

\[
\text{ISTATE = contains the updated state of the pseudo-random number generator.}
\]

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:

\[
\begin{array}{cccccc}
N & RMEAN & SIGMA & X & ISTATE & LISTATE \\
\hline
& & & \hline
\end{array}
\]

\[
\text{CALL DNRNG( 10 , 0.0 , 1.0 , X , ISTATE , 1000 )}
\]

\[
\text{ISTATE = contains the updated state of the pseudo-random number generator.}
\]
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:

```
| -0.426951 |
| -1.570857 |
| -1.858332 |
| -0.709286 |
| -1.528250 |
X    =  |  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>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$ Short-precision real</td>
<td>SURAND</td>
</tr>
<tr>
<td>$x$ Long-precision real</td>
<td>DURAND</td>
</tr>
<tr>
<td>$seed$ Short-precision real</td>
<td>SURAND</td>
</tr>
<tr>
<td>$seed$ 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>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SURAND</td>
</tr>
<tr>
<td>C and C++</td>
<td>surand</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 262 on page 1176. 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 $1.0 \leq seed < \left(2^{147483647.0} \approx 2^{31} - 1\right)$. 

1176 IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and Reference
**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 262 on page 1176. 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 262 on page 1176.

**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:

\[
s_i = (a(s_{i-1})) \mod(m) = (a's_0) \mod(m) \\
x_i = s_i/m \text{  for } i = 1, 2, ..., n
\]

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 [“92” on page 1258] and [“96” on page 1258]. 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:

<table>
<thead>
<tr>
<th>SEED</th>
<th>N</th>
<th>X</th>
</tr>
</thead>
</table>
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,
           0.4995853,
           0.4692344,
           0.1603608,
           0.1832564,
           0.9899062,
           0.3539868)
```

**Example 2**

This example shows a call to DURAND to generate 10 random numbers.

Call Statement and Input:

```
CALL DURAND( SEED , 10 , X )
SEED     =  80629.0
```

Output:

```
SEED     =  759150100.0
X        =  (0.6310323270182275,
           0.7603201953509451,
           0.7015232633340746,
           0.5014868557925740,
           0.4895853057920864,
           0.4602344475967038,
           0.1603607578018497,
           0.1832563756887132,
           0.9899062002030695,
           0.3539868129904134)
```

**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>
<thead>
<tr>
<th>Data Types</th>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x, aux )</td>
<td>( seed )</td>
</tr>
<tr>
<td>Short-precision real</td>
<td>Long-precision real</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>Long-precision real</td>
</tr>
</tbody>
</table>
Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL SNRAND</td>
</tr>
<tr>
<td>C and C++</td>
<td>snrand</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 263 on page 1178. 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 263 on page 1178. 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 263 on page 1178. It is a whole number whose value is $1.0 \leq seed < (2^{31} - 1)$.

`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 263 on page 1178.

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 \( n_{aux} \) dynamically returned to your program. For details, see “Using Auxiliary Storage in ESSL” on page 49.

**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 [“92” on page 1258].

1. Using seed, a vector of uniform (0,1) pseudo-random numbers, \( u_i \) for \( i = 1, n \), is generated by calling SURAND or DURAND, respectively. These \( u_i \) values are then used in the subsequent steps.

2. All \( (y_j, z_j) \) for \( j = 1, n/2 \) are set as follows, where each \( (y, z) \) is a point in the square -1 to 1:

\[
y_j = 2u_{2j-1} - 1 \\
z_j = 2u_{2j} - 1
\]

3. All \( p_j \) for \( j = 1, n/2 \) are set as follows, where each \( p \) measures the square of the radius of \( (y, z) \):

\[
p_j = y_j^2 + z_j^2
\]

If \( p_j \geq 1 \), 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:

\[
x_{2j-1} = y_j \left( \frac{-2 \ln p_j}{p_j} \right)^{0.5} \\
x_{2j} = z_j \left( \frac{-2 \ln p_j}{p_j} \right)^{0.5}
\]

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, \( n_{aux} = 0 \), and unable to allocate work area.

**Computational Errors**

None

**Input-Argument Errors**

1. \( n < 0 \) or \( n \) is an odd number
2. \( seed < 1.0 \) or \( seed \geq 2147483647.0 \)
3. Error 2015 is recoverable or \( n_{aux} \neq 0 \), and \( n_{aux} \) 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:

```
  CALL SNRAND( SEED , 10 , X , AUX , 5 )

  SEED     =  80629.0
```

**Note:** It is important to note that SEED is a long-precision number, even though X contains short-precision numbers.
Example 2

This example shows a call to DNRAND to generate 10 random numbers.

Call Statement and Input:

```
CALL DNRAND( SEED , 10 , X , AUX , 5 )
```

Output:

```
SEED     =  48669425.0
X        =  (0.660649538,
           1.312503695,
           1.906438112,
           0.014065863,
           -0.800935328,
           -3.058144093,
           -0.397426069,
           -0.370634943,
           -0.064151444,
           -0.275887042)
```
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 264 on page 1181.

\( n \)

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 264 on page 1181.

On Return

iseed

is set to 0 for subsequent calls to SURXOR or DURXOR. Returned as: an integer, as indicated in Table 264 on page 1181.

\( x \)

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 264 on page 1181.

\( vseed \)

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 264 on page 1181.

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 [“64” on page 1256], [“90” on page 1258], and [“115” on page 1259]. The values of $p$ and $q$ are selected according to the criteria stated in reference [“122” on page 1259].

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
```

Random number generation subroutines 1183
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,
0.329907556217007650,
0.66245283144861666,
0.249987282843358801,
0.80566368101257373,
0.73292147605172925,
0.748633579794102236,
0.80505169697755319,
0.9930249462139138,
0.84418748640269125,
0.68912952155489579,
0.31018611495627916,
0.63241786342211936,
```
0.32994081459690583,
0.35531452631408911,
0.0100134413132581,
0.02146199494672940,
0.80593898407597615)
Chapter 17. Utilities subroutines

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>
<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 1188</td>
</tr>
<tr>
<td>ERRSAV</td>
<td>“ERRSAV (ESSL ERRSAV Subroutine)” on page 1190</td>
</tr>
<tr>
<td>ERRSET</td>
<td>“ERRSET (ESSL ERRSET Subroutine)” on page 1191</td>
</tr>
<tr>
<td>ERRSTR</td>
<td>“ERRSTR (ESSL ERRSTR Subroutine)” on page 1192</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 1193</td>
</tr>
<tr>
<td>SETGPUS</td>
<td>“SETGPUS (Set the Number of GPUs and Identify Which GPUs ESSL Should Use)” on page 1194</td>
</tr>
<tr>
<td>STRIDE</td>
<td>“STRIDE (Determine the Stride Value for Optimal Performance in Specified Fourier Transform Subroutines)” on page 1195</td>
</tr>
</tbody>
</table>
| SLAMCH and DLAMCH | “SLAMCH and DLAMCH (Determine machine parameters for floating-point arithmetic)” on page 1210 }<br />
| DSRSM      | “DSRSM (Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode)” on page 1212 |
| DGKTRN     | “DGKTRN (For a General Sparse Matrix, Convert Between Diagonal-Out and Profile-In Skyline Storage Mode)” on page 1215 |
| DSKTRN     | “DSKTRN (For a Symmetric Sparse Matrix, Convert Between Diagonal-Out and Profile-In Skyline Storage Mode)” on page 1219 |

§ 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 959.

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 266 on page 1188. 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 141.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Receiver</th>
<th>Type of Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>2100</td>
<td>inf1</td>
<td>Lower range of a vector</td>
</tr>
<tr>
<td></td>
<td>inf2</td>
<td>Upper range of a vector</td>
</tr>
<tr>
<td>2101</td>
<td>inf1</td>
<td>Index of the eigenvalue that failed to converge</td>
</tr>
<tr>
<td></td>
<td>inf2</td>
<td>Number of iterations after which it failed to converge</td>
</tr>
</tbody>
</table>

Table 266: Computational Error Information Returned by EINFO.
Table 266: 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>2102</td>
<td>inf1</td>
<td>Index of the last eigenvector that failed to converge</td>
</tr>
<tr>
<td></td>
<td>inf2</td>
<td>Number of iterations after which it failed to converge</td>
</tr>
<tr>
<td>2103</td>
<td>inf1</td>
<td>Index of the pivot with zero value</td>
</tr>
<tr>
<td>2104</td>
<td>inf1</td>
<td>Index of the last pivot with nonpositive value</td>
</tr>
<tr>
<td>2105</td>
<td>inf1</td>
<td>Index of the pivot element near zero causing factorization to fail</td>
</tr>
<tr>
<td>2107</td>
<td>inf1</td>
<td>Index of the singular value that failed to converge</td>
</tr>
<tr>
<td></td>
<td>inf2</td>
<td>Number of iterations after which it failed to converge</td>
</tr>
<tr>
<td>2109</td>
<td>inf1</td>
<td>Iteration count when it was determined that the matrix was not definite</td>
</tr>
<tr>
<td>2114</td>
<td>inf1</td>
<td>Index of the last eigenvalue that failed to converge</td>
</tr>
<tr>
<td></td>
<td>inf2</td>
<td>Number of iterations after which it failed to converge</td>
</tr>
<tr>
<td>2115</td>
<td>inf1</td>
<td>Order of the leading minor that was discovered to have a nonpositive determinant</td>
</tr>
<tr>
<td>2117</td>
<td>inf1</td>
<td>Column number for which pivot value was near zero</td>
</tr>
<tr>
<td>2118</td>
<td>inf1</td>
<td>Row number for which pivot value was near zero</td>
</tr>
<tr>
<td>2120</td>
<td>inf1</td>
<td>Row number of empty row where factorization failed</td>
</tr>
<tr>
<td>2121</td>
<td>inf1</td>
<td>Column number of empty column where factorization failed</td>
</tr>
<tr>
<td>2126</td>
<td>inf1</td>
<td>Row number for which pivot value was unacceptable</td>
</tr>
<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

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL EINFO (icode[, inf1[, inf2]])</td>
</tr>
<tr>
<td>C and C++</td>
<td>einfo (icode, inf1, inf2);</td>
</tr>
</tbody>
</table>

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 266 on page 1188, 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 266 on page 1188.

inf1

See On Return.
inf2
See On Return.

On Return

inf1
has the following meaning, where:
If icode = 0, this argument is not used in the computation. In this case, inf1 is an optional argument, except in C and C++ programs.
If icode ≠ 0, then inf1 is the first information receiver, containing numerical information related to the computational error.
Returned as: an integer.

inf2
has the following meaning, where:
If icode = 0, this argument is not used in the computation.
If icode ≠ 0, then inf2 is the second information receiver, containing numerical information related to the computational error. It should be specified when the error code provides a second piece of information, and you want the information.
In both of these cases, inf2 is an optional argument, except in C and C++ programs. For more details, see “Notes” on page 1190.
Returned as: an integer.

Notes

1. If icode is not 0 and is not one of the error codes specified in Table 266 on page 1188, 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 programs,” on page 125.

ERRSAV (ESSL ERRSAV Subroutine)

Purpose
The ERRSAV subroutine copies an ESSL error option table entry into an 8-byte storage area that is accessible to your program.

For multithreaded application programs, if you want the error handling capabilities that this subroutine provides to be implemented on each thread created by your program, this subroutine must be called from each thread. If your application creates multiple threads, the action performed by a call to this subroutine applies to the thread that this subroutine was invoked from. For an example, see “Example of Handling Errors in a Multithreaded Application Program” on page 141.
ERRSAV (ESSL ERRSAV Subroutine)

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 40 on page 67.

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 programs,” on page 125.

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 141.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL ERRSET (ierno, inoal, inomes, itrace, iusadr, irange)</td>
</tr>
<tr>
<td>C and C++</td>
<td>errset (ierno, inoal, inomes, itrace, iusadr, irange);</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 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 40 on page 67.

  * **inoal**
    indicates the number of errors allowed before each execution is terminated, where:

    If *inoal* ≤ 0, the specification is ignored, and the number-of-errors option is not changed.
If $inoal = 1$, execution is terminated after one error.
If $2 \leq inoal \leq 255$, 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, $itrace = 0, 1, \text{or } 2$ (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; $iusadr = 0, 1, \text{or } ENOTRM$.
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; $iusadr = 0_8, 1_8, \text{or } ENOTRM$.

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 programs,” on page 125.
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 141.

Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL ERRSTR (ierno, tabent)</td>
</tr>
<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 40 on page 67.

**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 programs,” on page 125.

**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>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>IESSL ()</td>
</tr>
<tr>
<td>C and C++</td>
<td>iessl ();</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:

```
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>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>SETGPUS (ngpus,ids)</td>
</tr>
<tr>
<td>C and C++</td>
<td>setgpus (ngpus,ids);</td>
</tr>
</tbody>
</table>

On Entry
$ngpus$

is the number of GPUs ESSL should use.

Specified as: an integer; $0 < ngpus \leq$ 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:

```
NGPUS  IDS
|    |    |
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 1197.
Syntax

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</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; \( n > 0 \).

\( \text{incd} \)

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; \( \text{incd} > 0 \) or \( \text{incd} < 0 \).

\( \text{incr} \)

See On Return.

\( \text{dt} \)

is the data type of the numbers for the Fourier transform for which the optimal stride is being determined, where:

- If \( \text{dt} = 'S' \), the numbers are short-precision real.
- If \( \text{dt} = 'D' \), the numbers are long-precision real.
- If \( \text{dt} = 'C' \), the numbers are short-precision complex.
- If \( \text{dt} = 'Z' \), the numbers are long-precision complex.

Specified as: a single character; \( \text{dt} = 'S', 'D', 'C', \) or \( 'Z' \).

\( \text{iarg} \)

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; \( \text{iarg} = 0, 1, \) or \( 2 \).

On Return

\( \text{incr} \)

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; \( \text{incr} > 0 \) or \( \text{incr} < 0 \) and \( |\text{incr}| \geq |\text{incd}| \), where \( \text{incr} \) has the same sign (+ or -) as \( \text{incd} \).

Notes

1. In your C program, argument \( \text{incr} \) must be passed by reference.
2. All subroutines accept lowercase letters for the \( \text{dt} \) 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 \( \text{incd} \) argument. Details on how to compute each of these values is given in Example 1—SCFT. 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 959.

**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. 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 959.

**Error conditions**

**Computational Errors**

None

**Input-Argument Errors**

1. \( n \leq 0 \)
2. \( incd = 0 \)
3. \( iopt \neq 0, 1, \text{ or } 2 \)
4. \( dt \neq S, D, C, \text{ or } Z \)

**Examples**

**Example 1—SCFT**

This example shows the use of the STRIDE subroutine in computing one-dimensional row transforms using the SCFT subroutine.
If inc2x = 1, the input sequences are stored in the transposed form as rows of a two-dimensional array X(INC1X,N). In this case, the STRIDE subroutine helps in determining a good value of inc1x for this array. The required minimum value of inc1x is m, the number of Fourier transforms being computed. To find a good value of inc1x, use STRIDE as follows:

```
N   INCD  INCR    DT   IOPT
|   |     |       |    |
CALL STRIDE( N , M  , INC1X , 'C' , 0  )
```

Here, the arguments refer to the SCFT subroutine. In the following table, values of inc1x are given (as obtained from the STRIDE subroutine) for some combinations of n and m and for 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>65</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>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 can be used.

Example 2--DCOSF

This example shows the use of the STRIDE subroutine in computing one-dimensional row transforms using the DCOSF subroutine.

If inc2x = 1, the input sequences are stored in the transposed form as rows of a two-dimensional array X(INC1X,N/2+1). In this case, the STRIDE subroutine helps in determining a good value of inc1x for this array. The required minimum value of inc1x is m, the number of Fourier transforms being computed. To find a good value of inc1x, use STRIDE as follows:

```
N/2+1  INCD  INCR    DT   IOPT
|      |     |       |    |
CALL STRIDE( N/2+1 , M  , INC1X , 'D' , 0  )
```

Here, the arguments refer to the DCOSF subroutine. In the following table, values of inc1x are given (as obtained from the STRIDE subroutine) for some combinations of n and m and for 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>264</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:

```
N/2  INCD  INCR    DT   IOPT
|      |     |       |    |
CALL STRIDE( N/2 , M  , INC1X , 'D' , 0  )
```

Here, the arguments refer to the DSINF subroutine. In the following table, values of inc1x are given (as obtained from the STRIDE subroutine) for some combinations of n and m and for 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>264</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.
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:

\[
\begin{array}{ccccc}
N & \text{INCD} & \text{INCR} & \text{DT} & \text{IOPT} \\
\mid & \mid & \mid & \mid & \mid \\
\text{CALL STRIDE( N/2, M, \text{INC1X}, 'D', 0 )}
\end{array}
\]

Here, the arguments refer to the DSINF 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:

<table>
<thead>
<tr>
<th>( N )</th>
<th>( M )</th>
<th>( \text{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>264</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 (\( \text{inc2y} = 1 \)). In that case, in the above example, \( \text{inc1x} \) is replaced by \( \text{inc1y} \).

In computing column transforms (\( \text{inc1x} = \text{inc1y} = 1 \)), the values of \( \text{inc2x} \) and \( \text{inc2y} \) 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:

\[
\begin{array}{cccc}
N & \text{INCD} & \text{INCR} & \text{DT} \\
\mid & \mid & \mid & \mid \\
\text{CALL STRIDE( N2, N1, \text{INC2Y}, 'C', 0 )}
\end{array}
\]

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>240</td>
</tr>
<tr>
<td>512</td>
<td>512</td>
<td>528</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:

\[
\begin{array}{cccc}
N & \text{INCD} & \text{INCR} & \text{DT} \\
\mid & \mid & \mid & \mid \\
\text{CALL STRIDE( N1, N2, \text{INC1Y}, 'C', 0 )}
\end{array}
\]
Here, the arguments refer to the SCFT2 subroutine. In the following table, values of \textit{inc1y} are given (as obtained from the STRIDE subroutine) for some combinations of \textit{n1} and \textit{n2} and for POWER3 with 64K level 1 cache:

<table>
<thead>
<tr>
<th>N1</th>
<th>N2</th>
<th>INC1Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>120</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 (\textit{inc2x} = 1), the value of \textit{inc1x} is also important. The above example can be used to find a good value of \textit{inc1x}, by replacing \textit{inc1y} with \textit{inc1x}. If both arrays are stored in the transposed form, a good value for \textit{inc1y} is also a good value for \textit{inc1x}. In that situation, the two arrays can also be made equivalent.

\textbf{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 \textit{Y(INC2Y,N2)}, where the required minimum value of \textit{inc2y} is \textit{n1}/2+1. The STRIDE subroutine helps in picking a good value of \textit{inc2y}. To find a good value of \textit{inc2y}, use STRIDE as follows:

\begin{verbatim}
CALL STRIDE( N2 , N1/2 + 1 , INC2Y , 'C' , 0  )
\end{verbatim}

Here, the arguments refer to the SRCFT2 subroutine. In the following table, values of \textit{inc2y} are given (as obtained from the STRIDE subroutine) for some two-dimensional arrays with \textit{n1} = \textit{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 (\textit{inc2x}) is not important. If you want to use the same array for input and output, you should use \textit{inc2x} \geq 2(inc2y).

\textbf{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 \textit{Y(INC2Y,N2)}, where the required minimum value of \textit{inc2y} is \textit{n1}+2. The STRIDE subroutine helps in picking a good value of \textit{inc2y}. To find a good value of \textit{inc2y}, use STRIDE as follows:

\begin{verbatim}
CALL STRIDE( N2 , N1 + 2 , INC2Y , 'S' , 0  )
\end{verbatim}

Here, the arguments refer to the SCRFT2 subroutine. In the following table, values of \textit{inc2y} are given (as obtained from the STRIDE subroutine) for some two-dimensional arrays with \textit{n1} = \textit{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>
</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.

\[
\begin{array}{cccccc}
N & INCX & INCY & DT & IOPT \\
\hline
\end{array}
\]

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).

\[
\begin{array}{cccccc}
N & INCX & INCY & DT & IOPT \\
\hline
\end{array}
\]

CALL STRIDE( N3 , N2*INC2Y, INC3Y , 'C' , 0 )

If inc3y turns out to be a multiple of inc2y, then Y can be declared a three-dimensional array as Y(INC2Y,INC3Y/INC2Y,N3). For large problems, this may not happen. In that case, you can declare the Y array as a two-dimensional array Y(0:INC3Y-1,0:N3-1) or a one-dimensional array Y(0:INC3Y*N3-1). Using zero-based indexing, the element y(k1,k2,k3) is stored in the following location in these arrays:

- For the two-dimensional array, location (k1+k2*inc2y,k3)
- For the one-dimensional array, location (k1+k2*inc2y+k3*inc3y)

In the following table, values of inc2y and inc3y are given (as obtained from the STRIDE subroutine) for some three-dimensional arrays with n1 = n2 = n3 and for 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>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>4112</td>
</tr>
<tr>
<td>120</td>
<td>120</td>
<td>14408</td>
</tr>
<tr>
<td>128</td>
<td>136</td>
<td>17416</td>
</tr>
<tr>
<td>240</td>
<td>240</td>
<td>57698</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 inc2x ≥ inc2y and inc3x ≥ inc3y. A simple thing to do is to use inc2x = inc2y and make inc3x a multiple of inc2x not smaller than inc3y. Then X can be declared as a three-dimensional array X(INC2X,INC3X/INC2X,N3).

Example 8--SRCFT3

This example shows the use of the STRIDE subroutine in computing three-dimensional transforms using the SRCFT3 subroutine.

For this subroutine, the strides for the input array are not important. They are important for the output array. The STRIDE subroutine helps in picking good values of inc2y and inc3y. This requires two calls.
to the STRIDE subroutine as shown below. First, you should find a good value for \( inc2y \). The minimum acceptable value for \( inc2y \) is \( n1/2+1 \).

\[
\begin{array}{cccccc}
N & \text{INCD} & \text{INCR} & \text{DT} & \text{IOPT} \\
\hline
\text{CALL STRIDE( N2 , N1/2 + 1 , INC2Y , 'C' , 0 )}
\end{array}
\]

Here, the arguments refer to the SRCFT3 subroutine. Next, you should find a good value for \( inc3y \). The minimum acceptable value for \( inc3y \) is \( (n2)(inc2y) \).

\[
\begin{array}{cccccc}
N & \text{INCD} & \text{INCR} & \text{DT} & \text{IOPT} \\
\hline
\text{CALL STRIDE( N3 , N2*INC2Y , INC3Y , 'C' , 0 )}
\end{array}
\]

If \( inc3y \) turns out to be a multiple of \( inc2y \), then \( Y \) can be declared a three-dimensional array as \( Y(\text{INC2Y, INC3Y/INC2Y, N3}) \). 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 \( Y(0:INC3Y*N3-1) \).

If \( inc3y \) turns out to be a multiple of \( inc2y \), then \( Y \) can be declared a three-dimensional array as \( Y(\text{INC2Y, INC3Y/INC2Y, N3}) \). 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.

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 \( X(\text{INC2X, INC3X/INC2X, N3}) \).

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 \( n1+2 \).

\[
\begin{array}{cccccc}
N & \text{INCD} & \text{INCR} & \text{DT} & \text{IOPT} \\
\hline
\text{CALL STRIDE( N2 , N1 + 2 , INC2Y , 'S' , 0 )}
\end{array}
\]

Here, the arguments refer to the SCRFT3 subroutine. Next, you should find a good value for \( inc3y \). The minimum acceptable value for \( inc3y \) is \( (n2)(inc2y) \).

\[
\begin{array}{cccccc}
N & \text{INCD} & \text{INCR} & \text{DT} & \text{IOPT} \\
\hline
\text{CALL STRIDE( N3 , N2*INC2Y , INC3Y , 'S' , 0 )}
\end{array}
\]

If \( inc3y \) turns out to be a multiple of \( inc2y \), then \( Y \) can be declared a three-dimensional array as \( Y(\text{INC2Y, INC3Y/INC2Y, N3}) \). 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.
Using zero-based indexing, the element $y(k_1,k_2,k_3)$ is stored in the following location in these arrays:

- For the two-dimensional array, location $(k_1+k_2*inc2y,k_3)$
- For the one-dimensional array, location $(k_1+k_2*inc2y+k_3*inc3y)$

In the following table, values of $inc2y$ and $inc3y$ are given (as obtained from the STRIDE subroutine) for some three-dimensional arrays with $n_1 = n_2 = n_3$ 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>
<tr>
<td>64</td>
<td>66</td>
<td>4256</td>
</tr>
<tr>
<td>128</td>
<td>122</td>
<td>14640</td>
</tr>
<tr>
<td>128</td>
<td>130</td>
<td>16656</td>
</tr>
<tr>
<td>240</td>
<td>242</td>
<td>58128</td>
</tr>
<tr>
<td>256</td>
<td>258</td>
<td>66064</td>
</tr>
<tr>
<td>420</td>
<td>422</td>
<td>177240</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 $incx_1$ 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:

```
CALL STRIDE( N(1) , M  , INCX(1) , 'C' , 0 )
```

Here, the arguments refer to the SCFTD 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</th>
<th>M</th>
<th>INC1X</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 ($incy_1 = 1$). In that case, in the above example, $incx_1$ is replaced by $incy_1$.

In computing column transforms ($incx_1 = incy_1 = 1$), the values of $incmx$ and $incy$ are not very important. For these, any value over the required minimum of $n_1$ 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 $incy_1 = 1$, 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 $incy_2$ is $n_1$. The
STRIDE subroutine helps in picking a good value of incy₂. To find a good value of incy₂, use STRIDE as follows:

```
N INCD INCR DT IOPT
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:

<table>
<thead>
<tr>
<th>N(1)</th>
<th>N(2)</th>
<th>INCY(2)</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>240</td>
</tr>
<tr>
<td>512</td>
<td>512</td>
<td>520</td>
</tr>
<tr>
<td>840</td>
<td>840</td>
<td>840</td>
</tr>
</tbody>
</table>

If the input array is stored in the normal form (incx₁ = 1), the value of incx₂ is not important. However, if you want to use the same array for input and output, you should use incx₂ = incy₂.

If incy₂ = 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 incy₁ is n₂.

The STRIDE subroutine helps in picking a good value of incy₁. To find a good value of incy₁, use STRIDE as follows:

```
N INCD INCR DT IOPT
CALL STRIDE( N(1) , N(2) , INCY(1) , 'C' , 0 )
```

Here, the arguments refer to the SCFTD subroutine. In the following table, values of incy₁ are given (as obtained from the STRIDE subroutine) for some combinations of n₁ and n₂ 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>60</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>120</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>840</td>
</tr>
</tbody>
</table>

If the input array is stored in the transposed form (incx₂ = 1), the value of incx₁ is also important. The above example can be used to find a good value of incx₂, by replacing incy₁ with incx₁. If both arrays are stored in the transposed form, a good value for incy₁ is also a good value for incx₁. 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 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 n₁.

```
N INCD INCR DT IOPT
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 incy3. The minimum acceptable value for incy3 is \((n_2)/(incy_2)\) assuming incy_1 = 1.

\[
\begin{array}{cccc}
N & INCD & INCR & DT & IOPT \\
\hline
\text{CALL STRIDE( N(3) , N(2) + INCY(2) , INCY(3) , 'C' , 0 )}
\end{array}
\]

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 \(Y(0:INCY(3)\times N(3)-1)\). 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 \((k1+k2\times incy_2,k3)\)
- For the one-dimensional array, location \((k1+k2\times incy_2+k3\times incy_3)\)

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>128</td>
<td>128</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>176460</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 \(X(INCX(2) , INCX(3)/INCX(2) , N(3))\).

**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:

\[
\begin{array}{cccc}
N & INCD & INCR & DT & IOPT \\
\hline
\text{CALL STRIDE( N(1) , M  , INCX(1) , 'S' , 0 )}
\end{array}
\]

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>68</td>
<td>68</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 \( \text{incy}_1 \) for this array. The required minimum value of \( \text{incy}_1 \) is \( m \), the number of Fourier transforms being computed. To find a good value of \( \text{incy}_1 \), use STRIDE as follows:

\[
\begin{array}{cccccc}
N & \text{INCD} & \text{INCR} & DT & \text{IOPT} \\
\hline
\text{CALL STRIDE(} & N(1)/2+1 & , & M & , & \text{INCY}(1) & , & 'C' & , & 0 & ) \\
\end{array}
\]

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 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 \( \text{incmx} \) and \( \text{incmy} \) 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 \( Y(\text{INCY}(2) , \text{N}(2)) \), 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:

\[
\begin{array}{cccccc}
N & \text{INCD} & \text{INCR} & DT & \text{IOPT} \\
\hline
\text{CALL STRIDE(} & \text{N}(2) & , & N(1)/2+1 & , & \text{INCY}(2) & , & 'C' & , & 0 & ) \\
\end{array}
\]

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 \( Y(\text{INCY}(1) , \text{N}(1)/2+1) \), 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:

\[
\begin{array}{cccccc}
N & \text{INCD} & \text{INCR} & DT & \text{IOPT} \\
\hline
\text{CALL STRIDE(} & N(1)/2+1 & , & \text{N}(2) & , & \text{INCY}(1) & , & 'C' & , & 0 & ) \\
\end{array}
\]
Here, the arguments refer to the SRCFTD subroutine. In the following table, values of incy2 are given (as obtained from the STRIDE subroutine) for some combinations of n1 and n2 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>240</td>
</tr>
<tr>
<td>420</td>
<td>420</td>
<td>420</td>
</tr>
<tr>
<td>512</td>
<td>512</td>
<td>520</td>
</tr>
<tr>
<td>840</td>
<td>840</td>
<td>840</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>1032</td>
</tr>
<tr>
<td>2048</td>
<td>2048</td>
<td>2056</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 n1/2+1.

```
CALL STRIDE( N(2) , N(1)/2+1 , INCY(2) , 'C' , 0 )
```

Here, the arguments refer to the SRCFTD subroutine. Next, you should find a good value for incy3. The minimum acceptable value for incy3 is (n2)(incy2) assuming incy1 equal to 1.

```
CALL STRIDE( N(3) , N(2)*INCY(2) , INCY(3) , 'C' , 0 )
```

If incy3 turns out to be a multiple of n2, 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 yk1,k2,k3 is stored in the following location in these arrays:

- For the two-dimensional array, location (k1+k2*incy2,k3)
- For the one-dimensional array, location (k1+k2*incy2+k3*incy3)

In the following table, values of incy2 and incy3 are given (as obtained from the STRIDE subroutine) for some three-dimensional arrays with n1 equal to n2 equal to n3 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>16</td>
<td>480</td>
</tr>
<tr>
<td>32</td>
<td>18</td>
<td>576</td>
</tr>
<tr>
<td>64</td>
<td>34</td>
<td>2176</td>
</tr>
<tr>
<td>120</td>
<td>62</td>
<td>7440</td>
</tr>
<tr>
<td>128</td>
<td>66</td>
<td>8456</td>
</tr>
<tr>
<td>240</td>
<td>122</td>
<td>29289</td>
</tr>
<tr>
<td>256</td>
<td>130</td>
<td>33288</td>
</tr>
<tr>
<td>420</td>
<td>212</td>
<td>89040</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 incx2 equal to 2(incy2) and incx3 equal to 2(incy3).

**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 \(\text{incmx}\) equal to 1, the input sequences are stored in the transposed form as rows of a two-dimensional array \(X(\text{INCX}(1),N(1))\). In this case, the STRIDE subroutine helps in determining a good value of \(\text{incy}_1\) for this array. The required minimum value of \(\text{incy}_1\) is \(m\), the number of Fourier transforms being computed. To find a good value of \(\text{incy}_1\), use STRIDE as follows:

\[
\begin{array}{|c|c|c|c|c|}
\hline
N & \text{INCD} & \text{INCR} & \text{DT} & \text{IOPT} \\
\hline
\text{CALL STRIDE}( N(1)/2+1, M, \text{INCX}(1), 'C', 0 ) \\
\hline
\end{array}
\]

Here, the arguments refer to the SCRFTD 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{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 \(\text{incmy}\) 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 \(\text{incy}_1\) for this array. The required minimum value of \(\text{incy}_1\) is \(m\), the number of Fourier transforms being computed. To find a good value of \(\text{incy}_1\), use STRIDE as follows:

\[
\begin{array}{|c|c|c|c|c|}
\hline
N & \text{INCD} & \text{INCR} & \text{DT} & \text{IOPT} \\
\hline
\text{CALL STRIDE}( N(1), M, \text{INCY}(1), 'S', 0 ) \\
\hline
\end{array}
\]

Here, the arguments refer to the SCRFTD 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>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 (\(\text{incx}_1\) equal to \(\text{incy}_1\) equal to 1), the values of \(\text{incmx}\) and \(\text{incmy}\) 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 \(\text{incy}_2\) 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 \(n_1+2\). 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:

\[
\begin{array}{|c|c|c|c|c|}
\hline
N & \text{INCD} & \text{INCR} & \text{DT} & \text{IOPT} \\
\hline
\text{CALL STRIDE}( N(2), N(1)+2, \text{INCY}(2), 'S', 0 ) \\
\hline
\end{array}
\]

Here, the arguments refer to the SCRFTD 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>244</td>
</tr>
</tbody>
</table>
If the input array is stored in the normal form (incx equal to 1), the value of incx is not important. However, if you want to use the same array for input and output, you should use incy equal to 2(incx).

If incy 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 incly is n_2. The STRIDE subroutine helps in picking a good value of incy. To find a good value of incy, use STRIDE as follows:

```fortran
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 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>240</td>
</tr>
<tr>
<td>420</td>
<td>420</td>
<td>420</td>
</tr>
<tr>
<td>512</td>
<td>512</td>
<td>528</td>
</tr>
<tr>
<td>840</td>
<td>840</td>
<td>840</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>1040</td>
</tr>
<tr>
<td>2048</td>
<td>2048</td>
<td>2064</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 and incy. This requires two calls to the STRIDE subroutine as shown below. First, you should find a good value for incy. The minimum acceptable value for incy is n_1+2.

```fortran
CALL STRIDE( N(2) , N(1)+2 , INCY(2) , 'S' , 0 )
```

Here, the arguments refer to the SCRFTD subroutine. Next, you should find a good value for incy. The minimum acceptable value for incy is (n_2)(incy) assuming incy equal to 1.

```fortran
CALL STRIDE( N(3) , N(2)*INCY(2) , INCY(3) , 'S' , 0 )
```

If incy turns out to be a multiple of incy, then Y can be declared a three-dimensional array as Y(INCY(2),INCY(3)/ICY(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,k3)
- For the one-dimensional array, location (k1+k2*incy+k3*incy)
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>( \text{INCY}(2) )</th>
<th>( \text{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>128</td>
<td>124</td>
<td>4352</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 \( 2(\text{incx}_2) \) and \( \text{incy}_3 \) equal to \( 2(\text{incx}_3) \).

\[ \text{SLAMCH and DLAMCH (Determine machine parameters for floating-point arithmetic)} \]

**Purpose**

SLAMCH and DLAMCH determine the machine parameters for floating-point arithmetic.

**Table 267: Data Types.**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Subprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-precision real</td>
<td>SLAMCH</td>
</tr>
<tr>
<td>Long-precision real</td>
<td>DLAMCH</td>
</tr>
</tbody>
</table>

**Syntax**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>SLAMCH</td>
</tr>
<tr>
<td>C and C++</td>
<td>slamch</td>
</tr>
<tr>
<td>LAPACKE</td>
<td>LAPACKE_slamch</td>
</tr>
</tbody>
</table>

**On Entry cmach**

- If \( \text{cmach} = 'E' \), the relative machine precision (\( \text{eps} \)) is returned.
- If \( \text{cmach} = 'S' \), the safe minimum (\( \text{sfmin} \)) is returned, such that \( 1.0/\text{sfmin} \) does not overflow.
- If \( \text{cmach} = 'B' \), the base of the machine (\( \text{base} \)) is returned.
- If \( \text{cmach} = 'P' \), \( \text{eps} \times \text{base} \) (\( \text{prec} \)) is returned.
- If \( \text{cmach} = 'N' \), the number of (\( \text{base} \)) digits in the mantissa (\( \text{t} \)) is returned.
- If \( \text{cmach} = 'R' \), one is returned when rounding occurs in addition; otherwise, zero (\( \text{rnd} \)) is returned.
- If \( \text{cmach} = 'M' \), the minimum exponent before gradual underflow (\( \text{emin} \)) is returned.
If $cmach = 'U'$, $underflow\_threshold - base^{emin-1.0}$ ($rmin$) is returned.

If $cmach = 'L'$, the largest exponent before overflow ($emax$) is returned.

If $cmach = 'O'$, $overflow\_threshold - (base^{emax})*(1.0-eps)$ ($rmax$) is returned.

Specified as: a single character. It can be any value.

**On Return**

**Function value**

is the machine parameter. Returned as: a number of the data type indicated in Table 267 on page 1210.

**Note:** Declare this function in your program as returning a value of the type indicated in Table 267 on page 1210.

**Notes**

1. All subroutines accept lowercase letters for the $cmach$ argument.
2. If you specify a value that is not listed for the $cmach$ argument, a zero is returned.

**Function**

SLAMCH and DLAMCH determine the machine parameters for floating-point arithmetic.

**Error conditions**

**Computational Errors**

None

**Input-Argument Errors**

None

**Examples**

**Example 1**

The following example determines the relative machine precision.

Call Statement and Input:

```
CMACH
EPS = SLAMCH ('E')
```

Output:

```
EPS = 0.5960464478E-07
```

**Example 2**

The following example determines the relative machine precision.

Call Statement and Input:

```
CMACH
EPS = DLAMCH ('E')
```

Output:

```
EPS = 0.111022302462515654D-15
```
DSRSM (Convert a Sparse Matrix from Storage-by-Rows to Compressed-Matrix Storage Mode)

**Purpose**

This subroutine converts either a \( m \) 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>Syntax</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) \( nz \leq (output \ argument) nz \).

\( ac \)

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 110 and “Storage-by-Rows” on page 115.

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 [“89” on page 1258].

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. \( iopt \neq 0 \) or \( 1 \)
2. \( m < 0 \)
3. \( lda < 1 \)
4. \( lda < m \)
5. \( nz \leq 0 \)
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 $nz$ is specified as 5, columns 3 through 5 of arrays AC and KA are not used.

Call Statement and Input:

```
IOPT  AR   JA   IA   M   NZ  AC   KA  LDA
|    |    |    |    |   |   |    |    |   
CALL DSRSM( 0 , AR , JA , IA , 4 , 5 , AC , KA , 4 )
AR       = (11.0, 14.0, 22.0, 24.0, 33.0, 34.0, 44.0)
JA       = (1, 4, 2, 4, 3, 4, 4)
IA       = (1, 3, 5, 7, 8)
```

Output:

$$
\begin{bmatrix}
11.0 & 14.0 & . & . & . \\
22.0 & 24.0 & . & . & . \\
33.0 & 34.0 & . & . & . \\
44.0 & 0.0 & . & . & . \\
\end{bmatrix}
$$

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:

$$
\begin{bmatrix}
11.0 & 0.0 & 0.0 & 14.0 & 24.0 & 34.0 & 44.0 \\
0.0 & 22.0 & 0.0 & 24.0 & 34.0 & 44.0 & \\
0.0 & 0.0 & 33.0 & 34.0 & 44.0 & \\
14.0 & 24.0 & 34.0 & 44.0 & \\
\end{bmatrix}
$$

Because there is a maximum of only four nonzero elements in each row of $A$, and argument $nz$ is specified as 6, columns 5 and 6 of arrays AC and KA are not used.

Call Statement and Input:

```
IOPT  AR   JA   IA   M   NZ  AC   KA  LDA
|    |    |    |    |   |   |    |    |   
CALL DSRSM( 1 , AR , JA , IA , 4 , 6 , AC , KA , 4 )
AR       = (11.0, 14.0, 22.0, 24.0, 33.0, 34.0, 44.0)
JA       = (1, 4, 2, 4, 3, 4, 4)
IA       = (1, 3, 5, 7, 8)
```
Output:

```
NZ      =  4
| 11.0  14.0   0.0   0.0  .  . |
| 22.0  24.0   0.0   0.0  .  . |
| 33.0  34.0   0.0   0.0  .  . |
| 44.0  24.0  34.0  14.0  .  . |
AC      =  | 1  4  4  4  .  . |
| 2  4  4  4  .  . |
| 3  4  4  4  .  . |
| 4  2  3  1  .  . |
KA      =  | 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  .  . |
```

**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><code>dgktrn(n, au, nu, idu, al, nl, idl, itran, aux, naux);</code></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 $ITRAN(1) = 0$, $A$ is stored in diagonal-out skyline storage mode.
If $ITRAN(1) = 1$, $A$ is stored in profile-in skyline storage mode.

**Note:** Entries in AL for diagonal elements of $A$ are assumed not to have meaningful values.

Specified as: a one-dimensional array of (at least) length $nl$, containing long-precision real numbers.

$nl$

is the length of array AL.
Specified as: an integer; $nl \geq 0$ and $nl \geq (IDL(n+1)-1)$.

$idl$

is the array, referred to as IDL, containing the relative positions of the diagonal elements of matrix $A$ in input array AL.
Specified as: a one-dimensional array of (at least) length $n+1$, containing integers.

$itrans$

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 1217), 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

\[ \text{naux} \geq 4n \]

**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.

*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.

*al* is the array, referred to as AL, containing the lower triangular part of general sparse matrix *A*, stored as follows, where:

- If ITRAN(2) = 0, *A* is stored in diagonal-out skyline storage mode.
- If ITRAN(2) = 1, *A* is stored in profile-in skyline storage mode.

**Note:** You should assume that entries in AL for diagonal elements of *A* do not have meaningful values.

Returned as: a one-dimensional array of (at least) length \(nl\), containing long-precision real numbers.

*idl* is the array, referred to as IDL, containing the relative positions of the diagonal elements of matrix *A* in output array AL. Returned as: a one-dimensional array of (at least) length \(n + 1\), containing integers.

**Notes**

1. Your various arrays must have no common elements; otherwise, results are unpredictable.
2. The ITRAN(3) argument allows you to specify the direction of travel through matrix *A* that ESSL takes during the transformation. By properly specifying ITRAN(3), you can optimize the performance of the transformation, which is especially beneficial when transforming large matrices.

   The direction specified by ITRAN(3) should be opposite the most recent direction of access through the matrix performed by the DGKFS or DGKFSP subroutine, as indicated in the following table:

<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 119 and “Diagonal-Out Skyline Storage Mode” on page 117.

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 49.

**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 $naux=0$, 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 convert a 9 by 9 general sparse matrix $A$ from diagonal-out skyline storage mode to profile-in skyline storage mode. Matrix $A$ is:

$$
\begin{bmatrix}
11.0 & 12.0 & 13.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
21.0 & 22.0 & 23.0 & 24.0 & 25.0 & 0.0 & 0.0 & 0.0 & 29.0 \\
31.0 & 32.0 & 33.0 & 34.0 & 35.0 & 0.0 & 37.0 & 0.0 & 39.0 \\
41.0 & 42.0 & 43.0 & 44.0 & 45.0 & 46.0 & 47.0 & 0.0 & 49.0 \\
50.0 & 0.0 & 0.0 & 54.0 & 55.0 & 56.0 & 57.0 & 58.0 & 59.0 \\
60.0 & 62.0 & 63.0 & 64.0 & 65.0 & 66.0 & 67.0 & 68.0 & 69.0 \\
70.0 & 0.0 & 0.0 & 74.0 & 75.0 & 76.0 & 77.0 & 78.0 & 79.0 \\
80.0 & 0.0 & 0.0 & 84.0 & 85.0 & 86.0 & 87.0 & 88.0 & 89.0 \\
91.0 & 92.0 & 93.0 & 94.0 & 95.0 & 96.0 & 97.0 & 98.0 & 99.0 \\
\end{bmatrix}
$$

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.
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:**

```plaintext
CALL DGKTRN( 9 , AU , 33 , IDU , AL , 35 , IDL , ITRAN , AUX , 18

AU     = (11.0, 22.0, 12.0, 33.0, 23.0, 13.0, 44.0, 34.0, 24.0,
         55.0, 45.0, 35.0, 25.0, 66.0, 56.0, 46.0, 77.0, 67.0,
         78.0, 57.0, 47.0, 37.0, 88.0, 78.0, 68.0, 58.0, 99.0, 89.0,
         79.0, 69.0, 59.0, 49.0, 39.0, 29.0)
IDU    = (1, 2, 4, 7, 10, 14, 17, 22, 26, 34)
AL     = ( . , . , 21.0, . , 32.0, 31.0, . , 43.0, 42.0, 41.0, . ,
         54.0, . , 65.0, 64.0, 63.0, 62.0, . , 76.0, 75.0, 74.0, 73.0,
         . , 87.0, 86.0, 85.0, 84.0, . , 98.0, 97.0, 96.0, 95.0, 94.0,
         93.0, 92.0, 91.0)
IDL    = (1, 2, 4, 7, 11, 13, 18, 22, 27, 36)
ITRAN  = (0, 1, 1)
```

**Output:**

```plaintext
AU     = (11.0, 12.0, 22.0, 13.0, 23.0, 33.0, 24.0, 34.0, 44.0,
         25.0, 35.0, 45.0, 55.0, 46.0, 56.0, 47.0, 37.0, 47.0,
         57.0, 67.0, 77.0, 58.0, 68.0, 78.0, 88.0, 29.0, 39.0,
         49.0, 59.0, 69.0, 79.0, 89.0, 99.0)
IDU    = (1, 3, 6, 9, 13, 16, 21, 25, 33, 34)
AL     = ( . , 21.0, . , 31.0, 32.0, . , 41.0, 42.0, 43.0, . , 54.0,
         . , 62.0, 63.0, 64.0, 65.0, . , 74.0, 75.0, 76.0, 73.0,
         84.0, 85.0, 86.0, 87.0, . , 91.0, 92.0, 93.0, 94.0, 95.0,
         96.0, 97.0, 98.0, . )
IDL    = (1, 3, 6, 10, 12, 17, 21, 26, 35, 36)
```

**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**

<table>
<thead>
<tr>
<th>Language</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>CALL DSKTRN (n, a, na, idag, itran, aux, naaux)</td>
</tr>
<tr>
<td>C and C++</td>
<td>dsktrn (n, a, na, idag, itran, aux, naaux);</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 the upper triangular part of symmetric 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 \( na \), containing long-precision real numbers.

- **na**
  - is the length of array \( A \).
  - Specified as: an integer; \( na \geq 0 \) and \( na \geq (\text{IDIAG}(n+1)-1) \).

- **idag**
  - is the array, referred to as IDIAG, containing the relative positions of the diagonal elements of matrix \( A \) in input array \( A \).
  - Specified as: a one-dimensional array of (at least) length \( n+1 \), containing integers.

- **itran**
  - is an array of parameters, ITRAN(i), where:
    - \( \text{ITRAN}(1) \) indicates the input storage mode used for matrix \( A \). This determines the arrangement of data in arrays \( A \) and IDIAG on input, where:
      - If \( \text{ITRAN}(1) = 0 \), diagonal-out skyline storage mode is used.
      - If \( \text{ITRAN}(1) = 1 \), profile-in skyline storage mode is used.
    - \( \text{ITRAN}(2) \) indicates the output storage mode used for matrix \( A \). This determines the arrangement of data in arrays \( A \) and IDAIG on output, where:
      - If \( \text{ITRAN}(2) = 0 \), diagonal-out skyline storage mode is used.
      - If \( \text{ITRAN}(2) = 1 \), profile-in skyline storage mode is used.
    - \( \text{ITRAN}(3) \) indicates the direction of sweep that ESSL uses through the matrix \( A \), allowing you to optimize performance (see “Notes” on page 1221), where:
      - If \( \text{ITRAN}(3) = 1 \), matrix \( A \) is transformed in the positive direction, starting in row or column 1 and ending in row or column \( n \).
      - If \( \text{ITRAN}(3) = -1 \), 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:
    - \( \text{ITRAN}(1) = 0 \) or \( 1 \)
    - \( \text{ITRAN}(2) = 0 \) or \( 1 \)
    - \( \text{ITRAN}(3) = -1 \) 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, DSKTRN dynamically allocates the work area used by this subroutine. The work area is deallocated before control is returned to the calling program.
Otherwise
For 32-bit integer arguments
\( naux \geq n \)
For 64-bit integer arguments
\( naux \geq 2n \)

On Return
\( a \)
is the array, referred to as A, containing the upper triangular part of symmetric sparse matrix \( A \), stored as follows, where:
If \( \text{ITRAN}(2) = 0 \), \( A \) is stored in diagonal-out skyline storage mode.
If \( \text{ITRAN}(2) = 1 \), \( A \) is stored in profile-in skyline storage mode.
Returned as: a one-dimensional array of (at least) length \( na \), containing long-precision real numbers.

\( \text{idia} \)
is the array, referred to as IDIAG, containing the relative positions of the diagonal elements of matrix \( A \) in output array A. Returned as: a one-dimensional array of (at least) length \( n+1 \), containing integers.

Notes
1. Your various arrays must have no common elements; otherwise, results are unpredictable.
2. The ITRAN(3) argument allows you to specify the direction of travel through matrix \( A \) that ESSL takes during the transformation. By properly specifying ITRAN(3), you can optimize the performance of the transformation, which is especially beneficial when transforming large matrices.

The direction specified by ITRAN(3) should be opposite the most recent direction of access through the matrix performed by the DSKFS or DSKFSP subroutine, as indicated in the following table:

<table>
<thead>
<tr>
<th>Most Recent Computation Performed by DSKFS/DSKFSP</th>
<th>Direction Used by DSKFS/DSKFSP</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 119 and “Diagonal-Out Skyline Storage Mode” on page 117.
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 49.
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 $\text{ITRAN}(3)$ 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. $\text{IDIAG}(n+1) > na+1$
4. $\text{IDIAG}(i+1) \leq \text{IDIAG}(i)$ for $i = 1, n$
5. $\text{IDIAG}(i+1) > \text{IDIAG}(i)+i$ and $\text{ITRAN}(1) = 0$ for $i = 1, n$
6. $\text{IDIAG}(i) > \text{IDIAG}(i-1)+i$ and $\text{ITRAN}(1) = 1$ for $i = 2, n$
7. $\text{ITRAN}(1) \neq 0 \text{ or } 1$
8. $\text{ITRAN}(2) \neq 0 \text{ or } 1$
9. $\text{ITRAN}(3) \neq -1 \text{ or } 1$
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:

\[
\begin{bmatrix}
11.0 & 12.0 & 13.0 & 14.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
12.0 & 22.0 & 23.0 & 24.0 & 25.0 & 26.0 & 0.0 & 28.0 & 0.0 \\
13.0 & 23.0 & 33.0 & 34.0 & 35.0 & 36.0 & 0.0 & 38.0 & 0.0 \\
14.0 & 24.0 & 34.0 & 44.0 & 45.0 & 46.0 & 0.0 & 48.0 & 0.0 \\
0.0 & 25.0 & 35.0 & 45.0 & 55.0 & 56.0 & 57.0 & 58.0 & 0.0 \\
0.0 & 26.0 & 36.0 & 46.0 & 56.0 & 66.0 & 67.0 & 68.0 & 69.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 57.0 & 67.0 & 77.0 & 78.0 & 79.0 \\
0.0 & 28.0 & 38.0 & 48.0 & 58.0 & 68.0 & 78.0 & 88.0 & 89.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 69.0 & 79.0 & 89.0 & 99.0 & 0.0
\end{bmatrix}
\]

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:

```
CALL DSKTRN( 9 , A , 33 , IDIAG , ITRAN , AUX , 9 )
```

**Output:**

```
A        =  (11.0, 22.0, 12.0, 33.0, 13.0, 44.0, 34.0, 24.0,
           14.0, 55.0, 45.0, 35.0, 25.0, 66.0, 56.0, 46.0, 36.0,
           26.0, 77.0, 67.0, 57.0, 88.0, 78.0, 68.0, 58.0, 48.0,
           38.0, 28.0, 99.0, 89.0, 79.0, 69.0)
IDIAG    =  (1, 2, 4, 7, 11, 15, 20, 23, 30, 34)
ITRAN    =  (0, 1, -1)
```
Appendix A. Basic Linear Algebra Subprograms (BLAS) and C BLAS (CBLAS)

The following information 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>
</table>
| “ISAMAX, IDAMAX, ICAMAX, and IZAMAX (Position of the First or Last Occurrence of the Vector Element Having the Largest Magnitude)” on page 204 | ISAMAX  
ICAMAX  
cblas_isamax  
cblas_icamax  
   | IDAMAX  
IZAMAX  
cblas_idamax  
cblas_izamax  
   |
| “SASUM, DASUM, SCASUM, and DZASUM (Sum of the Magnitudes of the Elements in a Vector)” on page 214 | SASUM  
SCASUM  
cblas_sasum  
cblas_scasum  
   | DASUM  
DZASUM  
cblas_dasum  
cblas_dcasum  
   |
| “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 216 | SAXPY  
CAXPY  
cblas_saxpy  
cblas_caxpy  
   | DAXPY  
ZAXPY  
cblas_daxpy  
cblas_zaxpy  
   |
| “SCOPY, DCOPY, CCOPY, and ZCOPY (Copy a Vector)” on page 223                    | SCOPY  
CCOPY  
cblas_scopy  
cblas_ccopy  
   | DCOPY  
ZCOPY  
cblas_dccopy  
cblas_zcopy  
   |
| “SDOT, DDOT, CDOTU, ZDOTU, CDOTC, and ZDOTC (Dot Product of Two Vectors)” on page 225 | SDOT  
CDOTU  
CDOTC  
cblas_sdot  
cblas_cdotu_sub  
cblas_cdotc_sub  
   | DDOT  
ZDOTU  
ZDOTC  
cblas_ddot  
cblas_zdotu_sub  
cblas_zdotc_sub  
   |
| “SNRM2, DNRM2, SCNRM2, and DZNRM2 (Euclidean Length of a Vector with Scaling of Input to Avoid Destructive Underflow and Overflow)” on page 238 | SNRM2  
SCNRM2  
cblas_snrm2  
cblas_scnrm2  
   | DNRM2  
DZNRM2  
cblas_dnrm2  
cblas_dznrm2  
   |
| “SROTG, DROTG, CROTG, and ZROTG (Construct a Givens Plane Rotation)” on page 244 | SROTG  
CROTG  
cblas_srotg  
cblas_crotg  
   | DROTG  
ZROTG  
cblas_drotg  
cblas_zrotg  
   |
### Level 1 BLAS

#### Table 269: 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>“SROT, DROT, CROT, ZROT, CSROT, and ZDROT (Apply a Plane Rotation)” on page 250</td>
<td><img src="cblas_srot" alt="Subprogram" /></td>
<td><img src="cblas_zdrot" alt="Subprogram" /></td>
</tr>
<tr>
<td>“SROTMG and DROTMG (Construct a modified Givens Transformation)” on page 254</td>
<td><img src="cblas_srotmg" alt="Subprogram" /></td>
<td><img src="cblas_drotmg" alt="Subprogram" /></td>
</tr>
<tr>
<td>“SROT and DROT (Apply a modified Givens Transformation)” on page 256</td>
<td><img src="cblas_srotm" alt="Subprogram" /></td>
<td><img src="cblas_drotm" alt="Subprogram" /></td>
</tr>
<tr>
<td>“SSCAL, DSCAL, CSCAL, ZSCAL, CSSCAL, and ZDSCAL (Multiply a Vector X by a Scalar and Store in the Vector X)” on page 259</td>
<td><img src="cblas_sscal" alt="Subprogram" /></td>
<td><img src="cblas_zdscal" alt="Subprogram" /></td>
</tr>
<tr>
<td>“SSWAP, DSWAP, CSWAP, and ZSWAP (Interchange the Elements of Two Vectors)” on page 262</td>
<td><img src="cblas_sswap" alt="Subprogram" /></td>
<td><img src="cblas_zswap" alt="Subprogram" /></td>
</tr>
</tbody>
</table>

#### Level 2 BLAS

#### Table 270: 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 295</td>
<td><img src="cblas_sgemv" alt="Subprogram" /></td>
<td><img src="cblas_zgemv" alt="Subprogram" /></td>
</tr>
<tr>
<td>“SGER, DGER, CGERU, ZGERU, CGERC, and ZGERC (Rank-One Update of a General Matrix)” on page 306</td>
<td><img src="cblas_sger" alt="Subprogram" /></td>
<td><img src="cblas_zgerc" alt="Subprogram" /></td>
</tr>
</tbody>
</table>
Table 270: 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>“SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV, SSLMX, and DSLMX (Matrix-Vector Product for a Real Symmetric or Complex Hermitian Matrix)” on page 313</td>
<td>SSPMV</td>
<td>DspmV</td>
</tr>
<tr>
<td></td>
<td>CHPMV</td>
<td>ZhpMV</td>
</tr>
<tr>
<td></td>
<td>SSYMV</td>
<td>DsymV</td>
</tr>
<tr>
<td></td>
<td>CHEMV</td>
<td>ZhemV</td>
</tr>
<tr>
<td></td>
<td>cblas_sspmv</td>
<td>cblas_dspmv</td>
</tr>
<tr>
<td></td>
<td>cblas_chpmv</td>
<td>cblas_zhpvm</td>
</tr>
<tr>
<td></td>
<td>cblas_ssymv</td>
<td>cblas_dsymv</td>
</tr>
<tr>
<td></td>
<td>cblas_chemv</td>
<td>cblas_zhemv</td>
</tr>
<tr>
<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 321</td>
<td>SSPR</td>
<td>DSPR</td>
</tr>
<tr>
<td></td>
<td>CHPR</td>
<td>ZHPR</td>
</tr>
<tr>
<td></td>
<td>SSYR</td>
<td>DSYR</td>
</tr>
<tr>
<td></td>
<td>CHER</td>
<td>ZHER</td>
</tr>
<tr>
<td></td>
<td>cblas_ssspr</td>
<td>cblas_dspr</td>
</tr>
<tr>
<td></td>
<td>cblas_chpr</td>
<td>cblas_zhprr</td>
</tr>
<tr>
<td></td>
<td>cblas_ssyrr</td>
<td>cblas_dsymv</td>
</tr>
<tr>
<td></td>
<td>cblas_ cher</td>
<td>cblas_zher</td>
</tr>
<tr>
<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 328</td>
<td>SSPR2</td>
<td>DSPR2</td>
</tr>
<tr>
<td></td>
<td>CHPR2</td>
<td>ZHPR2</td>
</tr>
<tr>
<td></td>
<td>SSYR2</td>
<td>DSYR2</td>
</tr>
<tr>
<td></td>
<td>CHER2</td>
<td>ZHER2</td>
</tr>
<tr>
<td></td>
<td>cblas_ssspr2</td>
<td>cblas_dspr2</td>
</tr>
<tr>
<td></td>
<td>cblas_chpr2</td>
<td>cblas_zhprr2</td>
</tr>
<tr>
<td></td>
<td>cblas_ssyrr2</td>
<td>cblas_dsymv2</td>
</tr>
<tr>
<td></td>
<td>cblas_cher2</td>
<td>cblas_zher2</td>
</tr>
<tr>
<td>“SGBMV, DGBMV, CGBMV, and ZGBMV (Matrix-Vector Product for a General Band Matrix, Its Transpose, or Its Conjugate Transpose)” on page 336</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 343</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 347</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>
</tbody>
</table>
### Level 2 BLAS

<table>
<thead>
<tr>
<th>Descriptive Name</th>
<th>Short-Precision Subprogram</th>
<th>Long-Precision Subprogram</th>
</tr>
</thead>
<tbody>
<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 354</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>ZTRSV</td>
</tr>
<tr>
<td></td>
<td>CTRSV</td>
<td>cblas_stpsv</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cblas_ctpsv</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cblas_strsv</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cblas_ctrsv</td>
</tr>
<tr>
<td>“STBMV, DTBMV, CTBMV, and ZTBMV (Matrix-Vector Product for a Triangular Band Matrix, Its Transpose, or Its Conjugate Transpose)” on page 360</td>
<td>STBMV</td>
<td>DTBMV</td>
</tr>
<tr>
<td></td>
<td>CTBMV</td>
<td>ZTBMV</td>
</tr>
<tr>
<td></td>
<td>cblas_stbmv</td>
<td>cblas_dtbmv</td>
</tr>
<tr>
<td></td>
<td>cblas_ctbmv</td>
<td>cblas_ztbmv</td>
</tr>
<tr>
<td>“STBSV, DTBSV, CTBSV, and ZTBSV (Triangular Band Equation Solve)” on page 365</td>
<td>STBSV</td>
<td>DTBSV</td>
</tr>
<tr>
<td></td>
<td>CTBSV</td>
<td>ZTBSV</td>
</tr>
<tr>
<td></td>
<td>cblas_stbsv</td>
<td>cblas_dtbsv</td>
</tr>
<tr>
<td></td>
<td>cblas_ctbsv</td>
<td>cblas_ztbsv</td>
</tr>
</tbody>
</table>

### Level 3 BLAS

<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 413</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 422</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_cchemm</td>
<td>cblas_zhemm</td>
</tr>
<tr>
<td>“STRMM, DTRMM, CTRMM, and ZTRMM (Triangular Matrix-Matrix Product)” on page 430</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_ctrmm</td>
<td>cblas_ztrmm</td>
</tr>
<tr>
<td>“STRSM, DTRSM, CTRSM, and ZTRSM (Solution of Triangular Systems of Equations with Multiple Right-Hand Sides)” on page 437</td>
<td>STRSM</td>
<td>DTRSM</td>
</tr>
<tr>
<td></td>
<td>CTRSM</td>
<td>ZTRSM</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>
</tbody>
</table>
Table 271: 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>“SSYRK, DSYRK, CSYRK, ZSYRK, CHERK, and ZHERK (Rank-K Update of a Real or Complex Symmetric or a Complex Hermitian Matrix)” on page 445</td>
<td>SSYRK CSYRK CHERK cblas_ssyrk cblas_csyrk cblas_cherk</td>
<td>DSYRK ZSYRK ZHERK cblas_dsyrk cblas_zsyrk 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 451</td>
<td>SSYR2K CSYR2K CHER2K cblas_ssyr2k cblas_csyr2k cblas_cher2k</td>
<td>DSYR2K ZSYR2K ZHER2K cblas_dsyr2k cblas_zsyr2k 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:

  ```
  #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 179 for the required changes to your existing programs.

- For information about CBLAS calling sequences, see “Syntax” on page xxxi and [“11” on page 1253].
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 CGESV LAPACKE_sgesv LAPACKE_cgesv</td>
<td>DGESV ZGESV LAPACKE_dgesv LAPACKE_zgesv</td>
<td>“SGESV, DGESV, CGESV, ZGESV (General Matrix Factorization and Multiple Right-Hand Side Solve)” on page 480</td>
</tr>
<tr>
<td>SGETRF CGETRF LAPACKE_sgetrf LAPACKE_cgetrf</td>
<td>DGGETRF ZGETRF LAPACKE_dgetrf LAPACKE_zgetrf</td>
<td>“SGETRF, DGETRF, CGETRF and ZGETRF (General Matrix Factorization)” on page 484</td>
</tr>
<tr>
<td>SGETRS CGETRS LAPACKE_sgetrs LAPACKE_cgetrs</td>
<td>DGGETRS ZGETRS LAPACKE_dgetrs LAPACKE_zgetrs</td>
<td>“SGETRS, DGETRS, CGETRS, and ZGETRS (General Matrix Multiple Right-Hand Side Solve)” on page 488</td>
</tr>
<tr>
<td>SGECON CGECON LAPACKE_sgecon LAPACKE_cgecon</td>
<td>DGGECON ZGECON LAPACKE_dgecon LAPACKE_zgecon</td>
<td>“SGECON, DGECON, CGECON, and ZGECON (Estimate the Reciprocal of the Condition Number of a General Matrix)” on page 504</td>
</tr>
<tr>
<td>SGETRI CGETRI LAPACKE_sgetri LAPACKE_cgetri</td>
<td>DGGETRI ZGETRI LAPACKE_dgetri LAPACKE_zgetri</td>
<td>“SGETRI, DGETRI, CGETRI, ZGETRI, SGEICD, and DGEICD (General Matrix Inverse, Condition Number Reciprocal, and Determinant)” on page 511</td>
</tr>
<tr>
<td>SLANGE CLANGE LAPACKE_slange LAPACKE_clange</td>
<td>DLANGE ZLANGE LAPACKE_dlange LAPACKE_zlange</td>
<td>“SLANGE, DLANGE, CLANGE, and ZLANGE (General Matrix Norm)” on page 518</td>
</tr>
<tr>
<td>SPPSV CPPSV LAPACKE_sppsv LAPACKE_cppsv</td>
<td>DPPSV ZPPSV LAPACKE_dppsv LAPACKE_zppsv</td>
<td>“SPPSV, DPPSV, CPPSV, and ZPPSV (Positive Definite Real Symmetric and Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)” on page 521</td>
</tr>
<tr>
<td>SPOSV CPOSV LAPACKE_sposv LAPACKE_cposv</td>
<td>DPOSV ZPOSV LAPACKE_dposv LAPACKE_zposv</td>
<td>“SPOSV, DPOSV, CPOSV, and ZPOSV (Positive Definite Real Symmetric or Complex Hermitian Matrix Factorization and Multiple Right-Hand Side Solve)” on page 527</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>SPOTRF</td>
<td>DPOTRF</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 533</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_spotrf</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>
<tr>
<td>SPOTRS</td>
<td>DPOTRS</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 545</td>
</tr>
<tr>
<td>CPOTRS</td>
<td>ZPOTRS</td>
<td></td>
</tr>
<tr>
<td>SPPTRS</td>
<td>DPPTRS</td>
<td></td>
</tr>
<tr>
<td>CPPTRS</td>
<td>ZPPTRS</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_spptrrs</td>
<td>LAPACKE_dpptrrs</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpptrrs</td>
<td>LAPACKE_zpptrrs</td>
<td></td>
</tr>
<tr>
<td>SPOCON</td>
<td>DPPOCON</td>
<td>“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)” on page 555</td>
</tr>
<tr>
<td>CPOCON</td>
<td>ZPOCON</td>
<td></td>
</tr>
<tr>
<td>SPPCON</td>
<td>DPPOCON</td>
<td></td>
</tr>
<tr>
<td>CPPCON</td>
<td>ZPOCON</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_sppcon</td>
<td>LAPACKE_dppcon</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cppcon</td>
<td>LAPACKE_zppcon</td>
<td></td>
</tr>
<tr>
<td>SPOTRI</td>
<td>DPOTRI</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 569</td>
</tr>
<tr>
<td>CPOTRI</td>
<td>ZPOTRI</td>
<td></td>
</tr>
<tr>
<td>SPPTRI</td>
<td>DPPTRI</td>
<td></td>
</tr>
<tr>
<td>CPPTRI</td>
<td>ZPPTRI</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_spptr</td>
<td>LAPACKE_dpptr</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpptr</td>
<td>LAPACKE_zpptr</td>
<td></td>
</tr>
<tr>
<td>SLANSY</td>
<td>DLANSY</td>
<td>“SLANSY, DLANSY, CLANSY, ZLANSY, CLANHE, ZLANHE, SLANSP, DLANSP, CLANHP, and ZLANHP (Real Symmetric or Complex Hermitian Matrix Norm)” on page 579</td>
</tr>
<tr>
<td>&gt;</td>
<td>CLANSY</td>
<td>&lt;</td>
</tr>
<tr>
<td>CLANHE</td>
<td>ZLANHE</td>
<td></td>
</tr>
<tr>
<td>SLANSP</td>
<td>DLANSP</td>
<td></td>
</tr>
<tr>
<td>CLANHP</td>
<td>ZLANHP</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_slansy</td>
<td>LAPACKE_dlanhsy</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>LAPACKE_clansy</td>
<td>&lt;</td>
</tr>
<tr>
<td>LAPACKE_clanh</td>
<td>LAPACKE_zlanhe</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_slansp</td>
<td>LAPACKE_dlanhs</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_clanhp</td>
<td>LAPACKE_zlanhe</td>
<td></td>
</tr>
</tbody>
</table>
Table 272: 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>SSYSV</td>
<td>DSYSV</td>
<td>“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 584</td>
</tr>
<tr>
<td>CSYSV</td>
<td>ZSYSV</td>
<td></td>
</tr>
<tr>
<td>CHESV</td>
<td>ZHESV</td>
<td></td>
</tr>
<tr>
<td>SSPSV</td>
<td>DSPSV</td>
<td></td>
</tr>
<tr>
<td>CSPSV</td>
<td>ZSPSV</td>
<td></td>
</tr>
<tr>
<td>CHPSV</td>
<td>ZHPSV</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssysv</td>
<td>LAPACKE_dsytrf</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_csysv</td>
<td>LAPACKE_zsytrf</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chesv</td>
<td>LAPACKE_zhetrf</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_chpsv</td>
<td>LAPACKE_zhpshr</td>
<td></td>
</tr>
</tbody>
</table>

| SSYTRF                     | DSYTRF                    | “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 593 |
| CSYTRF                     | ZSYTRF                    |                                |
| CHETRF                     | ZHETRF                    |                                |
| SSPTRF                     | DSPTRF                    |                                |
| CSPTRF                     | ZSPTRF                    |                                |
| CHPTRF                     | ZHPTRF                    |                                |
| LAPACKE_ssytrf             | LAPACKE_dsytrf            |                                |
| LAPACKE_csytrf             | LAPACKE_zsytrf            |                                |
| LAPACKE_chetrf             | LAPACKE_zhetrf            |                                |
| LAPACKE_sspsv              | LAPACKE_dspsv             |                                |
| LAPACKE_cspsv              | LAPACKE_zspsv             |                                |
| LAPACKE_chpshr             | LAPACKE_zhpshr            |                                |

| SSYTRS                     | DSYTRS                    | “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 601 |
| CSYTRS                     | ZSYTRS                    |                                |
| CHETRS                     | ZHETRS                    |                                |
| SSPTRS                     | DSPTRS                    |                                |
| CSPTRS                     | ZSPTRS                    |                                |
| CHPTRS                     | ZHPTRS                    |                                |
| LAPACKE_ssytrs             | LAPACKE_dsytrf            |                                |
| LAPACKE_csytrs             | LAPACKE_zsytrf            |                                |
| LAPACKE_chetrs             | LAPACKE_zhetrs            |                                |
| LAPACKE_sspsv              | LAPACKE_dspsv             |                                |
| LAPACKE_cspsv              | LAPACKE_zspsv             |                                |
| LAPACKE_chpshr             | LAPACKE_zhpshr            |                                |

<p>| STRTRI                     | DTRTRI                    | “STRTRI, DTRTRI, CTRTRI, ZTRTRI, STPTRI, DTPTRI, CTPTRI, and ZTPTRI (Triangular Matrix Inverse)” on page 621 |
| STPTRI                     | DTPTRI                    |                                |
| CTRTRI                     | ZTRTRI                    |                                |
| CTPTRI                     | ZTPTRI                    |                                |
| LAPACKE_strtri             | LAPACKE_dtrtri            |                                |
| LAPACKE_stptri             | LAPACKE.dtptri            |                                |
| LAPACKE_ctri               | LAPACKE_ztrtri            |                                |
| LAPACKE_cptri              | LAPACKE_ztptri            |                                |</p>
<table>
<thead>
<tr>
<th>Short-Precision Subprogram</th>
<th>Long-Precision Subprogram</th>
<th>Descriptive Name and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLANTR CLANTR SLANTP CLANTP LAPACKE_slantr LAPACKE_clantr LAPACKE_slantp LAPACKE_clantp</td>
<td>DLANTR ZLANTR DLANTP ZLANTP LAPACKE_dlantr LAPACKE_zlantr LAPACKE_dlantp LAPACKE_zlantp</td>
<td>“SLANTR, DLANTR, CLANTR, ZLANTR, SLANTP, DLANTP, CLANTP, and ZLANTP (Trapezoidal or Triangular Matrix Norm)” on page 628</td>
</tr>
<tr>
<td>SGBSV CGBSV LAPACKE_sgbsv LAPACKE_cgbsv</td>
<td>DGBSV ZGBSV LAPACKE_dgbsv LAPACKE_zgbsv</td>
<td>“SGBSV, DGBSV, CGBSV, and ZGBSV (General Band Matrix Factorization and Multiple Right-Hand Side Solve)” on page 635</td>
</tr>
<tr>
<td>SGBTRF CGBTRF LAPACKE_sgbtrfs LAPACKE_cgbtrfs</td>
<td>DGBTRF ZGBTRF LAPACKE_dgbtrfs LAPACKE_zgbtrfs</td>
<td>“SGBTRF, DGBTRF, CGBTRF and ZGBTRF (General Band Matrix Factorization)” on page 639</td>
</tr>
<tr>
<td>SGBTRS CGBTRS LAPACKE_sgbtrss LAPACKE_cgbtrss</td>
<td>DGBTRS ZGBTRS LAPACKE_dgbtrss LAPACKE_zgbtrss</td>
<td>“SGBTRS, DGBTRS, CGBTRS, and ZGBTRS (General Band Matrix Multiple Right-Hand Side Solve)” on page 643</td>
</tr>
<tr>
<td>SPBSV CPBSV LAPACKE_spbsv LAPACKE_cpbsv</td>
<td>DPBSV ZPBSV LAPACKE_dpbsv LAPACKE_zpbsv</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 651</td>
</tr>
<tr>
<td>SPBTRF CPBTRF LAPACKE_spbtrfs LAPACKE_cpbtrfs</td>
<td>DPBTRF ZPBTRF LAPACKE_dpbftrs LAPACKE_zpbtrfs</td>
<td>“SPBTRF, DPBTRF, CPBTRF, and ZPBTRF (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Factorization)” on page 656</td>
</tr>
<tr>
<td>SPBTRS CPBTRS LAPACKE_spbtrss LAPACKE_cpbtrss</td>
<td>DPBTRS ZPBTRS LAPACKE_dpbftrs LAPACKE_zpbtrfs</td>
<td>“SPBTRS, DPBTRS, CPBTRS, and ZPBTRS (Positive Definite Real Symmetric or Complex Hermitian Band Matrix Multiple Right-Hand Side Solve)” on page 661</td>
</tr>
<tr>
<td>SGTSV CGTSV LAPACKE_sgtsv LAPACKE_cgtsv</td>
<td>DGTSV ZGTSV LAPACKE_dgtsv LAPACKE_zgtsv</td>
<td>“SGTSV, DGTSV, CGTSV, and ZGTSV (General Tridiagonal Matrix Factorization and Multiple Right-Hand Side Solve)” on page 666</td>
</tr>
<tr>
<td>SGTTRF CGTTRF LAPACKE_sgttrf LAPACKE_cgttrf</td>
<td>DGTTTRF ZGTTTRF LAPACKE_dgttrf LAPACKE_zgttrf</td>
<td>“SGTTRF, DGTTTRF, CGTTRF, and ZGTTTRF (General Tridiagonal Matrix Factorization)” on page 670</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>SGTTRS</td>
<td>DGTTTR</td>
<td>“SGTTRS, DGTTTR, CGTTRS, and ZGTTRS (General Tridiagonal Matrix Multiple Right-Hand Side Solve)” on page 674</td>
</tr>
<tr>
<td>CGTTRS</td>
<td>ZGTTTR</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgttrs</td>
<td>LAPACKE_dgttrs</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgttrs</td>
<td>LAPACKE_zgttrs</td>
<td></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 679</td>
</tr>
<tr>
<td>CPTSV</td>
<td>ZPTSV</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sptsv</td>
<td>LAPACKE_dptsv</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cptsv</td>
<td>LAPACKE_zptsv</td>
<td></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 684</td>
</tr>
<tr>
<td>CPTTRF</td>
<td>ZPTTRF</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spttrf</td>
<td>LAPACKE_dpttrf</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpttrf</td>
<td>LAPACKE_zpttrf</td>
<td></td>
</tr>
<tr>
<td>SPTTRS</td>
<td>DPTTRS</td>
<td>“SPTTRS, DPTTRS, CPTTRS, and ZPTTRS (Positive Definite Real Symmetric or Complex Hermitian Tridiagonal Matrix Multiple Right-Hand Solve)” on page 687</td>
</tr>
<tr>
<td>CPTTRS</td>
<td>ZPTTRS</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_spttrs</td>
<td>LAPACKE_dpttrs</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cpttrs</td>
<td>LAPACKE_zpttrs</td>
<td></td>
</tr>
<tr>
<td>SGESVD</td>
<td>DGESVD</td>
<td>“SGESVD, DGESVD, CGESVD, ZGESVD, SGESDD, DGESDD, CGESDD, and ZGESDD (Singular Value Decomposition for a General Matrix)” on page 794</td>
</tr>
<tr>
<td>CGESVD</td>
<td>ZGESVD</td>
<td></td>
</tr>
<tr>
<td>SGESDD</td>
<td>DGESDD</td>
<td></td>
</tr>
<tr>
<td>CGESDD</td>
<td>ZGESDD</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgesvd</td>
<td>LAPACKE_dgesvd</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgesvd</td>
<td>LAPACKE_zgesvd</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgesdd</td>
<td>LAPACKE_dgesdd</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgesdd</td>
<td>LAPACKE_zgesdd</td>
<td></td>
</tr>
<tr>
<td>SGEQRF</td>
<td>DGGEQRF</td>
<td>“SGEQR, DGEEQRF, CGEQR, and ZGEEQRF (General Matrix QR Factorization)” on page 807</td>
</tr>
<tr>
<td>CGEQRF</td>
<td>ZGEQRF</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgeqrf</td>
<td>LAPACKE_dgeqrf</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgeqrf</td>
<td>LAPACKE_zgeqrf</td>
<td></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 812</td>
</tr>
<tr>
<td>CGELS</td>
<td>ZGELS</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgels</td>
<td>LAPACKE_dgels</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgels</td>
<td>LAPACKE_zgels</td>
<td></td>
</tr>
<tr>
<td>SGELSD</td>
<td>DGELSD</td>
<td>“SGELSD, DGELSD, CGELSD, and ZGELSD (Linear Least Squares Solution for a General Matrix Using the Singular Value Decomposition)” on page 822</td>
</tr>
<tr>
<td>CGELSD</td>
<td>ZGELSD</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgelds</td>
<td>LAPACKE_dgelds</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgelds</td>
<td>LAPACKE_zgelds</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>SGEEV</td>
<td>DGEEV</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 850</td>
</tr>
<tr>
<td>CGEEV</td>
<td>ZGEEV</td>
<td></td>
</tr>
<tr>
<td>SGEEVX</td>
<td>DGEEVX</td>
<td></td>
</tr>
<tr>
<td>CGEEVX</td>
<td>ZGEEVX</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgeev</td>
<td>LAPACKE_dgeev</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgeev</td>
<td>LAPACKE_zgeev</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sgeevx</td>
<td>LAPACKE_dgeevx</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cgeevx</td>
<td>LAPACKE_zgeevx</td>
<td></td>
</tr>
<tr>
<td>SSYEV</td>
<td>DYSEV</td>
<td>“SSYEV, DSYEV, CHEEV, ZHEEV, SSPEVX, DSPEVX, CHPEVX, ZHPEVX, SSYEVX, DSYEVX, CHEEVX, ZHEEVX, SSYEVR, DSYEVR, CHEEVR, and ZHEEVR (Eigenvalues, and Optionally the Eigenvectors of a Real Symmetric or Complex Hermitian Matrix)” on page 867</td>
</tr>
<tr>
<td>CHEEV</td>
<td>ZHEEV</td>
<td></td>
</tr>
<tr>
<td>SSPEVX</td>
<td>DSPEVX</td>
<td></td>
</tr>
<tr>
<td>CHPEVX</td>
<td>ZHPEVX</td>
<td></td>
</tr>
<tr>
<td>SSYEVX</td>
<td>DSYEVX</td>
<td></td>
</tr>
<tr>
<td>CHEEVX</td>
<td>ZHEEVX</td>
<td></td>
</tr>
<tr>
<td>&gt;SSYEVX</td>
<td>&gt;DSYEVX</td>
<td></td>
</tr>
<tr>
<td>CHEEVR&lt;</td>
<td>ZHEEVR&lt;</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssyev</td>
<td>LAPACKE_dsyev</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cheev</td>
<td>LAPACKE_zheev</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sspevx</td>
<td>LAPACKE_dspevx</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chpevx</td>
<td>LAPACKE_zhpevx</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssyevx</td>
<td>LAPACKE_dsyevx</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cheevx</td>
<td>LAPACKE_zheevx</td>
<td></td>
</tr>
<tr>
<td>&gt;LAPACKE_ssyevr</td>
<td>&gt;LAPACKE_dsyevr</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cheevr&lt;</td>
<td>LAPACKE_zheevr&lt;</td>
<td></td>
</tr>
<tr>
<td>SSPEVD</td>
<td>DSPEVD</td>
<td>“SSPEVD, DSPEVD, CHPEVD, ZHPEVX, SSYEVX, DSYEVX, CHEEVX, ZHEEVX, SSYEVR, DSYEVR, CHEEVR, and ZHEEVR (Eigenvalues and, Optionally the Eigenvectors, of a Real Symmetric or Complex Hermitian Matrix Using a Divide-and-Conquer Algorithm)” on page 893</td>
</tr>
<tr>
<td>CHPEVD</td>
<td>ZHPEVD</td>
<td></td>
</tr>
<tr>
<td>SSYEVD</td>
<td>DSYEVX</td>
<td></td>
</tr>
<tr>
<td>CHEEVD</td>
<td>ZHEEVX</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sspevd</td>
<td>LAPACKE_dspevd</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_chpevd</td>
<td>LAPACKE_zhpevd</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_ssyevd</td>
<td>LAPACKE_dsyevd</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cheevd</td>
<td>LAPACKE_zheevd</td>
<td></td>
</tr>
<tr>
<td>&gt;LAPACKE_ssyevrd</td>
<td>&gt;LAPACKE_dsyevrd</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cheevrd&lt;</td>
<td>LAPACKE_zheevrd&lt;</td>
<td></td>
</tr>
<tr>
<td>SGGEV</td>
<td>DGGEV</td>
<td>“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 905</td>
</tr>
<tr>
<td>CGGEV</td>
<td>ZGGEV</td>
<td></td>
</tr>
<tr>
<td>SGGEVX</td>
<td>DGGEVX</td>
<td></td>
</tr>
<tr>
<td>CGGEVX</td>
<td>ZGGEVX</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sggev</td>
<td>LAPACKE_dggev</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cggev</td>
<td>LAPACKE_zggev</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_sggevX</td>
<td>LAPACKE_dggevX</td>
<td></td>
</tr>
<tr>
<td>LAPACKE_cggevX</td>
<td>LAPACKE_zggevX</td>
<td></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:

  ```c
  #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 datatypes 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 xxxi
Using LAPACK with ESSL

ESSL contains a subset of LAPACK subroutines. If you want to use some of the LAPACK subroutines that are not included in ESSL, you can create a modified LAPACK library, in which the existing LAPACK subroutines that are contained in ESSL are removed.

The following example script shows the steps to create a modified Netlib LAPACK library:

```bash
#!/bin/bash
# Assumptions
#  liblapack.a is in the current directory
#  ESSL libs are in /usr/lib64
#
# First Pass - get the symbols in ESSL and LAPACK
nm liblapack.a | grep " T " | awk '{ print $3 }' | sort -n > symbols.in.lapack
nm /usr/lib64/libessl.so | grep " T " | awk '{ print $3 }' | sort -n > symbols.in.essl
diff -y symbols.in.lapack symbols.in.essl | grep -v -e '>' -e '|' -e '<' | awk '{ print $1 }' > duplicate.symbols
wc -l duplicate.symbols

# Second Pass - remove the ESSL symbols from the LAPACK library
cp -p liblapack.a liblapackforessl.a
for SYMBOL in $(cat duplicate.symbols | awk '{ print $1 }')
do
    fn=$SYMBOL
    unders="no"
    echo $SYMBOL > /dev/null && unders="yes"
    if [ "$unders" = "yes" ]
    then
        fn=`echo $SYMBOL | awk -F'_' '{print $1}'`
    fi
    ar d liblapackforessl.a $fn.o
done

# Check it worked
nm liblapackforessl.a | grep " T " | awk '{ print $3 }' | sort -n > symbols.in.updatedlapack
diff -y symbols.in.updateclapack symbols.in.essl | grep -v -e '>' -e '|' -e '<' | awk '{ print $1 }' > duplicate.symbols.inupdatedlapack
wc -l duplicate.symbols.inupdatedlapack
```

1238 IBM Engineering and Scientific Subroutine Library for Linux on POWER Version 6.1: ESSL Guide and Reference
Appendix C. FFTW Version 3.1.2 to ESSL Wrapper Libraries

The following information 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 file:

/opt/ibmmath/essl/version.release/FFTW3/README

C and Fortran Wrappers

The following tables list the 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>fftw_execute</td>
<td>DFFTW_EXECUTE</td>
</tr>
<tr>
<td></td>
<td>fftwf_execute</td>
<td>SFFTW_EXECUTE</td>
</tr>
<tr>
<td></td>
<td>fftw_destroy_plan</td>
<td>DFFTW_DESTROY_PLAN</td>
</tr>
<tr>
<td></td>
<td>fftwf_destroy_plan</td>
<td>SFFTW_DESTROY_PLAN</td>
</tr>
<tr>
<td></td>
<td>fftw_cleanup</td>
<td>DFFTW_CLEANUP</td>
</tr>
<tr>
<td></td>
<td>fftwf_cleanup</td>
<td>SFFTW_CLEANUP</td>
</tr>
<tr>
<td>Basic interface (Complex DFTs)</td>
<td>fftw_plan_dft_1d</td>
<td>DFFTW_PLAN_DFT_1D</td>
</tr>
<tr>
<td></td>
<td>fftwf_plan_dft_1d</td>
<td>SFFTW_PLAN_DFT_1D</td>
</tr>
<tr>
<td></td>
<td>fftw_plan_dft_2d</td>
<td>DFFTW_PLAN_DFT_2D</td>
</tr>
<tr>
<td></td>
<td>fftwf_plan_dft_2d</td>
<td>SFFTW_PLAN_DFT_2D</td>
</tr>
<tr>
<td></td>
<td>fftw_plan_dft_3d</td>
<td>DFFTW_PLAN_DFT_3D</td>
</tr>
<tr>
<td></td>
<td>fftwf_plan_dft_3d</td>
<td>SFFTW_PLAN_DFT_3D</td>
</tr>
<tr>
<td></td>
<td>fftw_plan_dft</td>
<td>DFFTW_PLAN_DFT</td>
</tr>
<tr>
<td></td>
<td>fftwf_plan_dft</td>
<td>SFFTW_PLAN_DFT</td>
</tr>
<tr>
<td>Category</td>
<td>C Wrapper</td>
<td>Fortran Wrapper</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>------------------------------------</td>
<td>----------------------------------</td>
</tr>
</tbody>
</table>
| **Basic interface** (Real-data DFTs) | fftw_plan_dft_r2c_1d  
fftw_plan_dft_r2c_2d  
fftw_plan_dft_r2c_3d  
fftw_plan_dft_r2c     | DFFTW_PLAN_DFT_R2C_1D  
SFFTW_PLAN_DFT_R2C_1D  
DFFTW_PLAN_DFT_R2C_2D  
SFFTW_PLAN_DFT_R2C_2D  |
|                                  | fftwf_plan_dft_r2c_1d  
fftw_plan_dft_r2c_2d  
fftw_plan_dft_r2c_3d  
fftw_plan_dft_r2c     | DFFTW_PLAN_DFT_R2C_1D  
SFFTW_PLAN_DFT_R2C_1D  
DFFTW_PLAN_DFT_R2C_2D  
SFFTW_PLAN_DFT_R2C_2D  |
| **Advanced interface** (Complex DFTs) | fftw_plan_many_dft  
fftw_plan_many_dft     | DFFTW_PLAN_MANY_DFT  
SFFTW_PLAN_MANY_DFT    |
| **Advanced interface** (Real-data DFTs) | fftw_plan_many_dft_r2c  
fftw_plan_many_dft_r2c  
fftw_plan_many_dft_c2r  
fftw_plan_many_dft_c2r     | DFFTW_PLAN_MANY_DFT_R2C  
SFFTW_PLAN_MANY_DFT_R2C  
DFFTW_PLAN_MANY_DFT_C2R  
SFFTW_PLAN_MANY_DFT_C2R    |
| **Guru interface** (Complex DFTs) | fftw_plan_guru_dft  
fftw_plan_guru_dft  
fftw_execute_dft  
fftw_execute_dft     | DFFTW_PLAN_GURU_DFT  
SFFTW_PLAN_GURU_DFT  
DFFTW_EXECUTE_DFT  
SFFTW_EXECUTE_DFT     |
| **Guru interface** (Real-data DFTs) | fftw_plan_guru_dft_r2c  
fftw_plan_guru_dft_r2c  
fftw_plan_guru_dft_c2r  
fftw_plan_guru_dft_c2r  
fftw_execute_dft_r2c  
fftw_execute_dft_r2c  
fftw_execute_dft_c2r  
fftw_execute_dft_c2r     | DFFTW_PLAN_GURU_DFT_R2C  
SFFTW_PLAN_GURU_DFT_R2C  
DFFTW_PLAN_GURU_DFT_C2R  
SFFTW_PLAN_GURU_DFT_C2R  
DFFTW_EXECUTE_DFT_R2C  
SFFTW_EXECUTE_DFT_R2C  
DFFTW_EXECUTE_DFT_C2R  
SFFTW_EXECUTE_DFT_C2R     |
| **Memory allocation**          | fftw_malloc  
fftwf_malloc  
fftw_free  
fftwf_free     | (Not applicable)             |

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 32-bit integer, 64-bit pointer environment.

> | Global subroutine names with or without trailing underscores are supported for the Fortran wrappers. |
|<

For additional information about how to use the FFTW Wrapper libraries, see Chapter 5, “Processing Your Program,” on page 169.

Building the FFTW Wrapper libraries

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: 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

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
   - 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, 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/6.1/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/libesslsmp.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/libesslsmpcuda.so /usr/lib64/libblas.so.3
  ```

Set the options as suggested “Using the ESSL SMP CUDA Library” on page 42 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:
```java
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:

```bash
javac –cp <PATH_TO_netlib-java Jar files> ddot_example.java
java –cp <PATH_TO_netlib-java Jar files>: ddot_example
```

Figure 17 on page 1244 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

**Scalca 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:

```scala
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)
} else {
    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 on page 1245 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)
x: Int = 5
y: 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.

```
#!/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) | nx: Int = 5 |
| ny: 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 that uses standard Microsoft Windows navigation keys.
• 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.

The ESSL documentation that is located in the IBM Knowledge Center is accessibility-enabled. The accessibility features of IBM Knowledge Center are described in the Accessibility topic.

IBM and accessibility
For more information about the commitment that IBM has to accessibility, see IBM Accessibility.
Notices

This information was developed for products and services offered in the US.

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, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (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.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

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 jurisdictions 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 websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide 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 Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

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

© Copyright IBM Corp. 2018
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.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

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.

Statements regarding IBM’s future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM’s suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

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 actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate 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.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© IBM 2018

Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. 2018

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 at www.ibm.com/legal/copytrade.shtml.

Linux is a registered 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.

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.
Privacy policy considerations

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as the customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM’s Privacy Policy at http://www.ibm.com/privacy and IBM’s Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled “Cookies, Web Beacons and Other Technologies” and the “IBM Software Products and Software-as-a-Service Privacy Statement” at http://www.ibm.com/software/info/product-privacy.
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.

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.


3. This is the new one


For more information, see:

http://www.netlib.org/lapack/index.html


You can download this document from the following URL:

http://www.netlib.org/lapack/lawns/lawn03.ps


You can download this document from:

http://www.netlib.org/lapack/lawns/lawn162.pdf


You can download this document from:

http://www.netlib.org/benchmark/performance.ps


For more information, see:

http://www.netlib.org/linpack/index.html


50. 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


60. EISPACK software library; National Energy Software Center, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439 (312-972-7250); International Mathematical and Statistical Libraries, Inc., Sixth Floor, GNB Building, 7500 Bellaire Boulevard, Houston, Texas 77036 (713-772-1927)

61. 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.


106. Parlett, B.; Marques, O. "An implementation of the dqds Algorithm (Positive Case)," *LAPACK Working Note 155*.

You can download this document from:

http://www.netlib.org/lapack/lawns/lawn155.ps


Index

Special Characters

<complex> or <complex.h> header file
Linux 159

Numerics

3838 Array Processor
3838 Array Processor
   general signal processing routines 955

A

abbreviations
   for product names xxiv
   interpreting math and programming xxix
absolute value
   notation xxix
   sum of all absolute values 214
accuracy
   considerations for dense and banded linear algebraic equations 477
   considerations for eigensystem analysis 849
   considerations for Fourier transforms, convolutions, and correlations 958
   considerations for interpolation 1125
   considerations for linear algebra subprograms 203
   considerations for matrix operations 385
   considerations for numerical quadrature 1143
   considerations for related computations 961
   considerations for sorting and searching 1109
   error of computation 61
   of results 4, 60
   precisions 61
   what accuracy means 60
   where to find information on 60
acronyms
   associated with programming values xxix
   product names xxiv
adding
   absolute values 214
address notation xxix
Announcing ESSL brochure 1253
algebra 467
applications in the industry 2
arguments
   coding rules 48
   conventions used in the subroutine descriptions xxxiii
   diagnosing ESSL input-argument errors 184
   font for ESSL calling xxv
   list of ESSL input-argument errors 187, 193
   passing in C programs 143
   passing in C++ programs 156
array
   coding in C programs 146
   coding in C++ programs 159
   coding in Fortran programs 126
array (continued)
   conventions for xxviii
   real and complex elements 126
   setting up data structures inside 71
   storage techniques overview 47
array data
   storage and performance tradeoffs 62
arrow notation, what it means xxix
attention notation xxix
attention error messages, interpreting 186
autocorrelation of one or more sequences 1083
auxiliary working storage
   calculating 51
   dynamic allocation 51
   list of subroutines using 49
   provided by the user 51
B

background books 1253
band matrix
   definition of 94
   storage layout 95–97, 100
band matrix subroutines, names of 467
band width 94, 99
banded linear algebraic equation subroutines
   SGTSV, DGTSV, CGTSV, and ZGTSV 666
   SPTF and DPTF 716
   SPTSV, DPTSV, CPTSV, and ZPTSV 679
   SPTTRF, DPTTRF, CPTTRF, and ZPTTRF 684
   SPTTRS, DPTTRS, CPTTRS, and ZPTTRS 687
   STBSV, DTBSV, CTBSV, and ZTBSV 365
BLAS (Basic Linear Algebra Subprograms)
   ESSL subprograms 197, 1225
   migrating from 181
   migrating to ESSL 33
BLAS-general-band storage mode 97
bold letters, usage of xxv
books 1253

C

C (C programming language)
   coding programs 142
   ESSL header file 142, 145
   function reference 142
   handling errors in your program 147
   how to code arrays 146
   modifying procedures for using ESSL for Linux 171
   passing character arguments 143
   program calling interface 142
   setting up complex and logical data 145
C and C++
   publications xxiii
C++ (C++ programming language)
   coding programs 155
   ESSL header file 156
C++ (C++ programming language) (continued)
function reference 155
how to code arrays 159
passing character arguments 156
program calling interface 155
calculating auxiliary working storage 51
calculating transform lengths 56
CALL statement 125
calling sequence
for C programs 142
for C++ programs 155
for Fortran programs 125
specifying the arguments 48
subroutines versus functions 125, 142, 155
syntax description xxxi
CDOTC 225
CDOTCI 292
CDOTU 225
CDOTUI 292
calling sequence and meaning xxix
CGECMI 459
CGEQRF 807
CGESM 499
CGESUB 392
CGETMI 459
CGETRF 484
character data
conventions xxvi, 47
characters, special usage of xxix
CHEEVX 867
CHEGV 931
CHEQVD 922
CHEGVX 931
CHEMM 422
CHEMV 313
CHER2 328
CHER2K 451
CHERK 445
choosing the ESSL library 33
choosing the ESSL subroutine 33
CHPEVX 867
CHPGVX 931
CHPMV 313
CHPR2 328
citations 1253
CLANGE 518
CLANHE 579
CLANHP 579
CLANSY 579
CLANTP 628
CLANTR 628
CNORM2 241
coding your program (continued)
techniques that affect performance 62
comparison of accuracy for libraries 4
compilers, required by ESSL on the workstations 6
compiling your program
C programs 171
Fortran programs 170
calculation and real array elements 126
complex conjugate notation xxix
complex data
citations 1253
conventions xxvi, 47
Linux 145
setting up for C 145
complex Hermitian band matrix
definition of 101
complex Hermitian matrix
definition of 84
storage layout 84
complex Hermitian Toeplitz matrix
definition of 87
complex Hermitian tridiagonal matrix
definition of 109
storage layout 109
complex matrix 77
compressed-vector, definition and storage mode 76
computational areas, overview 2
computational errors
diagnosing 185
list of messages for 191
overview 65
conjugate notation xxix
conjugate transpose of matrix operation results for multiply 402, 412, 418
conjugate transpose of a matrix 77
conjugate transpose of a vector 72
continuation, convention for numerical data xxvi
conventions
for messages 186
mathematical and programming notations xxix
subroutine descriptions xxxi
convolution and correlation
autocorrelation of one or more sequences 1083
one sequence with one or more sequences 1060
convolution and correlation subroutines
accuracy considerations 958
performance and accuracy considerations 960
performance considerations 958
SACOR 1083
SCON and SCOR 1060
usage considerations 955
correlation 1060
cosine notation xxix
courier font usage xxv
CPOCON 555
CPOSM 545
CPOTRS 545
CPPCON 555
CPPSV 521
CPPTR5 545
CPTSV 679
CPTTRF 684
CPTTRS 687
CSYMM 422
CSYR2K 451
CSYRK 445
CTBMV 360
CTBSV 365
CTPMV 347
CTPSV 354
CTPTRI 621
CTRMM 430
CTRMV 347
CTRSM 437
CTRSV 354
CTRTRI 621
customer service, IBM 183
customer support, IBM 183
CWLEV 1102

D

DASUM 214
data

conventions for scalar data xxvi, 47
data structures (vectors and matrices) 71
DDOT 225
DDOTI 292
default values in the ESSL error option table 67
dense and banded subroutines

performance and accuracy considerations 477
dense linear algebraic equation subroutines

SGESM, DGESM, CGESM, and ZGESM 499
SGETF, DGETF, CGETF and ZGETF 484
SPOSM, DPOSM, CPOSM, ZPOSM, SPPTRS, DPPTRS,
CPPTRS, ZPPTRS, SPOTRS, DPOTRS, CPOTRS, and
ZPOTRS 545
SPPSV, DPPSV, CPPSV, ZPPSV 521
SSYSV, DSYSV, CSYSV, ZSYSV, CHESV, ZHESV, SSPSV,
DSPSV, CSPSV, ZSPSV, CHPSV, ZHPSV 584
SSYTRF, DSYTRF, CSYTRF, ZSYTRF, CHETRF, ZHETRF,
SSPTRF, DSPPTRF, CSPTRF, ZSPTRF, CHPTRF, ZHPTRF
593
SSYTRS, DSYTRS, CSYTRS, ZSYTRS, CHTRS, ZHTRS,
SSPTRS, DPSPTRS, CPSPTRS, ZSPTRS, CHPTRS, ZHPTRS
601
STPSV, DTPSV, CTTPSV, and ZTPSV 354
STRSM, DTRSM, CTRSM, and ZTRSM 437
STRSV, DTRSV, CTRSV, and ZTRSV 354
STRTRI, DTRTRI,CTRTRI, ZTRTRI, STPTRI, DTPTRI,
CTPTRI, ZTPTRI 621
dense matrix, definition 110
descriptions, conventions used in the subroutine xxxi
designing your program

accuracy of results 60
choosing the ESSL library 33
choosing the ESSL subroutine 33
error considerations 64
performance considerations 62
storage considerations 47
determinant

matrix notation xxix
DGELS 812
DGELSD 822
DGQRF 807
DGEMS 499
DGESUB 392
DGEMI 459
DGETRF 484
DGEMQ 1161
DGELQ 1155
DGELQ 1147
DGELQ2 1149
DGELQ 1158
diagnosis procedures

attention error messages 186
computational errors 185
ESSL messages, list of 186
in your program 183
informational error messages 186
initial problem diagnosis procedures (symptom index) 184
input-argument errors 184
miscellaneous error messages 186
program exceptions 184
resource error messages 185
diagnostics 186
dimensions of arrays

storage layout 126
direct sparse matrix solvers

usage considerations 477
DLAMCH 1210
DLANGE 518
DLANSP 579
DLANSY 579
DLANTP 628
DLANTR 628
DNDOT 233
DNRM2 241
DNRM2 238
DNRNRM 1173
dot product

notation xxix
of dense vectors 225
of sparse vectors 292
special (compute N times) 233
DPOCON 555
DPOS 545
DPOTRS 545
DPPOCON 555
DPSSV 521
DPPTRS 545
DPTF 716
DPTSV 679
DPTTRF 684
DPTTRRS 687
DRCTF2 1028
DROTMG 254
DSKTRN 1219
DSLMX 313
DSLX 328
DSPMV 867
DSPLMV 313
DSPLMV 313
DSYEVX 867
DSYGV 931
DSYGVX 931
DSYGVX 931
DSYMM 422
DSYMV 313
DSYPR 328
DSYQR 328
DSYQR 341
DSYQR 451
DSYQR 445
DTBMV 360
DTBSV 365
DTPMV 347
DTPSV 354
DTPTRI 621
DTRMM 430
DTRMV 347
DTRSM 437
DTRSV 354
DTRTRI 621
DWLEV 1102
dynamic allocation of auxiliary working storage 51
DZASUM 214
DZNRM2 238

E
efficiency of your program 62
eigensystem analysis subroutines
  performance and accuracy considerations 849
  SSYEV, DSYEV, CHEEV, ZHEEV, SSPEVX, DSPEVX,
  CHPEVX, ZHPEVX, SSYEV, DSYEV, CHEV, ZHEEV,
  SSYEVX, DSYEVX, CHEEVX, and ZHEEVX 867
eigenvalues and eigenvectors
  complex Hermitian matrix 867
  real symmetric matrix 867
EINFO, ESSL error information-handler
  considerations when designing your program 65
  subroutine description 1188
  using EINFO in C programs 147
  using EINFO in C++ programs 161
  using EINFO in Fortran programs 134
element of a matrix notation xxix
element of a vector notation xxix
error conditions, conventions used in the subroutine
descriptions xxiv
error messages 186
error option table default values 67
error-handling subroutines
  EINFO 1188
  ERRSAV 1190
  ERRSET 1191
  ERRSTR 1192
errors
  attention error messages, interpreting 186
  attention messages, overview 67
  calculating auxiliary storage 49
  computational errors 65, 185
  EINFO subroutine description 1188
  handling errors in your C program 147
  handling errors in your Fortran program 133
  how errors affect output 64
  informational error messages, interpreting 186
  input-argument errors 184
  input-argument errors, overview 64
  miscellaneous error messages, interpreting 186
  overview of 64
  program exceptions 64, 184
  resource error messages, interpreting 185
  resource errors, overview 66
  types of errors that you can encounter 64
  using ERRSAV and ERRSTR 69
  values returned for EINFO error codes 1188
errors (continued)
  when to use ERRSET 67
  where to find information on 64
ERRSAV
  subroutine description 1190
  using with large applications 69
ERRSET
diagnosis procedures using 185
  ESSL default values for 67
  handling errors in C 147
  subroutine description 1191
  using EINFO in C programs 147
  using EINFO in C++ programs 161
  using EINFO in Fortran programs 134
  using ERRSET, EINFO, and return codes in C 147
  using ERRSET, EINFO, and return codes in C++ 161
  using ERRSET, EINFO, and return codes in Fortran 134
  when to use 65, 67
  when to use ERRSAV and ERRSTR with ERRSET 69
ERRSTR
diagnosis procedures for ESSL errors 183
  eigensystem analysis subroutines 847
  error option table default values 67
  Fourier transform, convolutions and correlations, and related-computation subroutines 953
  functional capability 2
  informational error messages, interpreting 186
  input-argument errors, diagnosing 184
  input-argument errors, overview 64
  installation requirements 6
  interpolation subroutines 1125
  languages supported 5
  linear algebra subprograms 1197
  linear algebraic equations subroutines 467
  matrix operation subroutines 383
  message conventions 186
  messages, list of 186
  migrating from one IBM hardware platform to another 180
  migrating programs 179
  migrating to future releases or future hardware 180
  miscellaneous error messages, interpreting 186
  name xxiv
  number of subroutines in each area 2
  numerical quadrature subroutines 1143
  ordering publications 1253
  overview of the subroutines 2
  packaging characteristics 6
  parallel processing subroutines on the workstations 2
  program exceptions 64
  program number for 1253
  publications overview 1253
  random number generation subroutines 1165
  reference information conventions xxxi

1264
ESSL (Engineering and Scientific Subroutine Library) (continued)

related publications 1253
resource error messages, interpreting 185
resource errors, overview 66
setting up your data structures 71
sorting and searching subroutines 1109
when coding large applications 69
when to use ERRSET for ESSL errors 67

ESSL libraries
Blue Gene 47
shared 47

ESSL messages 186
Euclidean length
with no scaling of input 241
with scaling of input 238
Euclidean norm notation xxix
examples of matrices 77
examples, conventions used in the subroutine descriptions xxxiv
exponential function notation xxix
expressions, special usage of xxix
extended error-handling subroutines
handling errors in C 147
handling errors in your Fortran program 133
how they work 64, 69
using them in diagnosing problems 184
extended-error-handling subroutines, using in your C program 147
in your Fortran program 133

F
factor and solve
general tridiagonal matrix 666
factoring
general matrix 484
indefinite
complex Hermitian matrix 584, 593
complex symmetric matrix 584, 593
real symmetric matrix 584, 593
positive definite
complex Hermitian matrix 521
real symmetric matrix 521
symmetric tridiagonal matrix 716
tridiagonal matrix 684
fast Fourier transform (FFT) 960
FFT 960
floor notation and meaning xxix
fonts used xxv
formula for transform lengths, interpreting 57
formulas for auxiliary storage, interpreting 51
Fortran
languages required by ESSL for Linux on the workstations 6
modifying procedures for using ESSL 170
Fortran considerations
coding programs 125
function reference 197
handling errors in your program 134
Fortran function reference 125
Fortran program calling interface 125
Fourier transform
two dimensions
real-to-complex 1028

Fourier transform subroutines
accuracy considerations 958
how they achieve high performance 960
performance considerations 958
terminology used for 955
usage considerations 955

Frobenius norm notation xxix
full-matrix storage mode 110
full-vector, definition and storage mode 76
function
calling sequence in C programs 142
calling sequence in C++ programs 155
calling sequence in Fortran programs 125
function reference 197
functional capability of the ESSL subroutines 2
functional description, conventions used in the subroutine descriptions xxxiii
functions, ESSL 197
future migration considerations 180

G
Gaussian quadrature methods
Gauss-Hermite Quadrature 1161
Gauss-Laguerre Quadrature 1155
Gauss-Legendre Quadrature 1147
Gauss-Rational Quadrature 1158
two-dimensional Gauss-Legendre Quadrature 1149
general matrix 467
general matrix subroutines, names of 467
general tridiagonal matrix
definition of 106
storage layout 106
general-band storage mode 96
generation of random numbers 1165
globalization 184
greek letters notation xxix
guidelines for handling problems 183

H
half band width 99
handling errors
in your C program 147
in your Fortran program 133
header file, ESSL, for C 142, 145
header file, ESSL, for C++ 156
Hermitian band matrix
definition of 101
Hermitian matrix
definition of 84
definition of, complex 87
storage layout 84

I
i-th zero crossing 955
IBM publications 1253
IBM Request for Enhancement (RFE) Community xxiv
identify the GPUs ESSL should use 1194
identifying problems 184

1265
indefinite complex Hermitian matrix
  definition of 85
  storage layout 86
indefinite symmetric matrix
  definition of 84
  storage layout 84
industry areas 2
infinity notation xxix
informational error messages, interpreting 186
informational messages, for ESSL 186
input arguments, conventions used in the subroutine descriptions xxxiii
input data, conventions for 47
input-argument errors
diagnosing 184
  list of messages for 187, 193
overview 64, 66, 67
int notation and meaning xxix
integer data
  conventions xxvi, 47
integral notation xxix
interface, ESSL
  for C programs 142
  for C++ programs 155
  for Fortran programs 125
interpolation subroutines
  accuracy considerations 1125
  performance considerations 1125
  usage considerations 1125
inverse
  matrix notation xxix
  triangular matrix 621
italic font usage xxv
iterative linear system solver
  usage considerations 479

L
l(2) norm
  with no scaling of input 241
  with scaling of input 238
languages supported by ESSL 5
LAPACK and LAPACKE
  ESSL subprograms 1231
LAPACK, migrating from 180
leading dimension for matrices
  how it is used for matrices 78
  how it is used in three dimensions 123
least squares solution
  CGEQRF 807
  DGEQRF 807
  SGEQRF 807
  ZGEQRF 807
letters, fonts of xxv
level of ESSL, getting 1193
library
  migrating from a non-IBM 181
  migrating from ESSL for Linux on Power Version 5 Release 3.2 to Version 5 Release 4 179
  migrating from ESSL for Linux on Power Version 5 Release 4 to Version 5 Release 5 179
  migrating from ESSL for Linux on Power Version 5 Release 5 to Version 6 Release 1 179
library (continued)
  migrating from LAPACK 180
overview 2
Licensed Program Specification, ESSL 1253
linear algebra 467
linear algebra subprograms
  accuracy considerations 203
  list of matrix-vector linear algebra subprograms 200
  list of sparse matrix-vector linear algebra subprograms 202
  list of sparse vector-scalar linear algebra subprograms 200
  list of vector-scalar linear algebra subprograms 197
overview 197
  performance considerations 203
  usage considerations 203
linear algebraic equation subroutines
  CLANGE 518
  CLANHE 579
  CLANHP 579
  CLANSY 579
  CLANTP 628
  CLANTR 628
  CPOCON 555
  CPPCON 555
  DLANGE 518
  DLANSY 579
  DLANTP 628
  DLANTR 628
  DPOCON 555
  DPPCON 555
  SLANGE 518
  SLANSP 579
  SLANTR 518
  SPOCON 555
  SPPCON 555
  ZLANGE 518
  ZLANHE 579
  ZLANHP 579
  ZLANSY 579
  ZLANTP 628
  ZLANTR 628
  ZPOCON 555
  ZPPCON 555
linear algebraic equations
  accuracy considerations 477
  list of banded linear algebraic equation subroutines 472
  list of dense linear algebraic equations 467
  list of linear least squares subroutines 475
  list of sparse linear algebraic equation subroutines 474
overview 467
  performance considerations 477
  usage considerations 477
linear least squares solution
  QR factorization 812, 822
linear least squares subroutines
  CGEQRF 807
  DGELS 812
  DGELSD 822
  DGESRF 807
  SGELS 812
linear least squares subroutines (continued)
SGELSD 822
SGEQRF 807
ZGELS 812
ZGELSD 822
ZGEQRF 807
linking
C programs 171
Fortran programs 170
Linux
<complex> or <complex.h> header file 159
Fortran program procedures 170
logical data
conventions xxvi, 47
setting up for C 145
long precision
accuracy statement 4
meaning of 61
lower band width 94
lower storage mode 80
lower-band-packed storage mode 100
lower-packed storage mode 80
lower-storage-by-rows for symmetric sparse matrices 115
lower-trapezoidal storage mode 92
lower-trapezoidal-packed storage mode 92
lower-triangular storage mode 89
lower-triangular-band-packed storage mode 104
lower-triangular-packed storage mode 89
matrix (continued)
symmetric tridiagonal matrix 107
Toeplitz matrix 86
trapezoidal matrices 91
triangular band matrices 102
triangular matrices 88
matrix operation subroutines
accuracy considerations 385
performance considerations 385
SGESUB, DGESUB, CGESUB, and ZGESUB 392
SGETMI, DGETMI, CGETMI, ZGETMI, CGECMI, and ZGETMI 459
SSYM, DSYMM, CSYMM, ZSYMM, CHEMM, and ZHEMM 422
SSYR2K, DSYR2K, CSYR2K, ZSYR2K, CHER2K, and ZHER2K 451
SSYRK, DSYRK, CSYRK, ZSYRK, CHERK, and ZHERK 445
STMM, DTRMM, CTRMM, and ZTRMM 430
usage considerations 384
matrix-vector product
complex Hermitian matrix 422
complex symmetric matrix 422
real symmetric matrix 422
triangular matrix 430
matrix-vector linear algebra subprograms
SSPMV, DSPMV, CHPMV, ZHPMV, SSYMV, DSYMV, CHEMV, ZHEMV, SSLMX and DSLMX 313
SSPR2, DSPR2, CHPR2, ZHPR2, SSYR2, DSYR2, CSYR2, ZSYR2, CHER2, ZHER2, SSLR2, and DSLR2 328
STBMV, DTBMV, CTBMV, and ZTBMV 360
STPMV, DTPMV, CTPMV, ZTPMV, STRMV, DTRMV, CTRMV, and ZTRMV 347
matrix-vector product
complex Hermitian matrix 313
real symmetric matrix 313
triangular band matrix, its transpose, or its conjugate transpose 360
triangular matrix, its transpose, or its conjugate transpose 347
max notation and meaning xxix
messages
ESSL and attention messages, interpreting 186
ESSL informational messages, interpreting 186
ESSL miscellaneous messages, interpreting 186
ESSL resource messages, interpreting 185
list of ESSL messages 187, 193
message conventions 186
migrating
from LAPACK 180
from non-IBM libraries 181
from one IBM hardware platform to another 180
future migration considerations 180
migrating from ESSL for Linux on Power Version 5 Release 3.2 to Version 5 Release 4 179
migrating from ESSL for Linux on Power Version 5 Release 4 to Version 5 Release 5 179
programs to ESSL 179
min notation and meaning xxix
miscellaneous error messages, interpreting 186
mod notation and meaning xxix
modification level of ESSL, getting 1193
modified Givens transformations 254
modifying
C programs, for using ESSL for Linux 171
modifying (continued)
  Fortran programs, for using ESSL 170
modulo notation xxix
multiplying
  notation xxix

name usage restrictions 47
names of
  products and acronyms xxiv
  the eigensystem analysis subroutines 847
  the Fourier transform, convolution and correlation, and
  related-computation subroutines 953
  the interpolation subroutines 1125
  the linear algebra subprograms 197
  the linear algebraic equations subroutines 467
  the matrix operations subroutines 383
  the numerical quadrature subroutines 1143
  the random number generation subroutines 1165
  the sorting and searching subroutines 1109
  the utility subroutines 1187
negative definite complex Hermitian matrix
  definition of 85
negative definite complex Hermitian Toeplitz matrix
  definition of 87
negative definite Hermitian matrix
  storage layout 85
negative definite symmetric matrix
  definition of 84
  storage layout 84
negative definite symmetric Toeplitz matrix
  definition of 86
negative stride, for vectors 75
netlib-java 1243
non-IBM library, migrating from 181
norm notation xxix
normally distributed pseudo-random numbers, generate
  1173
notations and conventions xxvi
notes, conventions used in the subroutine descriptions xxxiii
number of subroutines in each area 2
numbers
  accuracy of computations 61
  accuracy of computations, for ESSL 4
numerical quadrature
  accuracy considerations 1143
  performance considerations 1143
  programming considerations for SUBF 1144
  usage considerations 1143
numerical quadrature performed
  on a function
    using Gauss-Hermite Quadrature 1161
    using Gauss-Laguerre Quadrature 1155
    using Gauss-Legendre Quadrature 1147
    using Gauss-Rational Quadrature 1158
    using two-dimensional Gauss-Legendre Quadrature 1149
numerical quadrature subroutines
  SGHMQ and DGHMQ 1161
  SGLGQ and DGLGQ 1155
  SGLNQ and DGLNQ 1147
  SGLNQ2 and DGLNQ2 1149
  SGRAQ and DGRAQ 1158

O
one norm notation xxix
option table, default values for ESSL errors 67
ordering numbers of the publications 1253
ordering IBM publications 1253
output
  accuracy on different processors 4
  how errors affect output 64
  output arguments, conventions used in the subroutine
  descriptions xxxiii
overflow, avoiding 238
overview
  of eigensystem analysis 847
  of Fourier transforms, convolutions and correlations,
  and related computations 953
  of interpolation 1125
  of linear algebra subprograms 197
  of linear algebraic equations 467
  of matrix operations 383
  of numerical quadrature 1143
  of random number generation 1165
  of sorting and searching 1109
  of the documentation 1253
  of utility subroutines 1187

P
packed band storage mode 95
packed-symmetric-Toeplitz storage mode 86
parallel processing
  introduction to 2
performance
  achieving better performance in your program 62
  aspects of parallel processing on the workstations 2
  coding techniques that affect performance 62
  considerations for dense and banded linear algebraic
  equations 477
  considerations for eigensystem analysis 849
  considerations for Fourier transforms, convolutions, and
  correlations 958
  considerations for interpolation 1125
  considerations for linear algebra subprograms 203
  considerations for matrix operations 385
  considerations for numerical quadrature 1143
  considerations for related computations 961
  considerations for sorting and searching 1109
  how the Fourier transforms achieve high performance
  960
  information on ESSL run-time performance 63
  tradeoffs for convolution and correlation subroutines
  960
  how to find information on 63
pi notation xxix
PL/I (Programming Language/I)
  handling errors in your program 147, 161
planning your program 33
positive definite Complex hermitian band matrix
  definition of 102
storage layout 102
positive definite complex Hermitian matrix
  definition of 85
positive definite complex Hermitian Toeplitz matrix
  definition of 87
positive definite complex Hermitian tridiagonal matrix
  definition of 110
  storage layout 110
positive definite Hermitian matrix
  storage layout 85
positive definite symmetric band matrix
  definition of 101
  storage layout 101
positive definite symmetric band matrix subroutines, names of 467
positive definite symmetric matrix
  definition of 84
  storage layout 84
positive definite symmetric matrix subroutines, names of 467
positive definite symmetric Toeplitz matrix
  definition of 86
positive definite symmetric tridiagonal matrix
  definition of 109
  storage layout 109
positive stride, for vectors 73
precision, meaning of 61
precision, short and long 4
problems, handling 183
problems, IBM support for 183
processing your program
  using parallel subroutines on the workstations 2
processor-independent formulas for auxiliary storage, interpreting 51
product
  matrix-matrix
    complex Hermitian matrix 422
    complex symmetric matrix 422
    real symmetric matrix 422
    triangular matrix 430
  matrix-vector
    complex Hermitian matrix 313
    real symmetric matrix 313
    triangular band matrix, its transpose, or its conjugate transpose 360
    triangular matrix, its transpose, or its conjugate transpose 347
product names, acronyms for xxiv
products, programming
  migrating from LAPACK 180
  required by ESSL for Linux on the workstations, programming 6
program
  attention messages, overview 67
  coding 125
  computational errors 65
  design 33
  errors 64
  handling errors in your C program 147
  handling errors in your Fortran program 133
  input-argument errors, overview 64
  interface for C programs 142
  interface for C++ programs 155
  interface for Fortran programs 125
  migrated to ESSL 179
  performance, achieving high 62
  resource errors, overview 66
  setting up your data structures 71
  types of data in your program 47
program (continued)
  when coding large applications 69
program exceptions
  description of ESSL 64
program exceptions, diagnosing 184
program number for ESSL 1253
programming considerations for SUBF in numerical quadrature 1144
programming items, font for xxv
programming products
  required by ESSL on the workstations 6
programming publications 1253
PTF
  getting the most recent level applied 1193
publications
  list of ESSL 1253
  math background 1253
  related 1253
Python 1243
Q
QR factorization 812, 822
quadratic interpolation 29
R
random number generation
  normally distributed 1173
  usage considerations 1165
random number generation subroutines
  SNRNG and DNRNG 1173
rank-2k update
  complex Hermitian matrix 451
  complex symmetric matrix 451
  real symmetric matrix 451
rank-k update
  complex Hermitian matrix 445
  complex symmetric matrix 445
  real symmetric matrix 445
rank-two update
  complex Hermitian matrix 328
  real symmetric matrix 328
real and complex array elements 126
real data
  conventions xxvi, 47
real general matrix eigensystem analysis subroutine 847
real symmetric matrix eigensystem analysis subroutine 847
reference information
  math background texts and reports 1253
  what is in each subroutine description and the conventions used xxxi
related publications 1253
related-computation subroutines
  accuracy considerations 961
  CWLEV and ZWLEV 1102
  performance considerations 961
  SWLEV and DWLEV 1102
release of ESSL, getting 1193
reporting problems to IBM 183
required publications 1253
requirements
  auxiliary working storage 51
requirements (continued)

software products on the workstations 6
transforms in storage, lengths of 56
resource error messages, interpreting 185
restrictions, ESSL coding 47
results
accuracy on different processors 4
how accuracy is affected by the nature of the computation 61
in C programs 142
in C++ programs 155
in Fortran programs 125
multiplication of NaN 61
results transposed and conjugate transposed for matrix multiplication 402, 412, 418
results transposed for matrix addition 388
results transposed for matrix subtraction 394
return code
in C programs 147
in C++ programs 161
in Fortran programs 134
using during diagnosis 185
routine names 47
run-time performance
optimizing in your program 62
run-time problems, diagnosing
attention error messages, interpreting 186
computational errors 185
informational error messages, interpreting 186
input-argument errors 184
miscellaneous error messages, interpreting 186
resource error messages, interpreting 185
running your program
C programs 171
Fortran programs 170

S

SACOR 1083
SASUM 214
scalar data
conventions xxvi, 47
scalar items, font for xxv
scale argument used for Fourier transform subroutines 959
scaling, when to use 62
SCASUM 214
SCNRM2 238
SCON 1060
SCOR 1060
SCOSFT, no documentation provided for 953
SDOT 225
SDOTI 292
selecting an ESSL library 33
selecting an ESSL subroutine 33
sequences
conventions for xxvii
description of 120
storage layout 121
service, IBM 183
set the number of GPUs 1194
SETGPU
set the number of GPUs 1194
setting up your data 47
SGELS 812
SGELSD 822
SGEQRF 807
SGESM 499
SGESUB 392
SGETMI 459
SGETRF 484
SGHMQ 1161
SGLGQ 1155
SGLNQ 1147
SGLNQ2 1149
SGRAQ 1158
short precision
accuracy statement 4
meaning of 61
SIGN notation and meaning xxix
signal processing subroutines 955
simple formulas for auxiliary storage, interpreting 51
sin notation xxix
size of array
required for a vector 72
skyline solvers
usage considerations 478
SLAMCH 1210
SLANGE 518
SLANSP 579
SLANSY 579
SLANTP 628
SLANTR 628
SLSS (Subscription Library Services System) 1253
SMP
performance 4
SNDOT 233
SNORM2 241
SNRM2 238
SNRNG 1173
software products
required by ESSL for Linux on the workstations 6
solve
indefinite
complex Hermitian matrix 601
complex symmetric matrix 601
real symmetric matrix 601
solving
general matrix 518, 555, 579, 628
general tridiagonal matrix 687
multiple right-hand sides
general matrix, its transpose, or its conjugate transpose 499
positive definite complex Hermitian matrix 545
positive definite real symmetric matrix 545
triangular matrix 437
triangular band matrix 365
triangular matrix 354
tridiagonal matrix multiple right-hand side solve 679
sorting and searching subroutines
accuracy considerations 1109
performance considerations 1109
usage considerations 1109
sparse linear algebraic equation subroutines 467
sparse matrix subroutines
direct solvers 477
iterative linear system solvers 479
performance and accuracy considerations 477–479
sparse matrix subroutines (continued)
skyline solvers 478
sparse vector-scalar linear algebra subprograms
SDOTI, DDOTI, CDOTUI, ZDOTUI, CDOTCI, and, ZDOTCI
292
sparse vector, definition and storage modes 75
special usage
of matrix addition 388
of matrix multiplication 402, 412, 418
of matrix subtraction 394
spectral norm notation xxix
SPOCON 555
SPOSM 545
SPOTRS 545
SPPSV 521
SPPTR5 545
SPTF 716
SPTSV 679
SPTTRF 687
SPTTTR 687
square root notation xxix
SRCFT2 1028
SROTMG 254
SSLMX 313
SSLR2 328
SSPEVX 867
SSPGVX 931
SSPMV 313
SSPR2 328
SSYEVX 867
SSYGV 931
SSYGV2 922
SSYGVX 931
SSYMM 422
SSYMV 313
SSYR 328
SSYRK 451
STRMV 347
STSMV 360
STPMMV 360
STBSV 365
stepping through storage, for matrices 78
stepping through storage, for vectors 73
storage
array storage techniques overview 47
auxiliary working storage requirements 51
considerations when designing your program 47
for matrices 78
for vectors 72
layout for a complex Hermitian matrix 84
layout for a complex Hermitian tridiagonal matrix 109
layout for a general tridiagonal matrix 106
layout for an indefinite complex Hermitian matrix 86
layout for an indefinite symmetric matrix 84
layout for a negative definite Hermitian matrix 85
layout for a negative definite symmetric matrix 84
layout for a positive definite complex Hermitian tridiagonal matrix 110
layout for a positive definite Hermitian matrix 85
layout for a positive definite symmetric matrix 84
layout for a positive definite symmetric tridiagonal matrix 109
layout for a sequence 121, 123
layout for a symmetric tridiagonal matrix 108
storage (continued)
layout for a Toeplitz matrix 86
layout for band matrices 95–97
layout for positive definite symmetric band matrices 101
layout for sparse matrices 110
layout for sparse vectors 76
layout for symmetric band matrices 99
layout for symmetric matrices 80
layout for trapezoidal matrices 92
layout for triangular band matrices 104
layout for triangular matrices 89
list of subroutines using auxiliary storage 49
list of subroutines using transforms 56
of arrays in Fortran 126
positive definite Complex hermitian band matrix 102
storage-by-columns for sparse matrices 114
storage-by-columns for sparse matrices 114
storage-by-indices for sparse matrices 114
storage-by-indices for sparse matrices 114
tradeoffs for input 62
transform length requirements 56
storage conversion subroutine
symmetric skyline sparse matrix 1219
storage-by-columns for sparse matrices 114
storage-by-columns for sparse matrices 114
storage-by-rows for sparse matrices 115
STRMM 430
STRMV 347
STRSM 437
STRSV 354
STRTRI 621
stride
defining vectors in arrays 73
how it is used in three dimensions 123
negative 75
optimizing for your Fourier transforms 959
positive 73
zero 74
subject code for ESSL documentation 1253
subprogram
linear algebra 197
meaning of xxiv, 197
subprogram, definition xxiv
subroutine
calling sequence format for C programs 142
calling sequence format for C++ programs 155
calling sequence format for Fortran programs 125
choose of 33
conventions used in the description of xxxi
overview of ESSL 2
subroutine, definition xxiv
subroutines, ESSL
CDOTC 225
CDOTCI 292
CDOTU 225
CDOTUI 292
CGECMI 459
CGEQRF 807
CGESM 499
CGESUB 392
CGETMI 459
subroutines, ESSL (continued)

<table>
<thead>
<tr>
<th>Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGETRF 484</td>
</tr>
<tr>
<td>CGTSV 666</td>
</tr>
<tr>
<td>CHEEV 867</td>
</tr>
<tr>
<td>CHEEVR 867</td>
</tr>
<tr>
<td>CHEEVX 867</td>
</tr>
<tr>
<td>CHEGV 931</td>
</tr>
<tr>
<td>CHEGVD 922</td>
</tr>
<tr>
<td>CHEGVX 931</td>
</tr>
<tr>
<td>CHEMM 422</td>
</tr>
<tr>
<td>CHEMV 313</td>
</tr>
<tr>
<td>CHER 328</td>
</tr>
<tr>
<td>CHER2 451</td>
</tr>
<tr>
<td>CHERK 445</td>
</tr>
<tr>
<td>CHESV 584</td>
</tr>
<tr>
<td>CHETRF 593</td>
</tr>
<tr>
<td>CHETRS 601</td>
</tr>
<tr>
<td>CHPEVX 867</td>
</tr>
<tr>
<td>CHPGVX 931</td>
</tr>
<tr>
<td>CHPMV 313</td>
</tr>
<tr>
<td>CHPR 328</td>
</tr>
<tr>
<td>CHPSV 584</td>
</tr>
<tr>
<td>CHPTRF 593</td>
</tr>
<tr>
<td>CHPTRS 601</td>
</tr>
<tr>
<td>CLANGE 518</td>
</tr>
<tr>
<td>CLANHE 579</td>
</tr>
<tr>
<td>CLANH 579</td>
</tr>
<tr>
<td>CLANSY 579</td>
</tr>
<tr>
<td>CLANTR 628</td>
</tr>
<tr>
<td>CNORM2 241</td>
</tr>
<tr>
<td>CPOSM 545</td>
</tr>
<tr>
<td>CPOTRS 545</td>
</tr>
<tr>
<td>CPPSV 521</td>
</tr>
<tr>
<td>CPPT 545</td>
</tr>
<tr>
<td>CPTSV 679</td>
</tr>
<tr>
<td>CPTTRF 684</td>
</tr>
<tr>
<td>CPTTRS 687</td>
</tr>
<tr>
<td>CSPSV 584</td>
</tr>
<tr>
<td>CPSTRF 593</td>
</tr>
<tr>
<td>CPSTRS 601</td>
</tr>
<tr>
<td>CSYMM 422</td>
</tr>
<tr>
<td>CSYR2K 451</td>
</tr>
<tr>
<td>CSYRK 445</td>
</tr>
<tr>
<td>CSYSV 584</td>
</tr>
<tr>
<td>CSYTRF 593</td>
</tr>
<tr>
<td>CSYTRS 601</td>
</tr>
<tr>
<td>CTBMM 360</td>
</tr>
<tr>
<td>CTBSV 365</td>
</tr>
<tr>
<td>CTPMV 347</td>
</tr>
<tr>
<td>CTPSV 354</td>
</tr>
<tr>
<td>CTPTRI 621</td>
</tr>
<tr>
<td>CTRMM 340</td>
</tr>
<tr>
<td>CTRMV 347</td>
</tr>
<tr>
<td>CTRSM 437</td>
</tr>
<tr>
<td>CTRSV 354</td>
</tr>
<tr>
<td>CTRTRI 621</td>
</tr>
<tr>
<td>CWLLEV 1102</td>
</tr>
<tr>
<td>DASUM 214</td>
</tr>
<tr>
<td>DDOT 225</td>
</tr>
<tr>
<td>DDOTI 292</td>
</tr>
<tr>
<td>DGELS 812</td>
</tr>
<tr>
<td>DGELSD 822</td>
</tr>
<tr>
<td>DGESQR 807</td>
</tr>
</tbody>
</table>
subroutines, ESSL (continued)

DWLEV 1102
DZASUM 214
DZNRM2 238
EINFO 1188
ERRSAV 1190
ERRSET 1191
ERRSTR 1192
IESSL 1193
SACOR 1083
SASUM 214
SCASUM 214
SCNRM2 238
SCON 1060
SCOR 1060
SDOT 225
SDTI 292
SETGPUS
1194
SGELS 812
SGELSD 822
SGEQRF 807
SGESM 499
SGESUB 392
GETM1 459
SGETRF 484
SGHQ 1161
SGLQ 1155
SGLQ 1147
SGLQ 1149
SGRAQ 1158
SGTSV 666
SLAMCH 1210
SLANGE 518
SLANSP 579
SLANTP 584
SNORM2 241
SNRM2 238
SNRNG 1173
SPOSM 545
SPOTRS 545
SPPSV 521
SPPTRS 545
SPTRF 716
SPTSV 679
SPTTRF 684
SPTTRF 687
SRCFT2 1028
SRQTMG 254
SSLQ 313
SSLR2 328
SSPEVX 867
SSPGVX 931
SSPMV 313
SSPR2 328
SSPSV 584
SSPTRF 593
SSPTRS 601
SSYEV 867
SSYEVX 867

subroutines, ESSL (continued)

SSYGV 931
SSYGVX 922
SSYGVX 931
SSYMM 422
SSYMV 313
SSYR 328
SSYSV 584
SSYTRF 593
SSYTRS 601
STBMV 360
STBSV 365
STPMV 347
STPSV 354
STPTRI 621
STRMM 430
STRMV 347
STRSM 437
STRSV 354
STRTRI 621
SWLEV 1102
ZDOTC 225
ZDOTC 292
ZDOTU 225
ZDOTU 292
ZGECM1 459
ZGELS 812
ZGELSD 822
ZGGEVX 807
ZGESM 499
ZGESUB 392
ZGETM1 459
ZGETRF 484
ZGTSV 666
ZHEEV 867
ZHEEVR 867
ZHEEVX 867
ZHEG 931
ZHEGV 931
ZHEGVD 922
ZHEGVX 931
ZHEMV 313
ZHER 328
ZHER2K 451
ZHERK 445
ZHESV 584
ZHETRF 593
ZHETRS 601
ZHPM 867
ZHPGVX 931
ZHPMV 313
ZHPMV 313
ZHPR 328
ZHPSV 584
ZHPTRF 593
ZHPTRS 601
ZLAN 518
ZLANHE 579
ZLANHP 579
ZLANSS 579
ZLANTP 628
ZLANTR 628
ZNORM2 241

1273
subroutines, ESSL (continued)

ZPOSM 545
ZPOTRS 545
ZPPSV 521
ZPPTRS 545
ZPTSV 679
ZPTTRF 684
ZPTTRS 687
ZPSV 584
ZPSV 593
ZPSV 601
ZSYMM 422
ZSYRK 451
ZSYRK 445
ZSYSV 584
ZSYTRF 593
ZSYTRS 601
ZTBMV 360
ZTBSV 365
ZTPMV 347
ZTPSV 354
ZTPTRI 621
ZTTRI 621
ZWLEV 1102

subroutines, Parallel ESSL

CPOCON 555
CPPCON 555
DPOCON 555
DPPCON 555
SPOCON 555
SPPCON 555
ZPOCON 555
ZPPCON 555

subscript notation, what it means xxix
subtracting
general matrices or their transposes 392
sum, calculating
absolute values 214
summ xxix
superscript notation, what it means xxix
support, IBM 183
SWLEV 1102
symbols, special usage of xxix
symmetric band matrix
definition of 99
storage layout 99

symmetric matrix
definition of 80
storage layout 80

symmetric tridiagonal matrix
definition of 107
storage layout 108

symmetric-tridiagonal storage mode 108

symmetry, names of products xxiv
textbooks cited 1253
time-varying recursive filter 29
timings, achieving high performance in your program 62

Toeplitz matrix
definition of 86, 87
storage layout 86

traceback map, using during diagnosis 185

transform lengths, calculating 56

transformation
modified Givens 254

transpose
conjugate, of a matrix 77
conjugate, of a vector 72
notation xxix
of a matrix 77, 78
of a matrix inverse notation xxix
of a vector 72
of a vector or matrix notation xxix
of matrix operation results for add 388
of matrix operation results for multiply 402, 412, 418
of matrix operation results for subtract 394

transposing
general matrix (In-Place) 459

trapezoidal matrices
storage layout 92

trapezoidal matrices, upper and lower
definition of 91

triangular band matrices
storage layout 104

triangular band matrices, upper and lower
definition of 102

triangular matrices
storage layout 89

triangular matrices, upper and lower
definition of 88

tridiagonal matrix
definition of 106
storage layout 106

tridiagonal storage mode 106

truncation
how truncation affects output 61

type font usage xxv

U

underflow
avoiding underflow 238
why mask it 62

upper band width 94
upper storage mode 80
upper-packed storage mode 80, 82
upper-storage-by-rows for symmetric sparse matrices 115

upper-trapezoidal storage mode 92

table, default values for ESSL error option 67

termination, program
attention messages 67
computational errors 65
input-argument errors 64
resource errors 66
terminology used for Fourier transforms, convolutions, and correlations 955
terminology, names of products xxiv
textbooks cited 1253

three-dimensional data structures, how stride is used for 123
time-varying recursive filter 29
times notation, multiply xxix
timings, achieving high performance in your program 62

Toeplitz matrix
definition of 86, 87
storage layout 86

traceback map, using during diagnosis 185

transform lengths, calculating 56

transformation
modified Givens 254

transpose
conjugate, of a matrix 77
conjugate, of a vector 72
notation xxix
of a matrix 77, 78
of a matrix inverse notation xxix
of a vector 72
of a vector or matrix notation xxix
of matrix operation results for add 388
of matrix operation results for multiply 402, 412, 418
of matrix operation results for subtract 394

transposing
general matrix (In-Place) 459

trapezoidal matrices
storage layout 92

trapezoidal matrices, upper and lower
definition of 91

triangular band matrices
storage layout 104

triangular band matrices, upper and lower
definition of 102

triangular matrices
storage layout 89

triangular matrices, upper and lower
definition of 88

tridiagonal matrix
definition of 106
storage layout 106

tridiagonal storage mode 106

truncation
how truncation affects output 61

type font usage xxv

U

underflow
avoiding underflow 238
why mask it 62

upper band width 94
upper storage mode 80
upper-packed storage mode 80, 82
upper-storage-by-rows for symmetric sparse matrices 115

upper-trapezoidal storage mode 92

1274
upper-trapezoidal-packed storage mode 92
upper-triangular storage mode 89
upper-triangular-band-packed storage mode 104
upper-triangular-packed storage mode 89
usability of the ESSL subroutines 2
usage considerations
direct sparse matrix solvers 477
for Fourier transforms, convolutions, and correlations 955
for interpolation 1125
for linear algebra subprograms 203
for linear algebraic equations 477
for matrix operations 384
for numerical quadrature 1143
for random number generation 1165
for sorting and searching 1109
for utility subroutines 1187
sparse matrix subroutines (iterative linear system solvers) 479
sparse matrix subroutines (skyline solvers) 478
usage, special
conventions used in the subroutine description xxxiv
for matrix addition 388
for matrix multiplication 402, 412, 418
for matrix subtraction 394
user applications 2
Using ESSL with netlib-java and Python 1243
utility subroutines
DSKTRN 1219
EINFO 1188
ERRSAV 1190
ERRSET 1191
ERRSTR 1192
IESSL 1193
SETGPUS 1194
usage considerations 1187
V
vector
compressed vector 76
conventions for xxvi
font for xcv
full vector 76
number of array elements needed for 72
sparse vector 75
storage of 72
stride for 73
vector-scalar linear algebra subprograms
SASUM, DASUM, SCASUM, and DZASUM 214
SDOT, DDOT, CDOTU, ZDOTU, CDOTC, and ZDOTC 225
SNDOT and DNDOT 233
SNRM2, DNRM2, CNORM2, and ZNORM2 241
SNRM2, DNRM2, SCNRM2, and DZNRMD 238
SROTMG and DROTMG 254
version of ESSL, getting 1193
versions of subroutines 2
W
Wiener-Levinson filter coefficients 1102
Wiener-Levinson filter coefficients subroutine 29
working auxiliary storage, list of subroutines using 49
working storage for band matrix 95
workstations
migrating from one IBM hardware platform to another 180
Z
ZDOTC 225
ZDOTCI 292
ZDOTTU 225
ZDOTUI 292
zero stride, for vectors 74
ZGECMI 459
ZGELS 812
ZGELSD 822
ZGQR 807
ZGESM 499
ZGESUB 392
ZGETMI 459
ZGETRF 484
ZHEEVR 867
ZHEGV 931
ZHEGVX 931
ZHEM 422
ZHEMV 313
ZHER2 328
ZHER2K 451
ZHERK 445
ZHPMV 313
ZHPR2 328
ZLANGE 518
ZLANHE 579
ZLANHP 579
ZLANSY 579
ZLANTP 628
ZLANTR 628
ZNORM2 241
ZPOSM 545
ZPT 545
ZPPCON 555
ZPPSV 521
ZPTRS 545
ZPTSV 679
ZPTTRF 684
ZPTTRS 687
ZSYMM 422
ZSYR2K 451
ZSYRK 445
ZTBMV 360
ZTBSV 365
ZTPMV 347
ZTPSV 354
ZTPT 621
ZTRMM 430
ZTRMV 347
ZTRSM 437
ZTRSV 354
ZTRT 621
ZWLEV 1102